A Cognitive Model and Gaze-Based Evaluation of Multiple Representation use during Program Comprehension and Debugging

Prateek Hejmady

A Thesis Submitted to the Graduate Faculty of
Auburn University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

Auburn, Alabama
December 12, 2011

Keywords: cognitive modeling, eye-tracking, program comprehension, program debugging, psychology of programming, visualizations

Approved by

N. Hari Narayanan, Chair, Professor, Computer Science and Software Engineering
James H. Cross, Professor, Computer Science and Software Engineering
Dean T. Hendrix, Associate Professor, Computer Science and Software Engineering
Margaret E. Ross, Professor, Department of Educational Foundations, Leadership, and Technology

ABSTRACT

Integrated Development Environments (IDE) for software development offer a wide variety of software visualization tools, which facilitate navigation and analysis, and generate multiple graphical presentations of program code. Co-ordination of these representations during program comprehension can be a complex task for a novice programmer, and at times be detrimental to debugging performance. This thesis develops a cognitive model of how multiple representations including visualizations are used by programmers to comprehend and debug a program in an IDE for object oriented programming. The model, based on literature review and analyses of the shortcomings of existing research, is more detailed than any model of program comprehension and debugging hitherto offered in the literature. The model was evaluated empirically with two debugging studies during which visual attention of participants was tracked with an eye-tracker. The first study found that a mental model created by static visualizations is not as extensive as the mental model created by dynamic visualizations. Mental model strength of programming constructs like data structure and function was consistently higher for dynamic visualizations, whereas strength of control flow and data flow was consistently stronger for static visualizations. On analyzing visual attention patterns during the second study, we found that program code and dynamic representations (viewer, variable watch and output) attracted the most attention. Static representations like UML and Control Structure diagrams saw significantly lesser usage. Gaze patterns were analyzed by breaking down the debugging sessions into segments of three, five and fifteen minute intervals, and classifying gaze durations as short and long gazes.

Data mining techniques were used to detect high frequency patterns from eye tracking data of participants. A significant pattern difference was found among the participants based on programming experience, familiarity with the IDE and debugging performance. These results are consistent with the proposed cognitive model and open up many more intriguing questions for future research.

Acknowledgements

There are many key individuals who have been instrumental in bringing this work to fruition. First of all, I would like to thank my advisor Dr. Hari Narayanan, for letting me chose my own path of research and most importantly guiding and mentoring me along the way. It has been a wonderful experience working with him and learning the ways of academia and academic research. Wonder what I would have done, without Dr. Dean Hendrix. My go to guy for advice ranging from composing strategic emails, applying for jobs to research guidance. I'm privileged to have him on my committee. I owe much gratitude to Dr. James Cross II and Dr. Margaret Ross for agreeing to be part of my thesis advisory committee and for giving me valuable advice during the course of this research.

Heartfelt thanks to Dr. Francisco (Pako) Arcediano, for his valuable advice ranging from nitty-gritty to convoluted matters of eye-tracking. Without his generous gesture of letting me use his state of the art eye-tracker for my experiments, this thesis would have been hard to come by.

A special thanks to all my friends who have made life worth living by sharing both remorse and joy. Harish, Gautham, Puneet & Vasavi, I sincerely appreciate your patience with me not returning your calls! I can only dread the feeling of not having your consistent encouragement and backing of my endeavors.

Most of all I would like to thank my parents, Poornima C. and Chandrashekar P. and brother Prakhyat, who have always afforded me the freedom to follow my own path. I know that they miss me a lot and would rather have me home, but they still encourage me to do what I want. Love you guys!

I dedicate this work to my grandfather, Sundar Hejmady who taught me that scientific knowledge is best used in the pursuit of a better world.

TABLE OF CONTENTS

LIST OF FIGURES

List of Tables

CHAPTER 1

INTRODUCTION

Most of the programming today is accomplished on sophisticated software applications called Integrated Development Environments (IDE). IDEs are extremely popular among programmers; primarily due to the increase in productivity when used for software development. These assist a programmer by providing a plethora of functionalities like source code editor, compiler/interpreter, build tools, debugger, version control system, etc. Several of these functionalities present multiple perspectives of the same program under development. These representations, also known as program visualizations, enable programmers to treat programs not just as code text, but as program entities produced when executed under different conditions. Program visualizations presented by IDEs range from either graphical to mostly textual and present different types of information about the program. A good example would be simultaneous use of both UML (Unified Modeling Language) diagram and control flow diagram by a programmer to grasp different perspectives of the same software project. A programmer uses these visualizations when appropriate to comprehend or debug a program and build up a mental model of the program. Usage of these functionalities differs from one programmer to another based on factors like programming language expertise, acquaintance with the IDE and personal preference.

Although these tools are designed to facilitate programming activities, they tend to be overwhelming and at times detrimental to a programmers' performance. Thus, effective

usage of these visualizations require a programmer to be skilled in: a) generating and testing hypotheses from the evidence in a program's output and visualizations, and; b) combining this strategic knowledge with his/her knowledge of coordinating appropriate visualizations and functionalities of the IDE. Novice programmers using IDEs face the additional challenge of having to learn abstract concepts of programming as well as these IDE usage skills. It is therefore beneficial to develop insights into the underlying processes at work during program comprehension/debugging in a rich software development environment, in order to help us better understand the effectiveness of existing IDEs and design better IDE interfaces in future.

Program comprehension, the ability to understand programs written by others, is widely recognized as central to programming. Previous research in the domain has established a solid body of knowledge about comprehension models and strategies employed in comprehension, expert novice differences, and comprehension outcome analysis. Majority of the research has however focused on using potentially intrusive verbal protocols to capture thought processing instead of applying a non invasive methodology like eye-tracking. Recently though, visual attention tracking is increasingly used by researchers, especially those studying the psychology of programmers. This methodology was first employed to investigate how programmers read program code. Other recent studies investigated program comprehension or debugging employing either a visual attention tracking tool called Restricted Focus Viewer (RFV) or an eye-tracker. The present research involves the most recent eye-tracking study of programmers.

The core of this thesis is an investigation, from theoretical and empirical perspectives, of the underlying cognitive processes active during programming tasks. We first develop a theoretical cognitive model of program comprehension and debugging by synthesizing existing research in the areas of text comprehension, comprehension of

picture and diagrams, graph comprehension and program comprehension. Then an empirical study of programmers was designed and carried out to explore processes of program comprehension and debugging, and to answer some of the research questions arising out of the proposed theoretical model. In chapter 2, we discuss relevant literature in areas pertaining to our research. Chapter 3 summarizes the problem statement of this research. The overall research and its results are presented in chapters 4, 5 and 6. A summary of research contributions and future work are presented in Chapter 7 and 8 respectively.

CHAPTER 2

BACKGROUND

This chapter presents the background material and related work relevant to this thesis. It specifically addresses the motivation behind our research, and the history and current state of the topics pertaining to our research: cognitive modeling – text, picture, multimedia and program comprehension, program debugging, program visualization and eye-tracking.

2.1 Cognitive Modeling

Cognitive science is concerned with understanding the processes that the human brain uses to accomplish complex tasks including perceiving, learning, remembering, thinking, predicting, inference, problem solving, decision making, planning, and moving around the environment. The goal of a cognitive model is to scientifically explain one or more of these basic cognitive processes, or explain how these processes interact (Busemeyer & Diederich, 2010). They help reveal information pertaining to cognitive and perceptual constraints on human performance. These models now appear in many fields that deal with cognition, ranging from perception to problem solving and decision making. Descriptions of cognitive models take various forms such as narrations of steps required in completion of a task and computer based simulations embodying cognitive architectures.

Cognitive models often incorporate mental models, which according to Johnson-Laird's theory (Johnson-Laird, 1983), is the basic structure of cognition: "It is now plausible to suppose that mental models play a central and unifying role in representing objects, states of affairs, sequences of events, the way the world is, and the social and psychological actions of daily life". Mental models are simplified versions of a complex scenario created in the working memory, which are much easier to conceive, interpret and help predict actions. Mental models can be constructed based on perception, comprehension, or imagination. These models help researchers evaluate how decisions are made, how deductive reasoning problems are solved and measure behavior in diverse environments.

We will now discuss some of the cognitive models proposed and studied in the areas of text comprehension, graph and picture comprehension, program comprehension and Human Computer Interaction (HCI).

2.1.1 Text Comprehension

Work on text comprehension is relevant to our research because reading and understanding code is an important activity in program comprehension. For many decades now, text comprehension has been one of the most researched areas in cognitive psychology. It is a complex interactive cognitive process that involves construction of logical representations and inferences at several levels of text and context within the limits of working memory. In general, comprehension is supported by cognitive resources such as working memory (Just & Carpenter, 1992) and inhibitory control (Gernsbacher, Varner, & Faust, 1990). Working memory serves as a mental workspace where information retrieved from memory (either world knowledge or previously read text) is available for integration with incoming text or contributes to updating and revision of the

mental representation of the unfolding text or discourse. Problems can arise because working memory resources are limited or become overloaded if suppression or inhibitory controls are lacking, preventing accurate and efficient integration.

Of late, researchers' focus has shifted from lower levels of comprehension (like lexical processing, interpretation of text, semantics and syntactic parsing) to higher levels of comprehension (involving pragmatics, knowledge-based inferences, world knowledge and problem solving). Of particular interest to us is the process involved during problem solving. There has been an emergence of sophisticated theoretical cognitive models in problem solving with empirical support. We will now discuss few of the relevant models.

In their early work, Kintsch & Van Dijk (1975) proposed that readers generate a variety of knowledge-based inferences when they comprehend stories. Since then, many studies have empirically found that "multiple levels of representation are involved in making meaning" of text (Van Oostendorp & Goldman, 1998). Knowledge-based inferences are constructed whenever knowledge structures from long-term memory are activated and incorporated into the meaning representation of the text. The meaning representation consists of the text base and the referential situation model of the text. The text base represents the meaning of the text, that is, the semantic structure of the text, and it "consists of those elements and relations that are directly derived from the text itself [...] without adding anything that is not explicitly specified in the text" (Van Oostendorp & Goldman, 1998). Whereas, the referential situation model is a life like mental representation of the people, setting, actions, goals and events that are either explicitly mentioned or inferentially suggested by the text. In later research (Graesser et al. 1997 and Kintsch, 1998), additional levels of representation like surface component, communication level and genre level have been established. Surface representation includes the detailed linguistic information, such as specific phrases, words and syntactic

structures but not their meaning. The communication level represents the pragmatic level of communication between reader and writer. The genre level represents the knowledge of the class of text and its corresponding text function.

The model of Trabasso & Van den Broek (1985) took a different stance and assumes that text comprehension is a problem solving process. According to the model, the meaning of a narrative text is represented in long-term memory as a network. The nodes of this network represent the individual clauses of the text, whereas the links represent causal and enabling relations among those clauses. A reader's ability to discover the causal connections is related with comprehension. Understanding an individual clause requires that the reader discovers its causal antecedents and consequences. Understanding the text as a whole requires that the reader finds a causal path that links its opening to its final outcome.

It is worth mentioning the Construction-Integration Model (Kintsch, 1988) here as it holds significant relevance in explaining the role of knowledge in overall comprehension process. The Construction-Integration Model (CI) emphasizes bottom-up, data-driven comprehension processes over more rigid top-down search strategies, common in the area of discourse comprehension. The CI Model is comprised of two ordered steps: knowledge Construction and knowledge Integration. During the Construction step relying on Linguistic Representation, a larger set of mental elements is generated when compared to the traditional methods. The result of this Construction step is a Propositional Network. If this network is influenced by the comprehender's own knowledge bases during the Construction step, due to past experience, then in effect what is produced is an Elaborated Propositional Network.

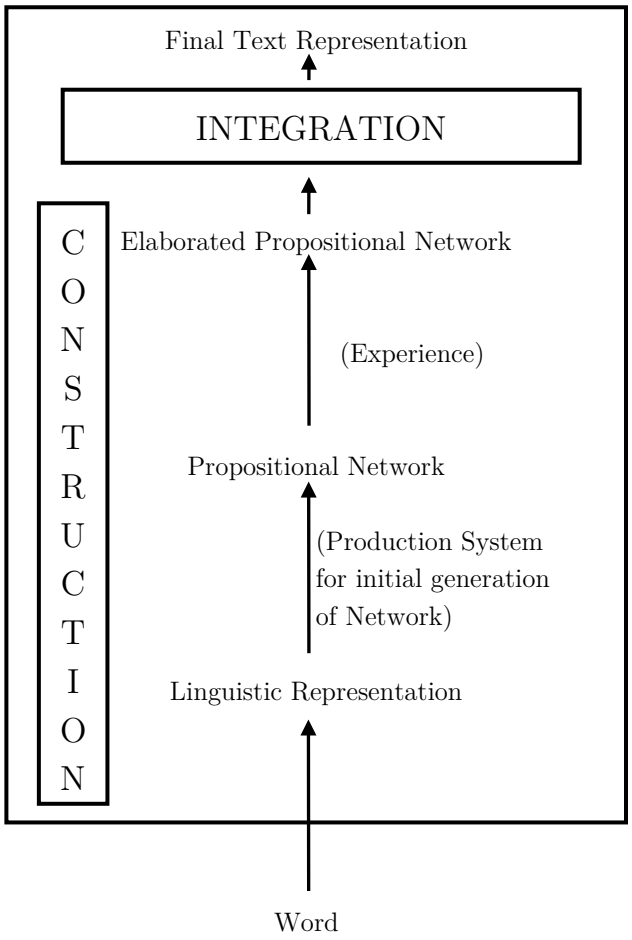


Figure 1. Construction-Integration model

At this point, we have a crude mental representation of the discourse in the form of an associative network of propositional nodes. During the integration stage the other possible meanings of the given sentence are constrained, which do not fit in with the context and strengthens the meanings that do. The overall result is a Final Text Representation which can then be interpreted and evaluated. The intended scope of the CI Model is somewhat limited. It does not concern itself with all problems within discourse comprehension. For instance, it does not concern itself with any specific strategies (or rules) for proposition construction. Rather, it is assumed that a text parser could be designed to do the necessary analyses and then added as a front-end for the CI

Model. Similarly, the model neglects the perceptual aspects of reading a text (or listening), as well as the issue of how the semantic representation of a text is constructed. In sum, the emphasis of the CI Model is on finding a coherent representation and using it, not on how the needed information is generated.

Learning strategy refers to learner's activity during learning aimed at improving learning outcomes. In 1996, an interesting model was proposed for learning strategies in text comprehension by Mayer (1996). This model talked about cognitive process like selecting, organizing and integration; it is known as the SOI model. This model will be elaborated in chapter 4, where it is of higher relevance. We will now discuss relevant work in the area of Picture and Text comprehension.

2.1.2 Picture and Text Comprehension

In a multi-visualization programming environment, a programmer is presented with many static visualizations of the program code along with descriptions. For example, a UML diagram represents the relationships among classes along with text descriptions of these relationships. It is hence of importance to understand the cognitive processes active during picture and text comprehension. While text comprehension has seen intense research over the past three decades, research on comprehension of visual displays has attracted much lesser attention. Earlier research in the area looked at the function of pictures with text. It was found that text supplemented with illustrations led to better retentions than text without illustrations (Levie & Lentz, 1982). Further work in the area also found that carefully constructed pictures as visual text adjuncts also facilitated representation, organization, interpretation and mnemonic encoding (Carney & Levin, 2002). Like text comprehension, during picture comprehension too an individual constructs several mental representations. These include surface structure representation,

a mental model, a propositional representation as well as a communication level and genre level representation. Before we discuss some of the proposed models in the area, it is of utmost importance to understand some of the assumptions (summarized in Table 2.1) about how the human mind works based on research in cognitive science. The first assumption is that the human information processing system consists of two separate channels. This is known as the dual channel assumption. The visual/pictorial channel processes visual input and pictorial representations whereas the auditory/verbal channel processes auditory input and verbal representations. The second assumption is that the information processing channel has a limited cognitive processing capacity. This is known as the cognitive load theory or working memory theory. Hence, only a limited amount of processing takes place in each of the channels. The third assumption is that meaningful learning requires substantial amount of cognitive processing in visual and verbal channels. These assumptions hold ground in not just picture and text comprehension but also in multimedia learning.

Assumption	**Definition**
Dual Channel	Humans possess separate information processing channels for verbal and visual material.
Limited Capacity	There is only a limited amount of processing capacity available in the verbal and visual channels.
Active Processing	Learning requires substantial cognitive processing in the verbal and visual channels.

Table 2.1. Three assumptions about how the mind works (Mayer & Moreno, 2003)

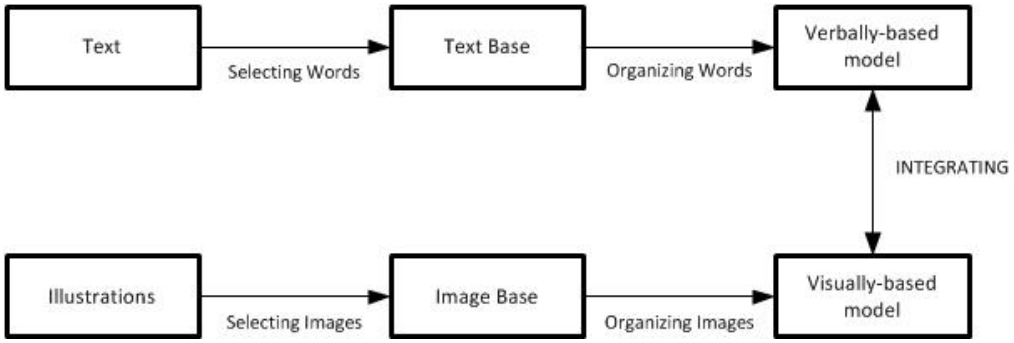


Figure 2. Two representational channels in text and picture comprehension

Mayer (1997) proposed a model (see Figure 2) where verbal and pictorial information are processed in different cognitive subsystems leading to parallel construction of two different mental models, which are finally mapped onto each other. According to this model, while comprehending text with picture, an individual first selects relevant words, constructs a text base, and then organizes the selected verbal information into a verbal based mental model. Likewise, relevant images are selected to form an image base followed by organization of this selected pictorial information into a visual mental model of the picture. During the final stage, one to one mappings are established between the verbal and the visual model. Integration takes place when both verbal and visual models are present in the working memory.

With an emphasis on representational principles, Schnotz and Bannert (2003), proposed an integrated model of text and picture comprehension (Figure 3). This model comprises two branches, with the left representing text comprehension and the right representing picture comprehension. Text comprehension components interact among themselves based on symbol processing. The text comprehension components consist of text as input, mental representation of text's surface structure and the propositional representation of text's semantic content. Text information is processed with regard

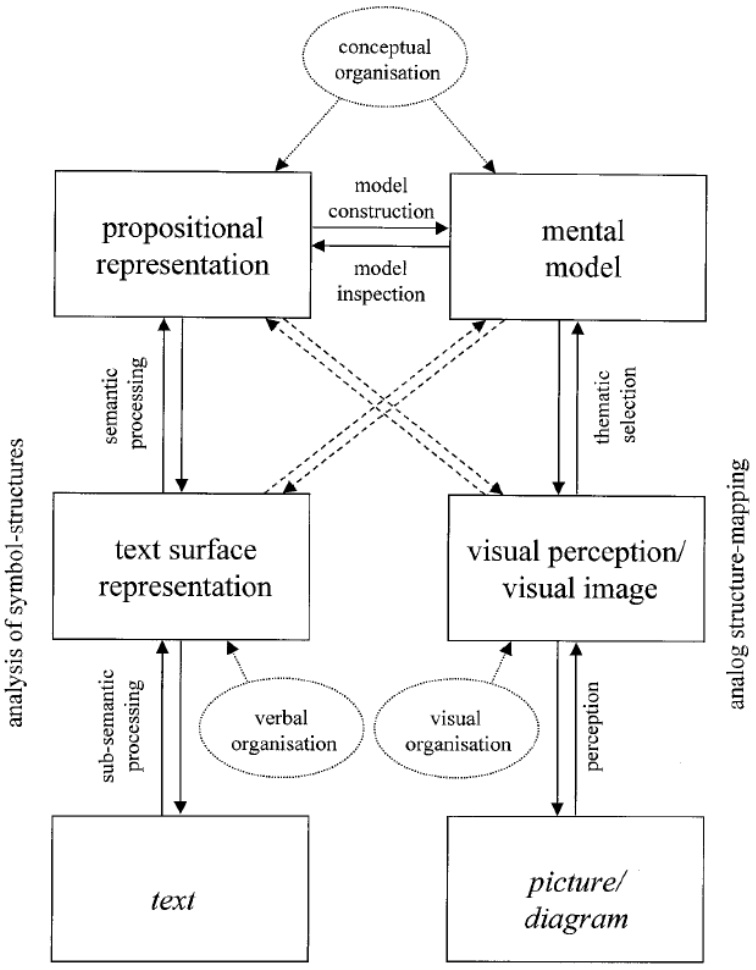


Figure 3. Integrated model of text and picture comprehension

to morphologic and syntactic aspects by verbal organization processes that lead to a mental representation of the text surface structure. This text surface structure in turn triggers conceptual organization processes that result in a structured propositional representation and eventually a mental model. The components for picture comprehension are the external picture, visual perception of the image and a mental model of the picture presented. During picture comprehension, the individual first creates through perceptual processing a visual mental representation of the picture's graphic display. Then, the

individual constructs through semantic processing a mental model and a propositional representation of the subject matter shown in the picture. When a mental model has been constructed, new information can be read from the model through a process of model inspection. There is a continuous interaction between the propositional representation and the mental model. Besides this interaction, there may also be an interaction between the text surface representation and the mental model, and between the perceptual representation of the picture and the propositional representation. We will now discuss in detail multimedia comprehension, which is related to and derives from picture and text comprehension research.

2.1.3 Multimedia Comprehension

It is of importance to us to understand the cognitive process active during multimedia comprehension, as IDEs present dynamic visualizations that are highly graphical and animations, as well as static graphics (e.g., UML diagrams, Control Structure Diagrams, etc.) and program code in the form of text.

Multimedia can be defined in multiple ways depending on the perspective. In terms of presentation, it means the use of different formats of text and pictures. From a sensory modality perspective, it refers to the use of eye and ear to retrieve information. In order to better understand learning from pictures and words (both auditory and printed text), Mayer (2001) proposed a cognitive model of multimedia learning (see Figure 4). In this model, the five columns represent modes of representations. The columns from left to right portray physical representations, sensory representations, shallow working memory representations, deep working memory representations and long-term memory representations. The two rows represent two information processing channels, with the auditory channel at the top and the visual channel below it.

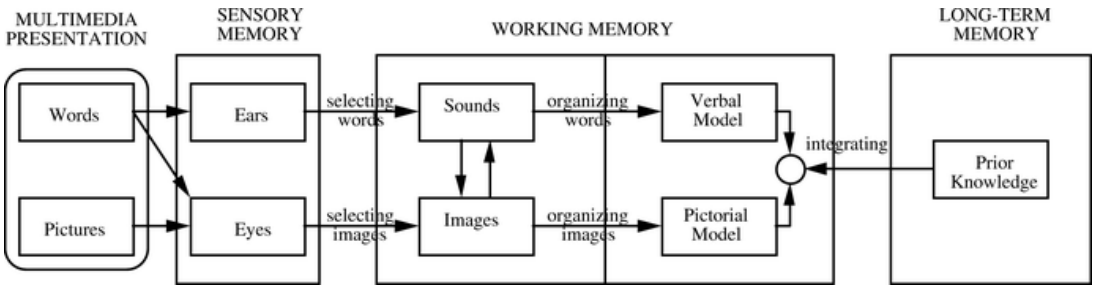


Figure 4. Cognitive theory of multimedia learning

There is virtually no restriction on the capacity for presenting physical representations and long term memory, but the capacity is limited for working memory to process words and images. The arrows from words to eyes and ears represent printed text registered in the eyes; and auditory text registered in the ears. The learner selects some of the incoming auditory sensations and likewise selects (pays attention to) some of the visual sensations coming in from his eyes. Following this, the learner constructs a coherent verbal and pictorial model during organization. Finally, this verbal model, pictorial model and relevant prior knowledge are merged to during integration. Prior knowledge can aid both the selecting and guiding processes in the working memory. Hence, in multimedia learning, active processing that places a high demand on cognitive capacity requires five cognitive processes: selecting images, selecting words, organizing images, organizing words, and integrating.

2.1.4 Program Comprehension

Many models of program comprehension have been proposed by researchers over the past 25 years. In the context of program comprehension, a mental model represents a programmer's mental representation of the program to be understood, and the cognitive

model describes the cognitive processes and information structures involved in the formation of this mental model. Most of the models in the literature consider program comprehension as either bottom-up, top-down or knowledge based understanding. Some models suggest systematic and as-needed strategies. Bottom up theories propose that program knowledge is built by reading the source code and then mentally chunking or grouping these statements into higher level abstractions. Higher order understanding of the program is then constructed by combining these abstractions (Shneiderman & Mayer, 1979). Shneiderman and Mayer (1979) proposed a cognitive framework (Figure 5) incorporating semantic and syntactic knowledge of programs. The internal semantic representation is created by chunking the program in short term memory.

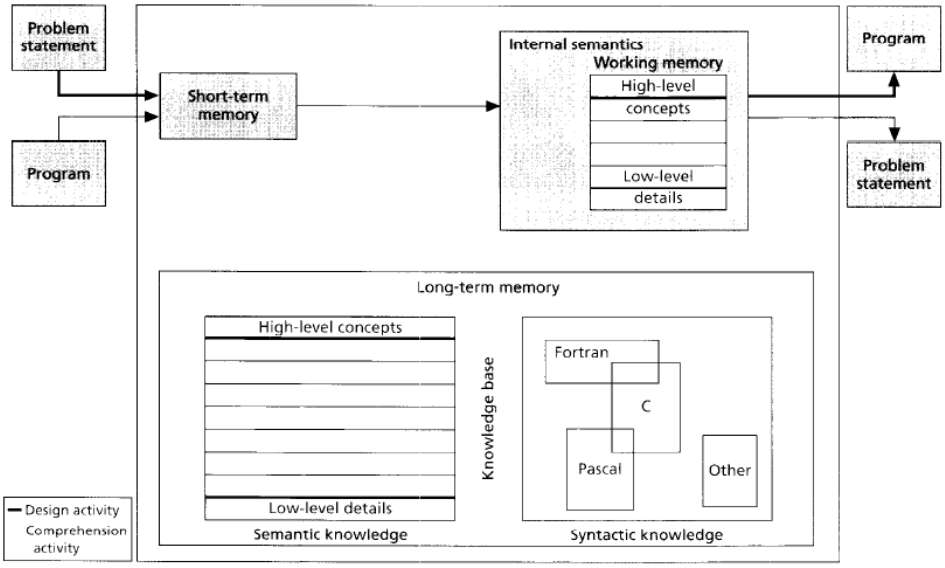


Figure 5. Shneiderman and Mayer program comprehension model

This representation is language independent and is built in progressive layers consisting of high level concepts like program goals at the top and low level details like algorithms used at the bottom. The semantic representation in long term memory assists

the creation of internal semantics. The syntactic knowledge represents the statements and basic units of the program and hence is language dependent. The final mental model is created by chunking and aggregation of other semantic components and syntactic fragments of text. This framework took a bottom-up approach to program comprehension.

Pennington (1987b) also took a bottom up approach and proposed a model (Figure 6) with two different mental representations: a program model and a situation model. She found that when programmers are completely new to a program, the first mental model they build is an abstraction of control flow capturing the sequence of operations taking place in the program. This model, known as the program model, is built via chunking of micro structures like statements and control constructs into macro structures like test structure abstractions or chunks and via cross referencing.

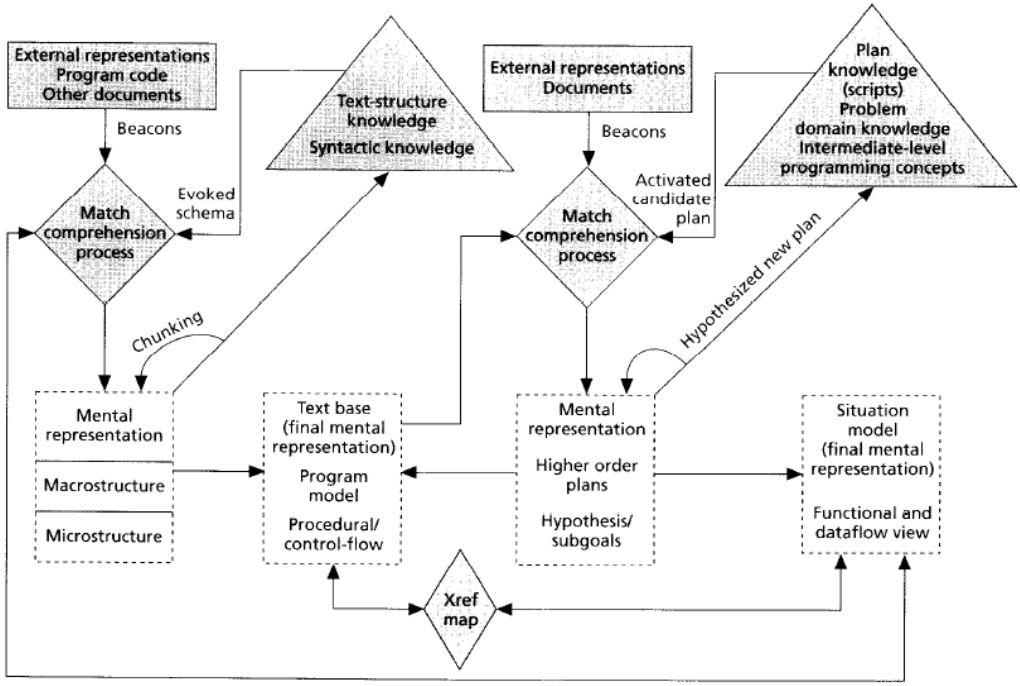


Figure 6. Pennington program comprehension model

On complete construction of this model, the situation model is developed that creates a dataflow/functional abstraction. Data flow abstraction refers to changes in meaning or values of program objects; functional abstraction refers to the program goal hierarchy. Knowledge of real world application domain is required to construct this model. This model too is built via cross referencing and chunking. Based on the program model, hypothesized higher order plans are constructed. The situation model is completed once the program goal has been reached.

Brooks (1983) proposed that programmers comprehended a program by reconstructing the domain knowledge used by the initial developer and mapping that to the actual code. This is a top-down approach to program comprehension. It involves creating a mental model based on an initial hypothesis about the global function of the program, which is then refined by forming auxiliary hypotheses. These are iteratively refined, based on the presence or absence of beacons, which are a set of features that match a hypothesized structure or operation.

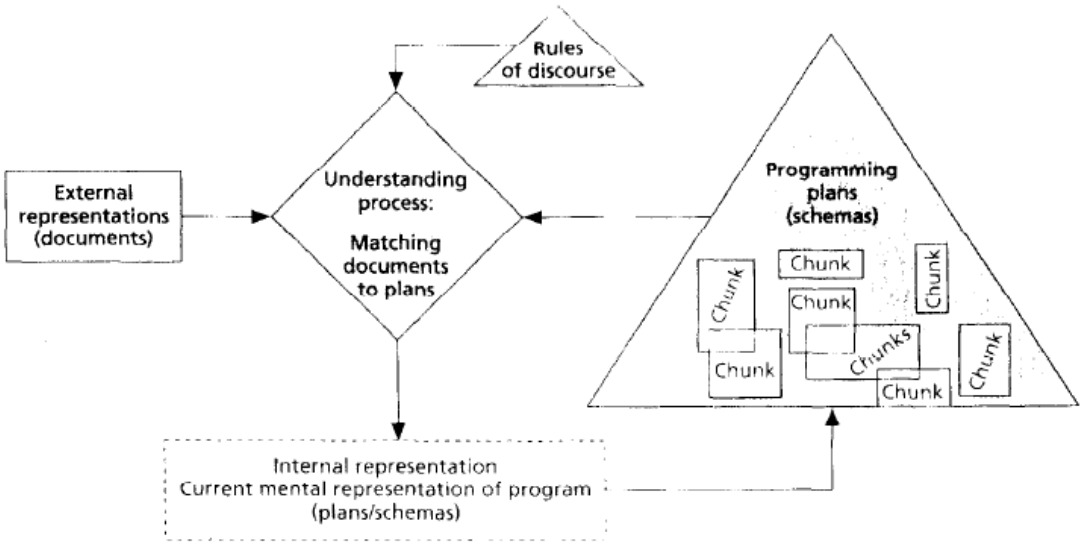


Figure 7. Soloway, Adelson and Ehrlich's program comprehension model

Soloway, Adelson and Ehrlich (1988) also supported the top down approach in cases where code or type of code is familiar to the programmer. They proposed that the mental model is developed top down by forming a hierarchy of goals and programming plans to achieve higher level goals. Their model involves usage of two types of programming knowledge represented by triangles in Figure 7.

- Programming plans are generic fragments of code that represent typical scenarios in programming. For example, a search algorithm which uses an index to iterate through each element in the list.
- Rules of programming discourse capture the conventions of programming, such as algorithm implementations and coding standards.

The rectangles represent the internal or external representations. The understanding process (represented by diamond) matches the external representations to programming plans using rules of discourse to select plans. On establishing a match, the internal representation is updated based on the gathered knowledge.

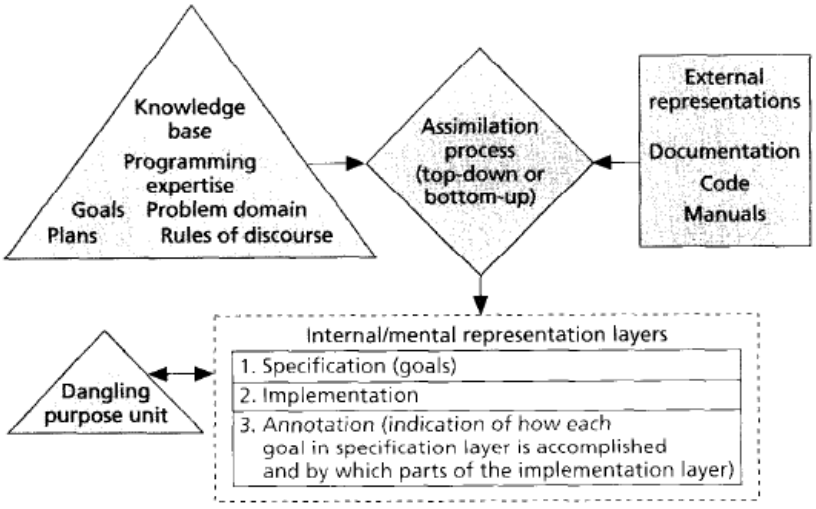


Figure 8. Letovsky program comprehension model

Letovsky (1986) proposed a high-level comprehension model with three main components: knowledge base, mental model, and an assimilation process. Programmer's

prior knowledge and expertise put together form the knowledge base. The mental model consists of three layers as shown in figure 8. The topmost layer - specifications - characterizes program goals. The implementation level layer contains the lowest level of abstraction, with data structures and functions as entities. The annotation layer links the goals in the specifications layer to the implementation layer. There could be some incomplete links, which are represented by the dangling purpose unit. In this model, there can be either top down or bottom up assimilation based on prior knowledge. Assimilation describes how the mental model evolves using programmer's knowledge with the program source code and documentation.

Littman et al. (1987) and Soloway et al. (1988) took a different approach and suggested that program comprehension strategy could be systematic and as-needed. Littman and colleagues observed that programmers either read the program systematically by tracing the control flow and data flow, or took an as needed approach by focusing only on code which is related to a particular task at hand. Soloway et al. proposed a model by merging concepts of systematic strategies, as needed strategies and inquiry episodes.

We see that there is some disparity among the models discussed. However, all models agree that programmers use existing knowledge during comprehension. The disparity arises due to characteristics of the programmer, goals and the program comprehended, which has been observed by many researchers and acknowledged. Table 2.2 summarizes the factors influencing comprehension strategies as found by Storey et al. (1999).

Maintainer characteristics	**Program characteristics**	**Task characteristics**
• Application domain knowledge • Programming domain knowledge • Maintainer expertise, creativity • Familiarity with program • Support tools expertise	• Application domain • Programming domain • Program size, complexity, quality • Documentation availability • Support tool availability	• Task type, purpose • Task size and complexity • Time and cost constraints • Environmental Factors

Table 2.2 Influences on program comprehension strategies (Storey, Fracchia, & Müller, 1999)

2.2 Program Debugging

Numerous investigations of debugging practices have been conducted since the mid 70's and cover a broad range of topics. Despite the wealth of knowledge, programming still remains both intricate for programmers to learn and taxing for educators to teach. Majority of the studies focus on two main components (McCauley, et al., 2008) : 1) types of knowledge critical for successful debugging and 2) strategies employed while debugging a program. We will now take a look at some of the relevant studies that looked at debugging strategies, followed by knowledge aids.

2.2.1 Strategies Employed

In one of the early studies, Gould (1975) investigated the debugging practices of experts. During the study it was observed that programmers began by either reading the code until something suspicious was detected, or by analyzing the output. Bugs that were considered easier to locate, such as index errors in loops or array references, were first looked at. This was followed by spending time to understand the program to find more subtle errors like bugs in assignment statements.

Vessey (1985) studied both expert and novice debuggers. Experts were more likely to take a breadth-first approach, trying to understand a program, whereas novices took a depth-first approach, focusing on finding and fixing an error without regard to the overall program. Furthermore, she suggested a hierarchy of debugging goals, similar to the process used by experts observed in (Gould, 1975): 1) discover the problem by comparing correct and incorrect output; 2) become familiar with the intended function of the program and how it is structured; 3) examine the flow of control; 4) form a hypothesis about the source of the error; and 5) fix the bug. Ducassé and Emde (1988) described four categories of debugging strategies focused on bug location: 1) using mental and paper tracing of programs and other means of dissecting and executing code; 2) comparing the intended program against the actual program for computational equivalence; 3) looking for language consistency and recognizing well-formed programs and algorithms; and 4) detecting stereotypical errors.

Katz and Anderson (1988) conducted multiple studies of students debugging programs and observed the tactics implemented to troubleshoot the program. They found two predominant strategies used in locating bugs. With the first strategy known as forward-reasoning, programmers start searching the bug from the program code. Two variants of this are program-order (programmer simulated the program's execution) and serial-order (programmer read the code in the order in which the lines appear). Forward-reasoning includes strategies like program comprehension, where a bug is located while creating a mental representation of the program, and hand simulation where the programmer evaluates the code like a computer to understand the program more closely. The second strategy known as backward reasoning involves starting from the erroneous behavior of the program and working backwards to the source of error in code. It includes strategies like simple mapping, where the program's output directs to

the erroneous line of code, and casual reasoning where beginning from the incorrect output, the programmer works backwards to the program code that caused the bug. Katz and Anderson also found that students typically used forward reasoning when debugging others' code, but backward reasoning when debugging their own. Students trained in a specific technique tended to reuse that technique. Their study also revealed that errors made by more experienced programmers are generally not repeated and easily fixed when found. This suggests that for many students, the difficulty of debugging is not in repairing the error, but rather in troubleshooting—understanding the program, testing the program, and locating the error.

2.2.2 Knowledge Aids

Ducasse´ and Emde (1988) based their work on Gould's framework and conducted a review of debugging systems and cognitive studies. They identified seven knowledge types that are utilized during debugging. It is not necessary that all knowledge types be known for every debugging task. They also stated that the wide range of knowledge made it difficult to incorporate them in one single debugging environment. The knowledge types were summarized as:

- knowledge of the intended program (program I/O, behavior, implementation);
- knowledge of the actual program (program I/O, behavior, implementation);
- an understanding of the implementation language;
- general programming expertise;
- knowledge of the application domain;
- knowledge of bugs; and
- knowledge of debugging methods.

The studies discussed in this section did not employ a development environment similar to the IDE's that are professionally used. Professional IDE's typically let

programmers use multiple visualizations of the same code to facilitate program understanding and debugging. Although few studies have used IDE's with dynamic visualizations for debugging studies (discussed in section 2.4.4), the IDES used were not professional IDEs. In our research, we investigated debugging with a professional IDE that provides a plethora of representations to the programmer. We will now look at some of the popular program visualizations available with IDE's.

2.3 Program Visualizations

In order to support programming activities like debugging and code comprehension, IDE's provide multiple visualizations that present the underlying code base in diverse abstract forms, such as animated views of program executions. Program visualization connotes a connection with the program at a lower level (e.g. data structures) rather than at the higher level of algorithms (Stasko et al., 1998). Both novice and expert programmers benefit from using appropriate visualizations while comprehending program code. The strategies employed in their usage may differ based on a programmer's expertise, familiarity with the IDE, experience with the visualizations and the current stage of program understanding. The object oriented programming paradigm specifically utilizes several graphical representations to describe program structure, given the nature of underlying program code.

According to Romero et al. (2003a), two important attributes of a representation is its information modality and the programming perspective highlighted by it. Information modality of a representation refers to the format in which the underlying data is presented. The modality could range from simple text (propositional) to highly graphic (diagrammatic), where both propositional and diagrammatic could be considered as two extremes of a scale containing representations with different degrees

of 'graphicality' (Cheng, Lowe, & Scaife, 2001). Program code, for example, is not purely propositional as there generally is a line per instruction format and is indented at varying degrees. In terms of taxonomy of graphic languages, it can be termed as a hybrid category of text between a list and a linear branching configuration (Romero et al., 2003a). A UML diagram on the other hand is more of a graphical representation even though it contains textual components. Programming perspective of a representation refers to the information structure highlighted by the representation. These information structures represent different types of information pertaining to program code. Programmers, when comprehending code, generate a mental model that consists of these different information structures representing different perspectives of the same program (Pennington, 1987b). Research has shown that these different perspectives are important: function, structure, operations, data-flow and control-flow. We will discuss these perspectives in detail in chapter 5. Some of the common visualizations bundled with a development environment and in widespread usage are UML Diagrams, variable watch windows, dynamic visualizations, output windows and expression watch windows. Again, these will be discussed in detail in chapter 5. Also see (Romero et al. 2003a) for a survey of external representations employed in object oriented programming environments.

Although there is evidence that these visualizations aid programmers in accomplishing programming tasks and pose no cognitive load individually, there might be issues with a programmer having to coordinate multiple visualizations and program code. Studies have been conducted on coordination of representations in other fields such as arithmetic (Ainsworth, Wood, & O'Malley, 1998b), first order logic (Oberlander, Stenning, & Cox, 1999), physics (Sime, 1996), and general problem solving (Cox & Brna, 1995). These studies have highlighted the difficulty in coordination faced by

learners, especially novices. This difficulty has not been researched a lot in the area of programming environment design and computer programming in general. Some recent studies (Romero et al. 2002b, Navalainen et al. 2004, Bednarik et al. 2005) investigated this issue. But there are several intriguing questions that have not been answered yet. Some of these questions will be addressed in our research. We will now look at visual attention tracking and how it can aid us understand the underlying processes active during programming activities.

2.4 Visual Attention and Eye Tracking Methodology

In order to visually perceive the world around us, our eye projects an image of the object onto the foveal region of the retina. Once an image is stabilized on the retina, information is extracted. This high concentration foveal region is small and gauges objects in a two- degree visual span and hence multiple fixations are required to process a visual scene. Tracking these movements of the eye can give insights into the visual attention of a person completing a task. Also, knowing which objects were looked at, their order and perspective can help one understand the underlying cognitive processes in action and provide clues on how that scene was perceived.

In eye-tracking research, the principle that visual attention links to eye gaze is called an eye-mind assumption (Just and Carpenter, 1980). Duchowski (2007) acknowledges that even though in eye-tracking we assume that attention is linked to foveal gaze direction, it may not always be true. He suggests that at times parafoveal or peripheral processing can be used to extract information. Nevertheless, this assumption between focus of visual attention and gaze direction is valid in a complex information processing task (Rayner, 1998). Debugging is a highly complex and task driven process, hence this thesis relies on the eye-mind assumption.

2.4.1 Eye-tracking in HCI

Eye-tracking has been explored in academia for over 40 years. Investigations in cognitive sciences, language and advertising extensively employed eye-tracking in the 1960s and 1970s (Jacob & Karn, 2003). It has increasingly been adopted in human computer studies over the past two decades. We will now present a brief history of research employing eye tracking in HCI.

Eye-tracking has been primarily used for two tasks, one as a form of input to computer and the other as a source of non-intrusive data for studying human computer interactions (Jacob & Karn, 2003). Some of the studies which have used eye-tracking as assistive technology for those with motor disabilities explored possibilities of replacing or supplementing input devices (Barreto, Gao, & Adjouadi, 2008). Gaze was used to control pointing and selection of objects as a complement to mice input (Biej, 2009) or used to completely replace them (Kumar, Paepcke, & Winograd, 2007). Majority of studies though have employed eye-tracking to study user behavior during their interaction with interfaces. Studies of navigation and web browsing have been particularly popular. These studies looked at placement of links in target links, patterns in eye movements while browsing, feature detection etc. (Cutrell & Guan, 2007). Menu selection tasks were also studied, where a significant difference was found between selection of menu items compared to reading the menu items (Aaltonen, Hyrskykari, & Räihä, 1998). Other studies have looked at use of gaze in immersive collaborative environments (Steptoe, et al., 2008), building document summaries based on focus of user attention (Xu, Jiang, & Lau, 2009) and visualization of hierarchical structures.

Although eye tracking has been used to evaluate both top down and hypothesis driven experiments, there is still a dearth of research incorporating eye tracking in HCI

studies. High-cost of eye-trackers and challenges in data interpretation are two major hurdles. Regardless of the fact that eye-trackers are now much more easily accessible both financially and technically, their use is still constrained by the complexities of processing and interpreting complex data. Commercial eye tracking companies are making a conscious effort to simplify data analysis by bundling software packages that present the raw data collected in a much more intuitive and easily comprehensible format. With these advancements, it has been observed that over the past decade eye-tracking has become more widely used in the commercial market, especially in studies of web usability. We will now look at the evolution of eye-trackers and the different types used by researchers and usability professionals.

2.4.2 Types of Eye Trackers

Advancement in technology has made eye trackers less complex, more usable and more affordable than in the past. Earlier trackers were cumbersome for participants to use as sensitivity to head motion meant that restraints like chin rests and bite bars had to be used to reduce head movements. Eye trackers nowadays are more tolerant of head movements and can easily address issues of stabilization.

In general, there are two types of eye tracking techniques: those that measure the position of the eye relative to the head, and those that measure the orientation of the eye in space (Young & Sheena, 1975). The first approach employs techniques like electrical oculography, where the potential differences of skin around the eyes are measured; scleral contact lens/search coil, where a device is mounted on the eye using contact lenses and photo-oculography, where features of the eye (such as the apparent shape of the pupil) is measured when it is in different positions (Duchowski, 2007). The second technique used for point of regard measurement requires that either the position

of the head must be fixed or multiple ocular features be measured. Corneal reflection and pupil center are examples of such features. These are measured by capturing infra-red reflections of the eye with a video camera and image processing. The infra-red rays are invisible and non intrusive. Furthermore, two approaches are used to determine the gaze location. The bright pupil technique results in a dramatic contrast between the pupil and the iris, making the pupil easily distinguishable and therefore easier to track. There is little interference from eyelashes and shadows because the image-processing algorithm recognizes a white elliptical region as the pupil. Creating a pronounced bright pupil effect, however, is highly dependent on pupil size, which is affected by several external factors like age, emotional response to stimuli, and lighting sources. This method tends to work better in a dark environment and on children and people with blue or light eyes. The dark pupil method detects the dark ellipse of the pupil within the iris. This methods works well in bright environments and outside in natural lighting conditions, but there are issues with eyelashes and shadows causing false positives during pupil detection. Dark colored eyes work best because the IR light reflection off the iris makes the dark color of the iris appear light in the digital image, thus making the pupil more easily discernible. Image processing algorithms use this image of the pupil combined with the reflection from cornea, also known as Purkinje image (Crane, 1994), to calculate gaze location. This location is then superimposed on the scene under evaluation either for real time calculations or recorded for delayed analysis.

Commercially available apparatus for eye tracking can be broadly categorized as high speed eye tracker, remote eye tracker or head mounted eye tracker. Remote eye trackers with no physical contact are more popular in usability studies, as wearing helmets and miniature cameras required by head mounted trackers are likely to be distracting for subjects (Jakob & Karn, 2003). Most of these eye trackers employ video

based corneal reflection tracking technique with infra-red light emitters. These eye trackers can track eyes with accuracies of less than 0.4 degree and sampling rates ranging between 60-1500 Hz.

2.4.3 Eye-tracking Measures

There are over 100 measures of eye-tracking reported in the literature and application of these measures is experiment dependent. These measures are selected while creating a study design, and derives from the research question. It might also be the case that none of the existing measure may fit a new experiment. However, there are some measures that can be used in most every eye tracking based experiments.

Two types of eye movements are tracked by eye-trackers, saccades followed by fixations. Saccades are rapid eye movements that allow the fovea to view a different portion of the display. During a saccade, vision is suppressed and does not become active until its destination has been reached. Often, a saccade is followed by one or more fixations when objects on the scene are viewed. Then small eye movements are made within a general viewing area for about 200-600ms. We define a gaze as one or more successive fixations on a particular object or area of a visual scene. In order to reduce the amount of data produced by eye-tracking, it is a common practice to separate a visual scene into 'Areas of Interest' (AOI), also known as 'Regions of Interest' to support aggregation of fixations. While assessing the quality of interfaces, Goldberg and Kotval (1998, 1999) assessed the validity of various eye tracking measures. They proposed a set of eye tracking measures that supported automation. These measures were either dependent or independent of the AOI's. They also proposed a classification of the eye tracking measures, according to which, if a measure describes a time based property of a scanpath, it was termed as temporal. Fixation duration

would be a temporal measure. If the measure described the spread and coverage of a scanpath, it was termed as spatial. Number of saccades is an example of a spatial measure. Jacob and Karn (2003) put together a set of eye tracking measures based on their analysis of usability studies. These measures include:

- *Number of fixations, overall:* The number of fixations overall is thought to be negatively correlated with search efficiency.
- *Gaze % (proportion of time) on each area of interest:* The proportion of time looking at a particular display element could reflect the importance of that element.
- *Fixation duration mean, overall:* Longer fixations (and perhaps even more so, longer gazes) are generally believed to be an indication of a participant's difficulty extracting information from a display.
- *Number of fixations on each area of interest:* The number of fixations on a particular display element should reflect the importance of that element. More important display elements will be fixated more (frequently).
- *Gaze duration mean, on each area of interest:* gazes on a specific display element would be longer if the participant experiences difficulty extracting or interpreting information from that display element.

Using these measures can significantly reduce the data gathered and make analysis more efficient. There has also been a shift from using raw gaze data to more sophisticated measures involving scanpath analysis based on context. For example, if the expected eye pattern for efficient usage was a straight line to a target, inefficient usage might show longer paths. Yoon and Narayanan (2004) investigated the order of fixations to measure how systematically a user attends to casually related areas of interest. Analysis of scan paths based on string editing of fixation sequences is also popular.

2.4.4 Visual attention in studies of programming

Most of the studies investigating cognitive and behavioral aspects of programming employed verbal utterances of participants. It has been argued that verbalizing thoughts interfere with a participants natural processing, by adding an extraneous cognitive load. The results could be biased as this load can also hinder the problem solving strategies of participants, especially novices. Users may also skip critical utterances due to different causes such as not being aware of some aspects of behavior or being less vocal by nature. As eye-tracking is non-intrusive, it has been suggested as a strong alternative to verbal protocols in capturing the cognitive processes involved in programming. Researchers have successfully employed this technique in studies of programming to better understand the underlying cognitive processes.

Crosby and Stelovsky (1989) studied the visual patterns of programmers while reading a binary search algorithm. Attributes like fixation times and number of fixations were captured by an eye tracker. They found that more experienced users paid attention to meaningful areas of source code and complex statements. Novice students paid more attention to comments and comparisons. Least attention was paid by both groups to keywords, and they did not exhibit any methodical differences in code reading strategies. Crosby and Stelovsky evaluated fixation durations and number of fixations with both qualitative approaches and parametric tests. The only representation available to participants was the program code. This study did not employ any static or dynamic visualization of the code.

Not many studies looked at visual attention following this early work, until 2002 when several experiments were conducted by Romero et al. (2002a, 2002b, 2003b). They evaluated co-ordination strategies of programmers while debugging in an environment

that provided multiple visualizations. Code, output and a static visualization of the program was available for comprehension. It was found that programmers frequently combined both forward and backward reasoning to debug a program. Frequent switches were made between code and output or code and graphical visualization of code. Balanced switching behavior was found among those with more programming experience. Statistical tests were used to analyze the visual data collected. The data analyzed was an aggregated average from the beginning to the end of a debugging session for each participant. Visual attention during the experiment was tracked by a Restricted Focus Viewer (RFV) (Blackwell, Jansen, & Marriott, 2000) where the programming environment was presented in a blurred format with a clear window at the location of the mouse cursor that the programmer could control. This way the RFV restricted the amount of stimulus shown to the user and facilitated the tracking of the visual attention of the programmer.

Nevalainen and Sajaniemi (2005) investigated the effect of graphical visualizations in the visual patterns of novice programmers. They implemented a within subject design where subjects used two different tools; a)a traditional text based environment and b) an environment that provided multiple graphical visualizations. Differences in visual pattern were found between these two tools. Usage of any of the tools led to a significant amount of time spent away from the source code or visualization itself. However, no significant effect of the tools was found on the mental model created. The visual attributes used in analysis were fixation duration and proportion of these durations over three different AOI's. Here again, the data analyzed was an aggregated average from the beginning to the end of a debugging session for each participant.

They followed up this study with another study to better understand program comprehension. They used the PlanAni program animator with two modes supporting either static or dynamic visualization of the program code. Between subject design was used with one group assigned as a static group and the other an animation group based on pre-test scores. It was found that most of the time was spent reading the program code irrespective of the group. The data analysis methodology was similar to the previous experiment. In addition, qualitative analysis of short segments of video protocols with gaze overlay was conducted.

A predominantly qualitative approach was taken by Umano et al. (2006) to analyze visual patterns among intermediate programmers. They studied six short source code review tasks while debugging. Based on the study, they identified a particular pattern, called scan, in the subjects' eye movements. It was found that reviewers who did not spend enough time for the scan tended to take more time for finding defects.

Betnarik et al. (2005 & 2006) conducted studies to investigate the effects of experience on debugging strategies in a multi representation dynamic environment. It was found that fixation counts and attention switching between representations (like code and graphical representation of execution) did not differ based on experience. An effect of experience was found, however, on overall strategies adopted to comprehend programs and on fixation durations. In these studies, data was analyzed with averaged data read from the entire session. In order to characterize and analyze cognitive processes, Bednarik and Tukiannen (2006) proposed a new methodological approach and conducted more detailed analyses. They subdivided the comprehension process into meaningful pieces and analyzed gradual changes in related eye-movement patterns. Instead of using a repeated measures analysis, binomial trials were conducted. All the experiments conducted by Bednarik used the Jeliot IDE. Even though Jeliot supports

dynamic representations, it was not reflective of the IDE features that are available in professional IDE's, and is primarily aimed at academic instruction. Bednarik et al. (2007b) later conducted a comparative study of program comprehension, evaluating RFV against an eye tracker, to investigate whether the blurring of the screen by RFV affected strategies. On analysis of the frequency of attention switching, they found that there was an effect of blurring on strategies. In terms of performance though, no effect of blurring was found.

CHAPTER 3

PROBLEM STATEMENT

Although multiple studies have investigated the different strategies and approaches involved with program comprehension and debugging, knowledge about how programmers build a mental model of a program based on multiple representations is still obscure.

Studies that investigated debugging strategies with multi visualization IDE's restricted the use of representations to a select few during experiments. These experiments did not replicate a more realistic program debugging environment comprising tools/visualizations used by professionals. Participants were devoid of access to all the visualizations restricted by either the limitations of the IDE or the conditions set by the experimenter. As a result of this, controlled experiment results in the literature may not be a clear or realistic representation of the actual behavior exhibited by programmers. We will overcome this by utilizing an IDE (jGRASP) that offers a plethora of visualizations, that is used both academically and professionally, and which gives programmers unrestricted access to multiple static and dynamic visualization aids along with program code.

Several important questions related to visual attention and its role during programming within these environments can be raised. A general question about what information sources programmers attend to when working with a

development environment leads one to first ask about how to record visual attention in programming. Whether and how the cognitive processes involved in programming are reflected in visual attention patterns, however, is not completely understood. Are there general patterns of visual attention with which programmers attend to the source code and the other representations while comprehending a program? What are the programmers' visual strategies and how can they be identified from eye-movement data? Does the focus of visual attention correlate with other information about the comprehension process? Is it possible to distinguish between good and poor comprehension based on information about visual attention? The lack of knowledge about these and related aspects of visual attention during programming motivates the research presented in this thesis. Eye-tracking technology seems to be a suitable tool to increase our understanding of the role of visual attention in programming and, therefore, the possibilities and limitations of it and the associated techniques need to be studied and understood.

The purpose of this research is two-fold: (1) to understand the underlying processes which are active during a program debugging activity and (2) to use eye tracking methodology to develop new analysis paradigms for program comprehension/ debugging studies.

CHAPTER 4

PROPOSED COGNITIVE MODEL

4.1 Foundation

In the following section, we discuss a cognitive model of debugging that we have developed in this research. It incorporates various static and dynamic representations a programmer uses while comprehending, and then debugging, a program. This model has been derived by synthesizing and extending some of the significant work in the area of text comprehension, comprehension of text and diagrams, graph comprehension and program comprehension. We first discuss this relevant literature, and then present our model.

4.1.1 Text Comprehension

While investigating the learning strategies used by students during explanative text comprehension, Mayer (1996) explored the various cognitive processes involved in knowledge construction. He proposed that the key to meaningful learning were three cognitive processes, namely selecting, organizing and integrating. The model derived from this was called the SOI (Selection-Organization-Integration) model of the architecture of human learning. It consists of sensory memory, short term memory and long term memory, as shown in Figure 9. The first process involved in comprehending an expository text is the reader

determining what is important by focusing conscious attention on relevant pieces of information. This information is then added to the working memory. Mayer termed this process as selecting; it is also known as selective coding. The next process involved in comprehending an expository text is organizing key pieces of information selected in the previous step and forming a coherent structure. The reader builds an internal connection between all the encoded information to form an integrated whole. This is represented in the model by the recursive arrow from the short term memory back to itself, and is called organizing or selective combination.

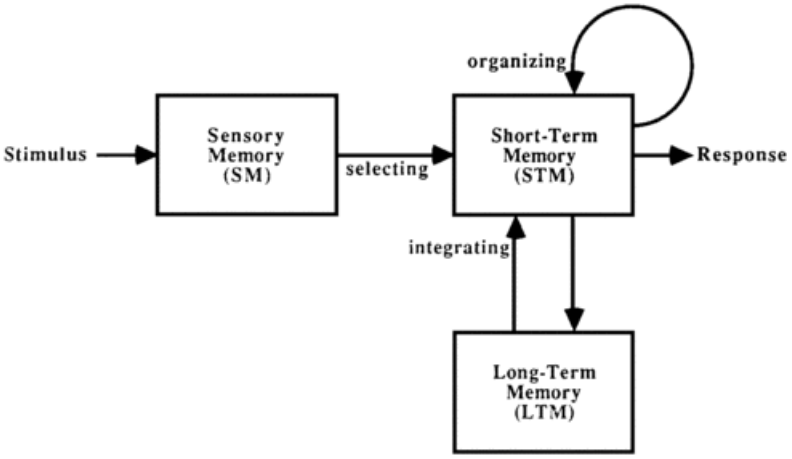


Figure 9. SOI Model

During the final process, the new knowledge constructed in the short term memory is related by building external connections with the analogous knowledge from long term memory. This essentially means that the reader relates his prior knowledge to the information presented. This final process is known as integrating or selective comparison, and is represented by the arrows between short term memory and long term memory.

4.1.2 Text and Diagram Comprehension

These cognitive strategies and the resulting mental representations have a bearing on effective comprehension from text and supporting multimedia. Narayanan and Hegarty (1998) proposed guidelines for interface design of hypermedia presentation systems, based on user's mental models and comprehension strategies. Comprehension was postulated as a constructive process during which an individual uses his/her domain knowledge, information presented in the external media and reasoning skills, to build a mental model of the presented material.

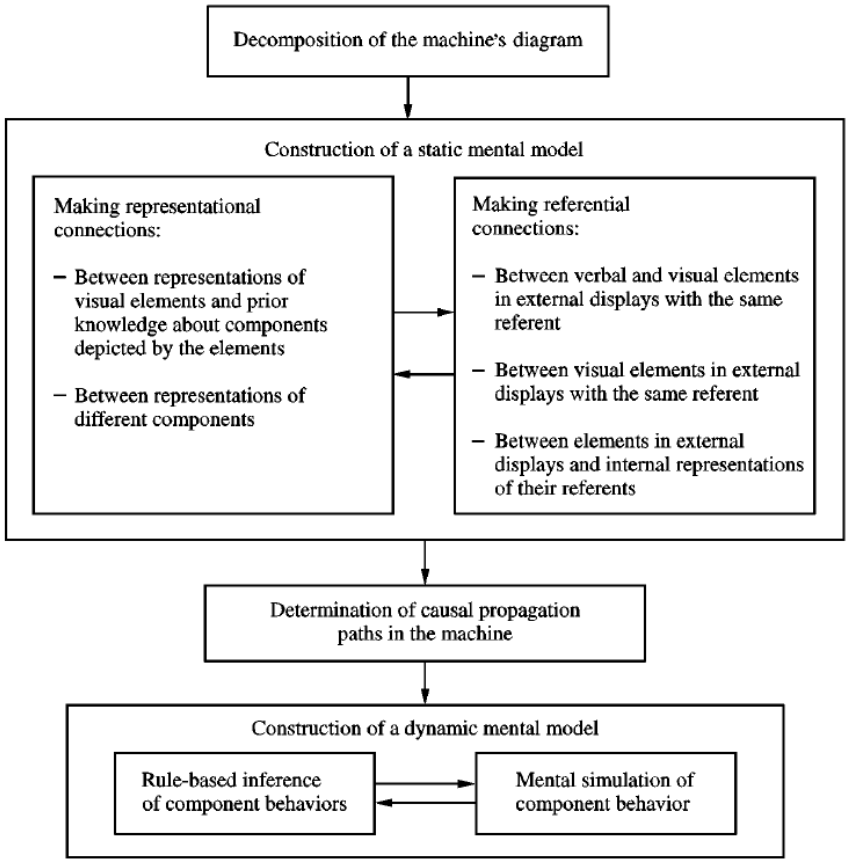


Figure 10. Cognitive model of diagram and text comprehension (Narayanan and Hegarty, 1998).

Their model was an extension of text comprehension models, which view comprehension as a construction of mental model representative of the text. For example, the process of comprehension of the description and depiction of a simple machine by an individual was segregated into multiple stages as shown in Figure 10. During stage one, the basic elements of the machine are identified from its diagrammatic representation. This consists of breaking down the connected diagram into elementary units that correspond to objects. In the next stage, a static mental model is constructed by making two types of connections. First, the user establishes connections between the diagrammatic elements identified in Stage 1 and their real world referents, identified from his/her prior knowledge. Following this, the user tries to comprehend spatial relations between different machine components by building connections between internal representations of these components. These spatial relations help determine how components affect and constrain other components, and further guides the reasoning about casual relations.

During Stage three, the user further builds on his/her static mental model by making referential connections between the text and the diagrammatic units that depict their referents. As this stage is critical in constructing an integrated representation of text and diagram in memory, failure here could lead to only a surface level interpretation of the text or a surface level interpretation of the diagram. Making referential connections is also a necessary process when users have to integrate information in two different pictorial displays of the same machine. In the next stage, potential causal chains of events in the operation of the machine are established. Determining these lines of action in a machine in advance reduces the computations required for predicting system behavior. In the fifth and final stage, the user constructs a dynamic model of the machine by inferring and integrating the dynamic behaviors of

individual components. This process is termed as mental animation. Mental animation is an iterative process wherein the user considers the components or sub-systems individually, assesses the influences acting on each, infers the resulting behavior of each, and then proceeds to consider how this behavior affects the next component or subsystem in the causal chain. Narayanan & Hegarty empirically validated this model by conducting experiments in two different domains of mechanics and computer algorithms (Narayanan & Hegarty, 2002).

4.1.3 Graph Comprehension

Research on graph comprehension has a long history (Carpenter & Shah, 1998; Lohse, 1993; Shah, Mayer, & Hegarty, 1999), and has yielded several useful theories of graph comprehension (e.g. Lewandowsky & Behrens, 1999; Shah & Hoeffner, 2002). To gain insight into how cognitive aids can help students understand scientific graphs, Mautone adopted the extended SOI model proposed by Mayer (2003) in their study (Mautone & Mayer, 2007). This study measured the effectiveness of scaffolding techniques in graph comprehension and empirically validated the proposed model. In graph comprehension, the cognitive process of organizing corresponds to mentally building a relation between the multiple variables shown on graphs. The construction of a relation between a variable on the x-axis to a variable on the y-axis would be an example of organizing. The cognitive process of integrating corresponds to combining new knowledge with existing knowledge.

Mautone also adapted cognitive aids like signaling, concrete graphic organizers and structural graphic organizers, originally proposed for text comprehension, for graph comprehension. In text comprehension, signaling refers to cues and aids that expose the prominence in the structure of text without adding new information. It helps highlight

key information and makes the relationships among various information items more visible. Various studies (Loman & Mayer, 1983; Rickards, Fajen, Sullivan, & Gillespie, 1997) have validated the effectiveness of signaling during the process of organizing, by helping learners form a coherent representation from the selected information. Signals include the use of highlighting, headings, summaries, outlines, and pointer words. Advance organizers refer to material presented prior to a text passage, such as a brief analogy or diagram showing the components of a to-be-explained system, and are intended to prime or provide prior knowledge of some of the more difficult subject matter content of the passage. These have been shown to be effective in helping students comprehend expository text under some conditions. Another type of graphic organizer, which Mautone referred to as structural graphic organizers, highlights the key structural relationships shown in the graph, independent of the content. As mentioned above, in text passages, advance organizers often involve presenting a brief analogous example prior to presenting the actual text passage. For example, in one study, prior to reading a passage about how radar works, participants were presented with a brief diagram depicting how radar waves might be compared to a rubber ball bouncing off of objects (Mayer R. E., 1983). Structural graphic organizers, such as signaling, are intended to guide the cognitive process of organizing. They help learners attend to and interpret important patterns and relationships, which, in turn, help them in constructing a meaningful understanding of the functional relationships among key variables in the graphs.

4.2 Proposed Cognitive Model

Based on our literature review and analysis of the shortcomings of existing research on program debugging, we propose a cognitive model of how multiple

representations are used by programmers to comprehend and debug a program within an IDE for object oriented programming. We *co-opted* ideas from text and graph comprehension literature (signaling, advance graphic organizers, organizing and integrating from the SOI model), the program comprehension literature (notions of program slice, data structure, function, data and control flow), cognitive processes from the text and diagram comprehension literature, and representations provided by typical IDEs (program code, CSDs, UML diagrams, visualizations and dynamic windows) and *integrated* these to develop this comprehensive and cohesive model. It is more detailed than any model of program comprehension and debugging hitherto offered in the literature. The rest of this section describes components and processes of this model.

There are three main components of the model, as illustrated in Figure 11, are the following. (a) The type of cognitive aids/ representations used while debugging. These aids have been categorized on the basis of their information modality, programming perspective (Romero et al., 2003a) and the cognitive dimensions they highlight (Green, 1989). (b) The cognitive processes, each of which is either primed by a cognitive aid or a process that is inherently evoked. (c) The mental representations derived from the cognitive processes and cognitive aids. The programmer constructs and manipulates his/her mental representations over the course of interacting with the programming environment and understanding the information presented. Mautone and Mayer (2007) took a similar approach in categorizing three components in their graph comprehension model.

Although in the proposed model the cognitive processes are described in sequence, we do not believe that the cognitive processes and internal representations depicted in Figure 11 will always occur in a specific, fixed sequence. The order will differ based on an individual's experience in programming, prior knowledge and reasoning.

According to Katz and Anderson (1988), a programmer could take two types of approaches in locating bugs. With forward reasoning, comprehension in particular, we postulate that the flow of cognitive processes will be as depicted in the model. The programmer first creates a static mental model based on the static visualizations presented by the IDE, followed by construction of a dynamic mental model. When the reasoning is bottom up, also called backward reasoning, bug location commences from the incorrect behavior of the program, typically from the output, and is traced back to the origin of the problem. Because this approach requires only partial/opportunistic program comprehension, the flow of cognitive processes is not predictable. Our research focuses on debugging strategies when the program is written by someone else, and hence models explicit program comprehension as well as debugging. We will first discuss the comprehension model (Figure 11), followed by a variant of this model for debugging (Figure 12). We view program comprehension as a constructive process, where prior domain knowledge, information from representations and reasoning skills contribute to assembling a mental model of the program. What follows is a detailed description of the various components of our cognitive model.

4.2.1 Cognitive Aids

Different representations help a programmer visualize the program through different perspectives or information types. For example, some perspectives highlight the transformations which data elements undergo as they are processed, while others show the sequence of actions that will occur when the program is executed. Visualizations can be presented in formats that range from textual to graphical (Romero et al., 2003a).

Two important aspects of a representation are its information modality and programming perspectives. The first aspect refers to the characteristics, advantages and

disadvantages of representations that are propositional and those that are diagrammatic. The diagrammatic degree of the representation can range from propositional to purely graphical. This is known as degree of 'graphicality' (Cheng, Lowe, & Scaife, 2001). For example, diagrams, unlike propositional representations, exploit perceptual processes by grouping relevant information together and therefore make the search and recognition of information easier. Propositional representations permit the expression of abstraction or indeterminacy, while diagrams compel the representation of specific information. On an IDE, program code cannot be considered as fully propositional because it uses formatting conventions to enhance its comprehension. Multi-modal external representations are common in IDEs that support complementary processes. Even though some representations highlight some information type, it does not mean that other information types are not present or cannot be derived from it.

The second aspect of a representation is the programming perspective highlighted by it. Computer programs are information structures that comprise different types of information, and programming notations usually highlight some of these perspectives at the cost of obscuring others. It has been established that programs can be looked at from different perspectives (Pennington, 1987b), and programmers when comprehending code are able to develop a mental representation that comprises these different perspectives or information types as well as rich mappings between them (Pennington, 1987a). We propose that multiple external representations provided by IDEs can be grouped under five categories of cognitive aids: signaling, textual representation, structural visualization, dynamic visualization and dynamic windows. These cognitive aids are illustrated on the left side of Figure 11.

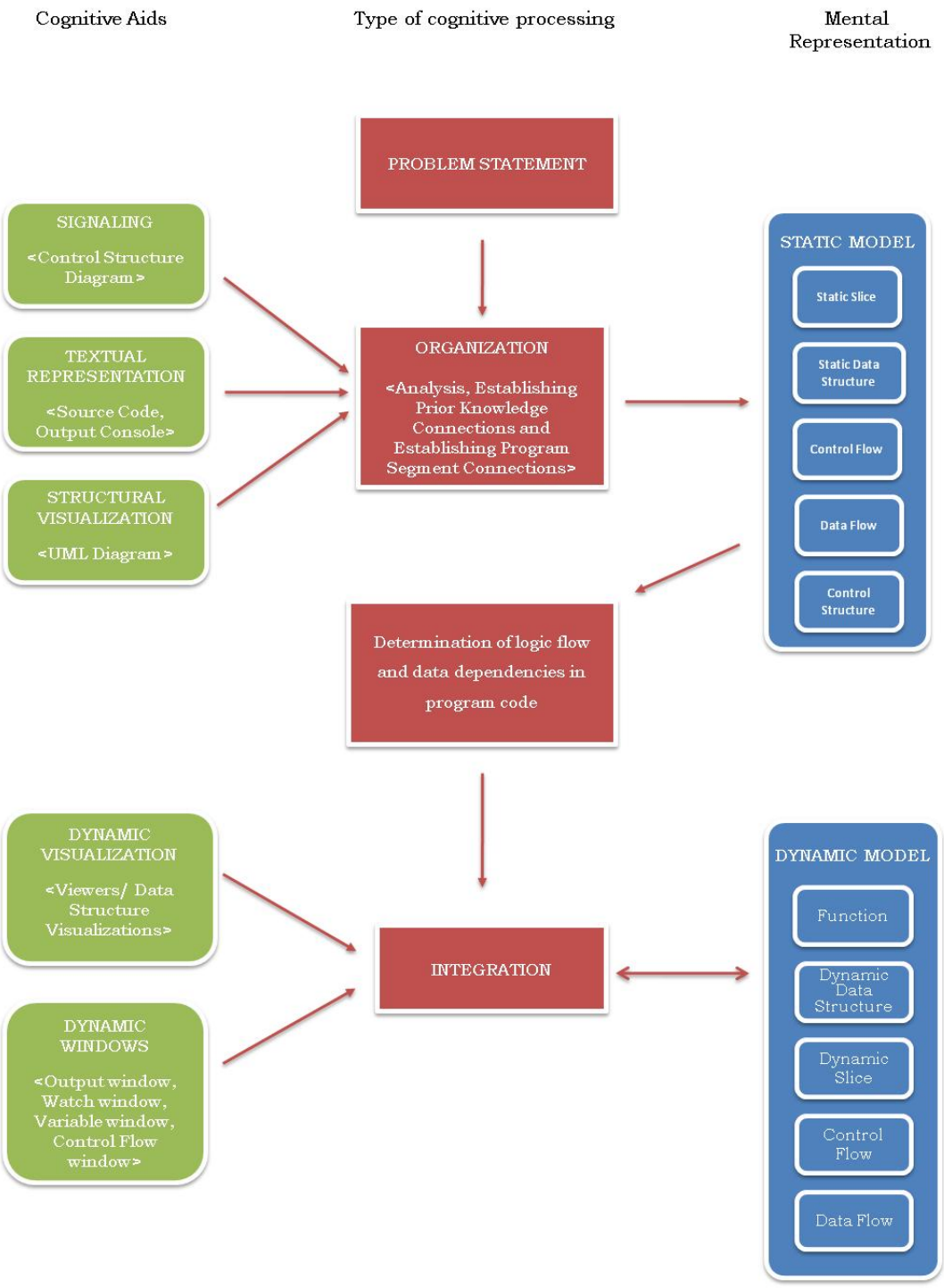


Figure 11. Cognitive model of multi representational program comprehension

Signaling refers to any technique that makes the structure of text more pertinent by highlighting key information and relationships among segments of text, without adding new information (Mautone and Mayer, 2001). It includes indentation, highlighting, formatting (bold, italic, etc.), and the use of color on code or a graphical representation associated with code. Signaling is intended to help guide the cognitive process of organizing, during which learners organize selected information into a coherent representation. Control Structure Diagrams (Figure 11a), which automatically highlights the structure of code and indents it with graphical notations, is an example of signaling. It improves the comprehensibility of source code by clearly depicting control constructs, control paths, and the overall structure of each program unit (Cross II et al., 1998).

Textual Representation: Source Code – According to (Grubb & Takang, 2003), "Source code can be divided into program code (which consists of machine-translatable instructions); and comments (which include human-readable notes and other kinds of annotations in support of the program code)". Program code is a sequence of instructions written to perform a specified task. Although it can be considered as plain text, there is a degree of graphicality involved in its representation in almost all the higher level programming languages. It is formatted by extensive tabbing and grouped as constructs with special characters. This formatting improves the comprehensibility of code. Studies have shown that program formatting is used in comprehension (Katz & Anderson, 1987). Comments on the other hand are embedded with program code as annotations to aid a programmer in understanding the source code.

Textual Representation: Program Output – This refers to the information produced by the program. This information could be an output in the form of explicit display on a console, processed data files or influences on the behavior of a dependent

program. Early research by (Gould, 1975; Gould & Drogowski, 1974) established that program output was used by programmers to establish hypothesis of a bug. The importance of output was further confirmed by Katz and Anderson (1987) and Romero et al. (2003b) who investigated programmers' usage of program output on console.

Structural Visualization is a diagrammatic representation that highlights key structural relationships independent of content. These help learners attend to and interpret important patterns and relationships, which, in turn, help them in constructing a meaningful understanding of the functional relationships among classes in a project's architecture. Structural visualizations are intended to guide the cognitive process of organizing (Mautone and Mayer, 2001). For example, in one study, prior to reading a passage about how radar works, participants were presented with a brief diagram depicting how radar waves might be compared to a rubber ball bouncing off of objects (Mayer R. E., 1983). Radar waves and rubber balls do not share the same surface features, but the two do share the same structural features: Both bounce off objects and return, more or less, to the point of origin in a given amount of time. Unified Modeling Language (UML) class diagrams (Figure 11b) highlight the relationships between multiple classes in an object oriented programming project by employing visual modeling. These diagrams visualize a system's architecture using design elements such as classes, packages and objects. They also display relationships such as containment, inheritance, associations and others (Booch et al., 1999). Sequence diagrams that illustrate the control flow within classes would be another example of a structural visualization for program comprehension and debugging.

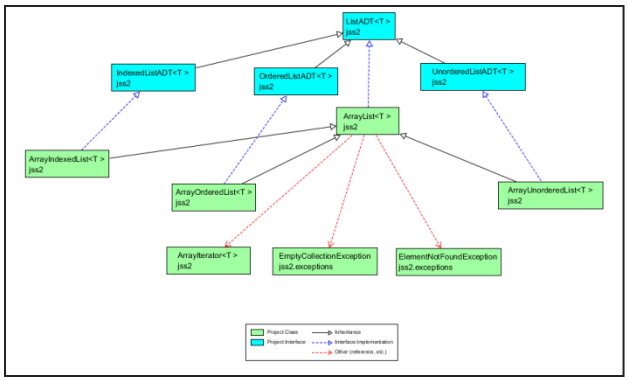


Figure 11a UML Diagram

```
//-----------------------------------------------------------------
//  Returns true if this list contains the specified element.
//-----------------------------------------------------------------
public boolean contains (T target)


//-----------------------------------------------------------------
//  Returns the array index of the specified element, or the
//  constant NOT_FOUND if it is not found.
//-----------------------------------------------------------------
private int find (T target)
{
   int scan = 0, result = NOT_FOUND;
   boolean found = false;

   if (! isEmpty())
      while (! found && scan < rear)
         if (target.equals(list[scan]))
            found = true;
         else
            scan++;

   if (found)
      result = scan;

   return result;
}
```

Figure 11b Control Structure

```
public class DoublyLinkedListExample {

   public static void main(String[] args) {
      while (true) {
         DoublyLinkedList list = new DoublyLinkedList();
         for (int i = 0; i < 3; i++) {
            list.add(String.valueOf(i));
         }
         list.add(null);

         for (int i = 3; i >= 0; i--) {
            list.insert("x" + i, i);
         }
         for (int i = 0; i < 3; i++) {
            list.remove(i);
         }
      }
   }
}
```


Figure 11c Dynamic Data Structure Viewers

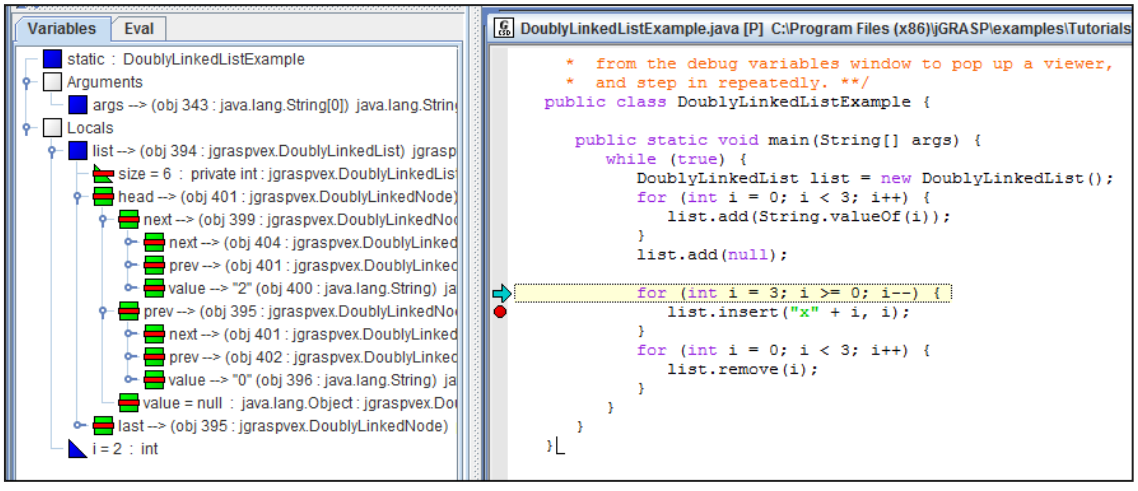


Figure 11d Variable Watch

Dynamic Visualization is a graphical representation that shows change, e.g., a diagrammatic representation that shows how underlying data structures are updated as a programmer steps through a program. With the assistance of such representations, a programmer can establish relationships between known data structures and the program under execution, and thus accomplish the cognitive process of integration. Many studies have conducted research validating the application and effectiveness of dynamic data structures (Myers, 1983; Baker et al., 1999; Shimomura and Isoda, 1991). An example of a dynamic visualization is the object viewer (Figure 11c) in jGRASP that provides structural views of java collections, classes and arrays during debugging (Cross II et al., 2007). When a class has more than one type of view associated with it, the programmer can open multiple viewers in order to compare different aspects of the data structure.

Dynamic Windows are representations that highlight the status of various attributes of a program during execution. The information modality of these representations is predominantly propositional, but can also be graphical (e.g., table or histogram). These help a programmer establish a relation between the pre existing structural representation of a program in short term memory and its current execution by highlighting control flow and data flow, and hence help the process of integration. Variable windows, output windows and call stack windows are some examples of dynamic windows (Romero et al., 2002a). The dynamic window shows a measure of execution activity and memory for threads, packages, classes, methods or objects. In Figure 11d, a variable window is shown, which displays the variable state during program execution. Here, every variable visible at current program state is displayed in different lines, and if they are complex structures they can be expanded to show their components. When they expand, their components are shown with an indentation to

denote this hierarchical relation. These components, if complex, can in turn be expanded in a recursive fashion displaying a hierarchical tree.

4.2.2 Mental Representations

During program comprehension, programmers build their own mental representation of the program to be understood; a mental model. They start by reading code statements and group these statements until a high-level mental representation of the program is constructed. Pennington (1987a, b) describes two program abstractions that are formed by the programmer during comprehension of a structural program: the program model, which is a low-level abstraction, and the domain model, which is a high level abstraction. She also describes four basic categories of program information making up the programmer's mental representation: elementary operations in the code, control flow, data flow and program goals. Burkhardt et al. (1997) further extended this model to account for object oriented programs. They added information about objects as well as the relationships among objects to the situational/domain model. Information about objects and goals represents the static aspects of the program, whereas information about data-flow and class dependence represents more dynamic aspects of the program.

The proposed model postulates that there are two categories of mental models constructed by the programmer during comprehension, namely static and dynamic models. Each of these include further sub constructs of mental representations that correspond to program specific information. The sub constructs are either primed by the cognitive aids or are generated from the problem statement during the cognitive process of organizing. The usage of these sub constructs is completely dependent on a programmer's expertise, the task and the development of understanding over time

(Burkhardt et al., 1997). Also, some of these information types might dominate the mental representations (Pennington, 1987b).

According to our cognitive model, during program comprehension, programmers build a *static mental model* of the program from the program code and problem statement, and any signaling and static visualizations that may be provided by the IDE they are using, through the cognitive process of *organizing*. The static model represents the static aspects of a program and consists of the following sub constructs.

Static Slice of a program consists of all statements that may directly or indirectly affect the value of a variable at some point in the program (Weiser, 1984). Building a static slice requires finding all statements that could influence the value of the variable for any input, not just the statements that did affect its value for the current input (this is the *Dynamic Slice* as explained later). Static slices are identified by finding consecutive sets of indirectly relevant statements, according to data and control dependencies. Signaling can prime static slices by emphasizing program control structures and constructs (e.g., CSDs, see Cross II et al., 1998).

Static Data Structure is a data structure that does not change within the scope of the program (Guzdial and Ericson, 2010). Examples are class hierarchies of a software project and array structures. These data structures are easily identifiable using cognitive aids like a structural visualization.

Control Flow refers to the order in which the individual statements, instructions, or function calls of a program are executed or evaluated. Signaling aids, as mentioned earlier, clearly depict control constructs, control paths, and the overall structure of each program unit. This knowledge of control would be local to the program for an object oriented program (Corritone and Wiedenbeck, 1998) and be limited to sequence, branching and iteration. Hence, a programmer's view of control flow in his static mental

model is fragmentary, as dynamic aspects (data structure transformations, function/method calls, etc.) are not depicted by signaling aids and not represented in the static mental model. This view becomes cohesive when dynamic information is also incorporated into the programmer's internal representation of control flow during the building of the dynamic mental model. This is explained later.

Data Flow represents transformations that data elements undergo as they are executed in a program (Pennington, 1987b). Data flow analysis does not imply execution of the program under analysis (incorporation of information about program execution into the programmer's internal representation of data flow is explained later). Instead, the program is scanned in a systematic way and information about the use of variables is collected so that certain inferences can be made about the effect of these at other points of the program. This is often a difficult task because of data transformations which occur in delocalized plans (Soloway et al., 1988), i.e. plans whose elements are not physically contiguous but rather spread throughout the program text. Partial data flow is detected by the programmer through static analysis of the text of a program. Corritore and Wiedenbeck (1999) extended this knowledge of data flow to object oriented programming by including cases where (1) effects of one variable on another occurring either in the same program module or across module boundaries, and (2) how complex data structures are modified.

Control Structure represents the control constructs, control paths, and the overall structure of each program unit in a programming language. Modern programming languages examples of a control structure would be sequence, selection, iteration, exits and exception handling.

According to our cognitive model, during program comprehension, programmers build a *dynamic mental model* of the program from the program logic that they inferred

during the building of the static mental model, and any dynamic visualizations and dynamic windows that may be available in the IDE they are using, through the cognitive process of *integrating*. The dynamic model represents the communication between object instances at a high level of granularity and the communication between variables at a fine level of granularity. These relationships trace the delocalized plans and the local plans involved in the problem solution as implemented by the program. This model is generated by inferring and integrating the dynamic behaviors of individual program constructs. The generation of the dynamic model is aided by dynamic visualizations and/or dynamic windows. It consists of the following sub constructs.

Function refers to what the program does and is an important information paradigm in object oriented programming (Wiedenbeck & Ramalingam, 1999; Corritore & Wiedenbeck, 1999). Program execution is the main source of information about function. This is not fine grained enough for programmers to understand/debug programs. The required fine granularity is offered by debugging tools that allow line by line execution in synchrony with dynamic windows like output window or variable window. Output window, for example, displays error messages and exceptions in textual form during a program's step by step execution.

Dynamic Data Structure is a data structure that changes within the scope of the program. In an object-oriented program, it is a representation of the way objects execute their methods, representing the dynamic aspects of program execution (Guzdial and Ericson, 2010). In an object oriented environment, graphical representations are often used to display the data structure information. For instance, viewers in jGRASP present structural views of java collections classes. Some IDE's provide a propositional

representation of dynamic data structure, by displaying variable watch windows. If the variables are complex structures, they could be expanded to show their elements.

Dynamic Slice is a representation that contains all statements that actually affect the value of a variable at a program point for a particular execution of the program (Agrawal & Horgan, 1990). This is in contrast to all statements that could potentially affect the value of a variable at a program point for any arbitrary execution of the program (this is the Static Slice as explained earlier). This information for a variable can be extracted through step by step execution of the program and viewing the results synchronously in a dynamic window like the variable watch window. Agrawal and Horgan (1990) suggested that while debugging a program we try to find the dynamic slice of the program.

Control Flow concerns the sequence of actions that will occur when the program is executed, and the transformations that data elements undergo as they are processed (Pennington, 1987b). One common representation of control flow that most IDEs provide is the call stack browser. For example, the IDE VBCCE has a locals window with call stack browser along with other extensive debugging facilities. These windows present a list of threads/methods that, similar to the complex variables in the watch window, can be expanded to show the associated methods. In some IDE's this information is presented as a tree whose nodes are the methods executed and the parent-child hierarchical relation is determined by the program's calling sequence. Through the use of dynamic visualizations and windows provided by an IDE, the programmer's fragmentary view of control flow in his static mental model becomes cohesive with the inclusion of dynamic aspects.

Data Flow focuses on the dynamic aspect of threading data objects through the execution of the program (Pennington, 1987b). The internal data flow representation

built as part of the static model is enriched with the incorporation of information about program execution gleaned from dynamic visualizations and windows. Some dynamic windows show the path that data objects traverse as the program executes by showing lines joining variables within nested methods, e.g, Prograph (Matwin and Pietrzykowski, 1984).

4.2.3 Cognitive Process Flow

The Problem Statement is regarded as text from which the programmer must glean propositional and situational information and make critical inferences (Nathan, Kintsch, & Young, 1992). In programming, this statement is the specification of the program, i.e., what it is intended to accomplish. This need not be textual but can also be either verbal or pictorial, or a combination of both. The expected behavior of the program is derived based on the problem statement. When designing and coding, all the information is derived from problem statement, whereas when debugging or comprehending code written by someone else, equal information is derived from program code and problem statement (Gilmore, 1991).

Organization - When presented with a problem statement and program code, the programmer analyzes the code by identifying basic components of the program such as smaller chunks of code called static slices (Weiser, 1984), static data structures, and data and control flow of the program, and builds a static mental model. This process is facilitated by cognitive aids such as source code, structural visualizations, and signaling. This is analogous to the cognitive process of diagram decomposition (Narayanan & Hegarty, 1998). Mayer's (2003) selecting-organizing-integrating (SOI) model of text comprehension includes a similar process called organizing, in which relevant surface level information is combined into a coherent structure in working memory.

Determination of logic and data flow dependencies. During the building of the static mental model, two types of representational connections are established.

(a) *Connections to prior knowledge.* Narayanan & Hegarty (1998) proposed that during diagram comprehension a viewer identifies the components of the depicted machine and establishes relationships with his/her real world knowledge in the domain. For example, the viewer might represent that a circle in the diagram denotes a wheel and associate this with his/her prior knowledge about wheels. In the context of our model, structural visualizations like UML diagrams that depict the relationships and dependencies among classes aid in establishing connections to prior knowledge that the programmer has. For example, if a program is specific to a book repository, then the class diagram helps establish connections to real world knowledge about organizing books.

(b) *Connections to representations of other program segments.* Second, the user must represent the logical relations (i.e., relations regarding data and control flow) among different program components by building mental connections that encode these logical relationships among his/her internal representations of multiple program modules or classes. Cognitive aids help programmers establish such connections. For instance, CSD, a signaling aid, helps establish this by providing explicit visual information about control constructs and control paths to allow the programmer to establish relationships among different components of the same program. This corresponds to the step of establishing connections that encode spatial relations among components of a machine in the cognitive model of Narayanan & Hegarty (1998).

For machine diagram comprehension, this encoded knowledge of spatial relations aids in guiding the viewer's reasoning process along the chain of causality (called "lines of action") in the operation of the machine (Hegarty, 1992; Narayanan & Hegarty,

1998). Similarly, we postulate that the encoded knowledge of logical relations among program components resulting from the above step facilitates a reasoning process for logic flow and data dependence, through which the programmer determines the "logical lines of action" in the code. This reasoning process of the programmer is termed "determination of logic flow and data dependence in program code" in Figure 11. It helps reduce the mental computation required for predicting a program's behavior while creating the dynamic mental model. For example, the programmer might predict the change in attribute values of an object during the execution of a method.

Integration - Based on the static mental representation and determination of logic flow, the programmer now creates a dynamic mental model. This last and final step, termed integrating (Mautone et. al., 2007), involves constructing a dynamic mental model of the program by inferring and integrating the dynamic behaviors of individual program components. Narayanan and Hegarty's cognitive model of text and diagram comprehension (1998) includes a similar stage. Program perspectives like function, dynamic data structure, data flow and dynamic slice emerge during this incremental process. It involves constant restructuring of the mental representations by hypothesizing a module's logic and validating its operation, leading up to an integrated representation of the dynamic aspects of program execution. During this iterative process, relationships between the existing static model and internal dynamic representations are established by stepping through the program execution with the assistance of cognitive aids like dynamic visualizations and windows. Narayanan and Hegarty (1998) argued that referential connections are crucial, during text and diagram comprehension, to constructing an integrated internal representation of the common referent of text and diagram in memory as opposed to separate representations of the text and diagram. This applies to program comprehension as well, given that multiple

representations (e.g., a snippet of code and data structure visualization) could represent the same entity and hence their internal coordination is vital. Thus, we propose that program comprehension results in a dynamic mental model of program execution.

Building on this model of program comprehension, we now propose a cognitive model for program debugging with forward reasoning (Figure 12). This extended cognitive model introduces a new mental model called Posit Dynamic Mental Model and a cognitive process called Hypothesis Testing.

Posit Dynamic Mental Model – The program comprehension process described above produces a dynamic mental model that captures the dynamic aspects of program execution. If the program executes correctly, this mental model, which is in part derived from external dynamic representations of program execution such as dynamic visualizations and windows, correctly captures both static and dynamic aspects of the program that is comprehended. However, if the program is buggy, the execution data it produces will be erroneous. Therefore, the dynamic mental model created through the process of integration would be that of an erroneous program. Therefore, we postulate that the programmer generates two dynamic mental models if he/she is engaged in debugging as opposed to just program comprehension. One is the dynamic mental model described previously, which encodes the erroneous execution of the program. In addition, the programmer would generate a second dynamic mental model of the expected (correct) behavior of the program from his/her static mental model and determination of logic flow and data dependencies. This dynamic model, which we call the posit dynamic mental model, would not be based on the external representations of buggy program execution such as dynamic visualizations or windows. The posit model is similar to the dynamic mental model in terms of the sub constructs that constitute this model, but these constructs are predicted or inferred from their counterparts in the

static mental model, as opposed to verified from external representations. This model is later used by the programmer to compare with the dynamic mental model produced from actual (and buggy) program behavior based on the programmer's debugging hypothesis (Hypothesis Testing).

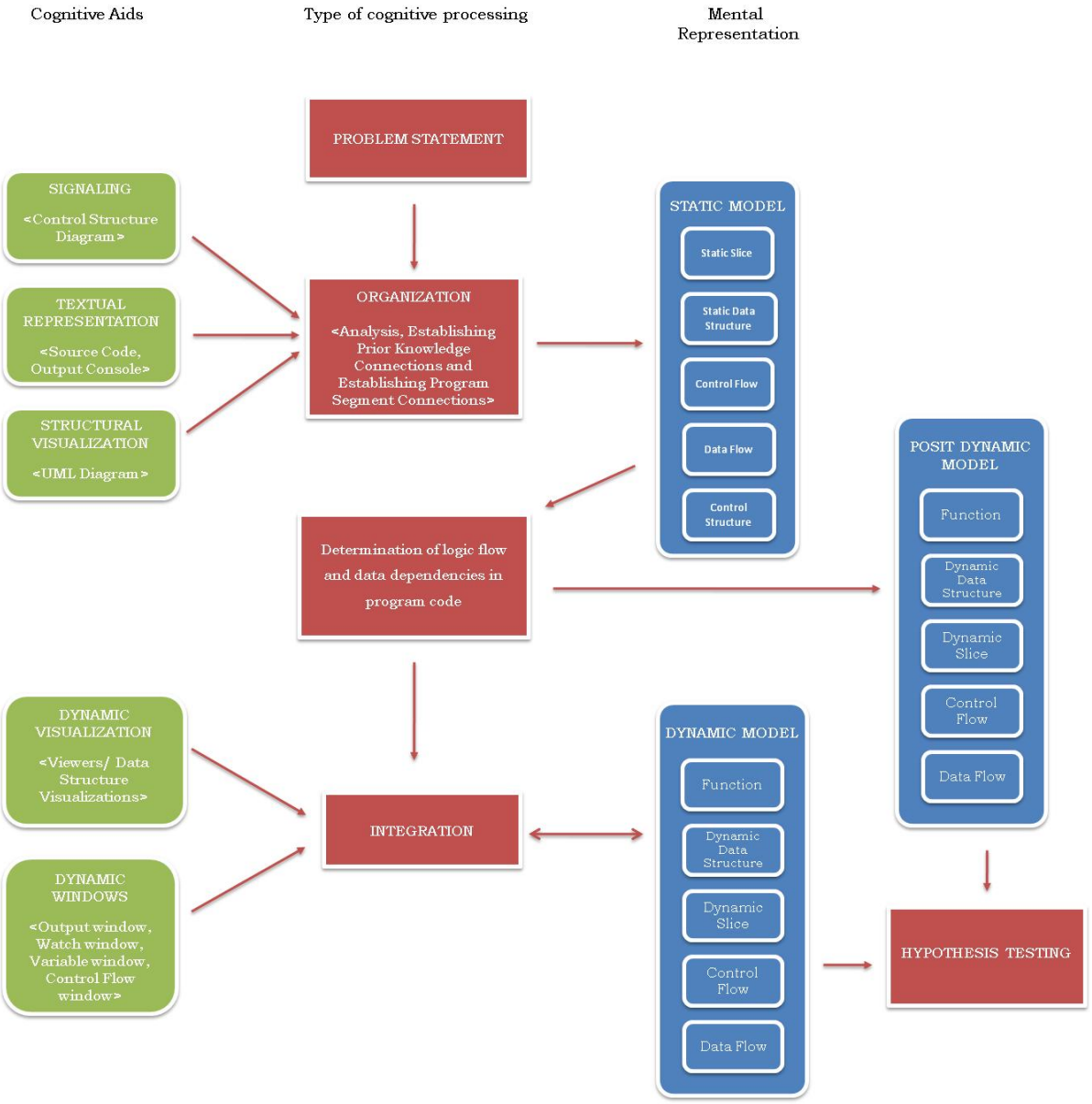


Figure 12. Cognitive model of multi representational program debugging

Hypothesis Testing - Hypotheses are key drivers in program understanding and influence the direction program understanding can take (Mayrhauser and Vans, 1997). A hypothesis about the program component or behavior causing the bug or error helps detect the difference between the desired behavior from specification and the behavior performed by the program. According to Araki et al. (1991), in locating the errors and grasping their causes, programmers develop hypotheses about the errors and their causes, and verify or refute these hypotheses by examining the program. During dynamic analysis, a programmer executes the program with appropriate input data and examines its behavior and output (Gould and Drongowski, 1974). Many consider correction of errors found through dynamic analysis to be debugging, and often use debugging tools to execute dynamic analysis. We propose that during hypothesis testing, the posit dynamic mental model is compared with the mental model created from actual program behavior and this leads to either no action or acceptance or rejection of the hypothesis.

Hypothesis testing would produce several external behaviors, such as stepping through the program, inspecting dynamic visualizations or windows, etc. The programmer may also modify the code to achieve the desired program behavior, and test it by executing it. We expect that the dynamic mental representations will change after the programmers have made significant modifications to the program over time. In particular, we expect a convergence between the dynamic mental model and the posit dynamic mental model. This is similar to a cross referenced or mixed representation (Pennington, 1987a, b).

Though our model of program comprehension and debugging is derived from extant research on text, diagram and graph comprehension as well as program comprehension, its constructs and processes need empirical validation. It is not yet clear

how constructs of the different mental representations we have proposed influence the debugging performance. We also suspect that co-ordination of these multiple representations is an important expert skill in debugging, and could be a potential problem for novice programmers. More theoretical and empirical knowledge about the way these representational systems influence the comprehension and debugging of computer programs is therefore needed. This leads to a variety of research questions, only some of which (as explained in the next chapter) are addressed by the present research:

- Do cognitive aids lead to constructs other than the ones represented in the static and dynamic models?
- Are all visualizations preferred/used equally by the programmer or is any visualization preferred more than the others?
- Does the modality and perspective of a representation in the cognitive aids influence its effectiveness or preference?
- How does the depth and/or accuracy of posit dynamic model affect debugging performance?
- Are there any particular patterns in representation use, which leads to superior debugging performance?
- To what extent do programmers use each type of representation?
- Under what circumstances do programmers switch between representations?
- Are graphical representations more helpful to Java programmers (because of the OO paradigm) than textual ones?
- Are representations that highlight data structures more useful than those that highlight control-flow for Java debugging?

- Do graphical visualizations promote a more judicious representation use than textual ones for program debugging in a multi-representational IDE?
- Do representation characteristics such as the information type highlighted or its format (graphical or textual) affect representation use and debugging strategy employed?
- Is there a relationship between programmers' cognitive characteristics such as visual vs. verbal, their level of familiarity with representation formalisms, format preference and programming experience and their debugging behavior?
- Does higher interactivity with the IDE lead to a better debugging performance ?
- Do participants with a high level of debugging skill interact less with the visualizations?
- What is the extent to which novices and experts exhibit forward reasoning vs. backward reasoning in their debugging strategies?.
- How effective are individual IDE representations (e.g. the CSD of jGRASP) in aiding debugging?
- Is there a difference between the step by step comprehension and debugging activities of novices and experts? If so, how can our model account for the differences?

CHAPTER 5

SCOPE OF RESEARCH

The cognitive models of program comprehension and debugging pose several intriguing questions as discussed in the previous chapter. We selected a few of these questions for experimental investigation using the jGRASP IDE. In this chapter, we discuss each selected research question and our approach toward data collection and analysis for answering the question.

Research Question 1: How does the depth of the mental model affect debugging performance?

According to our cognitive model, while debugging, students first construct a static mental model of the program, and then during program comprehension stage derive a dynamic mental model of its execution called the posit dynamic model. This is followed by the construction of a third mental model representing the buggy program, which is constructed while interacting with various IDE visualizations. A comparison between this buggy model and the posit model of expected program behavior allows a programmer to locate the bugs in a program. The key to a programmer's performance while debugging is the depth or strength of the mental models they create. This research question delves more into establishing a relationship between strength of the mental model to debugging performance. The independent variable here is the depth of the model and the dependent variable is the debugging performance. As there is no direct and complete measure of a programmer's mental model, questionnaires will be used

to measure the depth of knowledge regarding programming constructs (mental representations) of the program being debugged. For example, the question "what does node 'p' refer to after 3 iterations in the 'move' method of the List class?" addresses knowledge of data flow in the programmer's dynamic model (see Appendix D for the complete set of questions). Each response will be scored as either correct or incorrect. Each correct response will add one point to the total score. In order to measure the debugging performance of a participant, we will use a scale shown in the Table 5.1.

Time to completion	**No of Bugs found**	**Performance scale**
Before the end of 15 minutes	4	4
End of 15 minutes	3	3
End of 15 minutes	2	2
End of 15 minutes	1	1
End of 15 minutes	0	0

Table 5.1 Debugging performance measurement scale

Pearson correlation (or Spearman correlation) will be used to establish the correlation between the mental model strength and the debugging performance.

Research Question 2: How is the depth of the mental model built from static visualizations different from that resulting from the dynamic visualizations?

Expanding further on the previous research question, we will investigate how the depth of the mental model is affected by availability/use of either dynamic or static visualizations. This will be achieved by creating two groups of participants, one will be allowed to use only static visualizations while debugging whereas the other will be allowed to use only dynamic visualizations. The mental model strength of programmers

will be measured at fixed intervals using questions as discussed above. Here, 'visualization type' will be the independent variable and mental model strength will be the dependent variable. A repeated measure ANOVA will be conducted on mental model strength for the time intervals to answer this question.

Research Question 3: How do the components of the mental model (in terms of various internal representations/constructs) built from static visualizations differ from those resulting from dynamic visualizations?

The proposed cognitive model postulates that static visualizations lead to mental representations or internal constructs static slice, static data structure, control structure, minimalist control flow and minimalist data flow, and dynamic visualizations lead to the internal constructs dynamic slice, dynamic data structure, function, control flow and data flow. In order to answer this question, we have to look at the relationship between type of cognitive aid and the mental model strength for each mental representation. The independent variables here are the visualization type and the internal constructs. The dependent variable is the mental model strength, which will be measured by the questionnaire discussed earlier. A two-way ANOVA with repeated measure on one factor will be conducted to determine whether there is a statistically significant difference between the two visualization types (static and dynamic) in influencing mental model strength of different internal constructs.

Research Question 4: Is there a difference in the programmers' usage of static and the dynamic visualizations? Does this usage difference lead to a performance difference?

In order to answer this question, the usage of static and dynamic cognitive aids (i.e., representations provided by the IDE) by the programmer in the course of

debugging has to be measured. We will use eye tracking data for this purpose. Each type of representation will be marked as an Area-of-Interest (AOI). Dwell time on each AOI will be a measure of representation usage. A simple independent samples t-test between two groups with access restricted to either only static or only dynamic visualizations will be performed. This will be performed for each representation type, Program Code, Visualization and Output. Further we will look at the difference between mean fixation duration for each AOI between the two groups. As longer fixation signifies difficulty in interpretation, this will give us an insight into programmer behavior in the two groups. Again, a simple independent samples t-test between two groups for the three AOI's will be conducted. A t-test comparing the debugging performance (based on Table 5.1) of the two groups will help answer the second part of this question.

Research Question 5: Is any representation (cognitive aid) preferred more than the others?

To answer this question, experiment participants will be given unrestricted access to all the representations available with the IDE. The question can then be answered in part by analyzing the visual attention of participants and in part by evaluating participant's interview responses. Each visualization will be defined as an AOI. The visual attention attributes (dependent variable) considered here will be average dwell time, average fixation count, and visit count for each available representation (independent variable). Analysis can get fairly complex with multiple AOI's in question. Hence, we will be using the table below (Table 5.2) for grouping similar AOI's into four categories.

AOI Categories	**Consisting of**
Code	All classes of the program
Static Visualization	CSD, UML
Dynamic Visualization	jGRASP Viewer, Variable Watch Window, Expression Evaluation Window
Output	All textual representations of program results

Table 5.2 AOI Categories

For each visual attention attribute, we will conduct an ANOVA resulting in three ANOVA analyses. As a fallback back strategy in case of sparse data where all the representations are not attended to, we will switch to binomial analysis as observed in earlier research (Bednarik, 2005).

In addition, qualitative results from interview responses can provide good insight into representation usage, further substantiating the results. The questions for each individual participant will be framed based on the strategy employed by him/her during the debugging session. For example, if a participant was seen to have used jGRASP viewers a lot, questions on that, such as the following examples, will be asked. Why was the Viewer used? Why did you think it was appropriate? Was it helpful in the end? Does the fact that jGRASP viewer shows you real time manipulations help you? Why did you not use the UML diagram? Why did you choose to use viewer over the variable window to debug? (See Appendix D1 for the semi-structured interview questions.)

Research Question 6: How do programming experience, familiarity with the IDE and debugging performance influence the strategies employed in representation use during debugging?

The independent variable here are programming experience, familiarity with IDE and debugging performance. We will categorize experiment participants under these based on the criteria summarized in Table 5.3

Independent Variable	**Categories**	**Criteria**
Programming Experience	Novice	Less than 12 months of programming experience in Java
	Expert	More than or equal to 12 months of programming experience in Java
jGRASP experience	Low	Less than 6 months of experience with jGRASP IDE
	High	Greater than or equal to 6 months of experience with jGRASP IDE
Debugging Performance	Bad	The programmer was not able to debug all 3 bugs from the assigned task
	Good	The programmer successfully debugged all 3 bugs from the assigned task

Table 5.3 Independent variable categorization

In order to better understand the strategies of participant programmers, the dependent variable, the visual patterns of each participant will be coded as character strings that represent short or long gaze durations on the previously discussed four categories of AOIs (see Table 5.4).

Character Representation	**Gaze Duration on AOI (*Short*)**	**Character Representation**	**Gaze Duration on AOI (*Long*)**
A	Code	B	Code

C	Static Visualization	D	Static Visualization
E	Dynamic Visualization	F	Dynamic Visualization
G	Output	H	Output

Table 5.4 Gaze duration based AOI categorization

Attention on each AOI will be categorized as a short duration gaze if the duration of each visit to an AOI is lower than a certain threshold value (in ms), it will otherwise be categorized as a long duration gaze. Thus, we will have eight categories of visual attention. For example, a string AFG translates to a programmer spending a short duration of time on code followed by a long duration on a dynamic representation, further followed by a short duration on output. The string AEAEAEAE…. represents frequent switching between code and dynamic visualization with short gazes on the two AOI's. In order to separate short and long durations, we will use the threshold value of 500ms, since it is known that at least 200 ms are needed for a fixation and more than one fixation is needed for cognition. Recurring patterns from these sequences will be algorithmically analyzed to understand the underlying strategies and to answer this research question.

Chapter 6

EXPERIMENTAL DESIGN AND PROCEDURE

To test the hypotheses arising out of our research questions, we conducted an experiment using a remote eye tracker to record the gaze behavior of participants during a program debugging task aided by multiple representations that the jGRASP IDE presents. Their gaze behavior and other data provided us with specific knowledge of how explicit areas of the jGRASP IDE were used by different programmers and how it influenced their mental model construction and debugging performance.

6.1 Method

6.1.1 Participants

The participants in the experiment were graduate and undergraduate students from the department of Computer Science & Software Engineering at Auburn University who had a minimum of 6 months programming experience in Java. All participants were volunteers and received $10 for each hour of their participation. We recruited 19 participants, 2 female and 17 male, all with normal or corrected vision. None of them had previously participated in an eye tracking study. Their level of programming experience varied, ranging from a sophomore in computer science having taken or currently enrolled in a data structures class to graduate students who had substantial programming experience, with some who had professional experience in building enterprise applications in Java. The median and mode of general programming experience was 1 to 2 years. The median for Java programming experience was 1 to 2

years, and the mode was 6 to 12 months. Four of the participants had never worked with jGRASP before and of those who had prior experience with jGRASP, all but one participant had used jGRASP for a minimum period ranging 6 - 12 months. Demographic details of the participants are listed in Appendix A.

6.1.2 Materials and Apparatus

Two short Java programs – string reversal using stacks (program 1) and binary search on a doubly linked list (program 2) were developed. Program 1 was seeded with 4 bugs and 3 bugs were introduced in program 2. The errors can be classified as control flow, data flow, data structure and functional errors. Details of the two programs and bugs are provided in Appendix B. Participants were notified that there were no syntactical errors in the program. On execution, the program was designed to display the expected output and the current output. In addition, a warm up program was used to familiarize participants with the IDE and the visualizations available with it. The names of the methods, variables and class names were altered so that recognition of a program and the underlying data structure based on surface features would be difficult. These programs were debugged by the participants using the jGRASP IDE, during which their eye movements were tracked. We used a Tobii T60 XL, a remote and unobtrusive eye tracker with sampling rate set to 60Hz. This eye tracker was set up in a sound proof laboratory with consistent fluorescent illumination. Participants were seated comfortably in an ordinary office chair, facing a twenty four inch TFT widescreen monitor and maintained a viewing distance of 55-65cm. The screen resolution was set to 1920 x 1200. Tobii Studio™ 2.1 was used for setting up the experiment. The stimuli sequence was created by combining all the debugging tasks into one jGRASP project. Tobii Studio™ was also employed to create a holistic view of user behavior during debugging by integrating data captured from the recording of eye tracking data with

user video, screen capture, sound, keystrokes and mouse clicks. During the experiment, user actions were supervised on a remote computer using Tobii Studio Logger™, which displayed the test screen with real time gaze data overlay.

6.1.3 Procedure and Design

After becoming familiar with the experiment and signing a consent form, the participants were given 10 minutes to understand the functionalities of jGRASP IDE. This was cut short if a participant had prior experience with jGRASP. Following this, two debugging sessions of 15 minutes each were administered. Prior to each debugging session, the participant had to pass an automatic eye tracking calibration routine, which consisted of tracking their eyes as they followed nine points on the computer screen. This process was repeated if necessary, to achieve good accuracy and precision. Each session was split into two sections for each of the two programs. Each section consisted of two phases; first a description of the program to be debugged was presented. Next the participants were asked to locate the bugs in the code and fix them within a time limit of 15 minutes.

While debugging the first program, subjects were allowed to use only static or only dynamic visualizations depending on their grouping. Participants were assigned program 1 (string reversal using stacks) for debugging. This section of the experiment was designed to help us answer Research Questions 1, 2 and 3. During this session, a participant's mental model strength was measured at regular intervals with a questionnaire on his/her knowledge of the program constructs. The questionnaire consisted of 23 questions (see Appendix C) pertaining to program constructs: function, control flow, program structure, static slice/dynamic slice and control structure. The questions were objective with the response scored as either correct or incorrect except for one question on static/dynamic slice that was subjective. The questionnaire was

administered after every 5 minutes, leading to 3 measurements after 5 minutes, 10 minutes and either at the end of session or after 15 minutes, whichever came earlier. The verbal response of a participant to each question was audio-taped and later scored and tabulated by the researcher. The order of questions was randomized for every interval to counter learning from the ordering of questions. If a participant fixed all the bugs within 10 minutes, he/she was administered only two questionnaires.

The same procedure was followed for the second program. First a description of the program (binary search on a doubly linked list) to be debugged was presented. Next, the participants were asked to locate the bugs in the code and fix them. For this program, they were allowed to use any of the visualizations available with the IDE, which included the dynamic representations and the dynamic windows. The questionnaire was not administered for this experiment, and students were given 15 continuous minutes to debug the program. On completion of the debugging sessions, each participant was interviewed based on a semi structured interview protocol (see Appendix D). In order to counter confounding factors like fatigue, learnability etc., half of the participants were assigned program 1 first and the other half program 2 first. A pilot study was first conducted with three volunteers. Minor issues were unearthed based on volunteer feedback and researcher's observations. These issues were fixed before the actual experiment.

For performing gaze analysis, Areas of Interests (AOIs) were defined corresponding to different visualizations, menu bar, file browser and animation controls of jGRASP. With the first program, all of the program code was visible to participants on the screen. Participants who were given only static visualizations did not have to move any windows and hence the screen AOI's remained constant throughout (Figure 13).

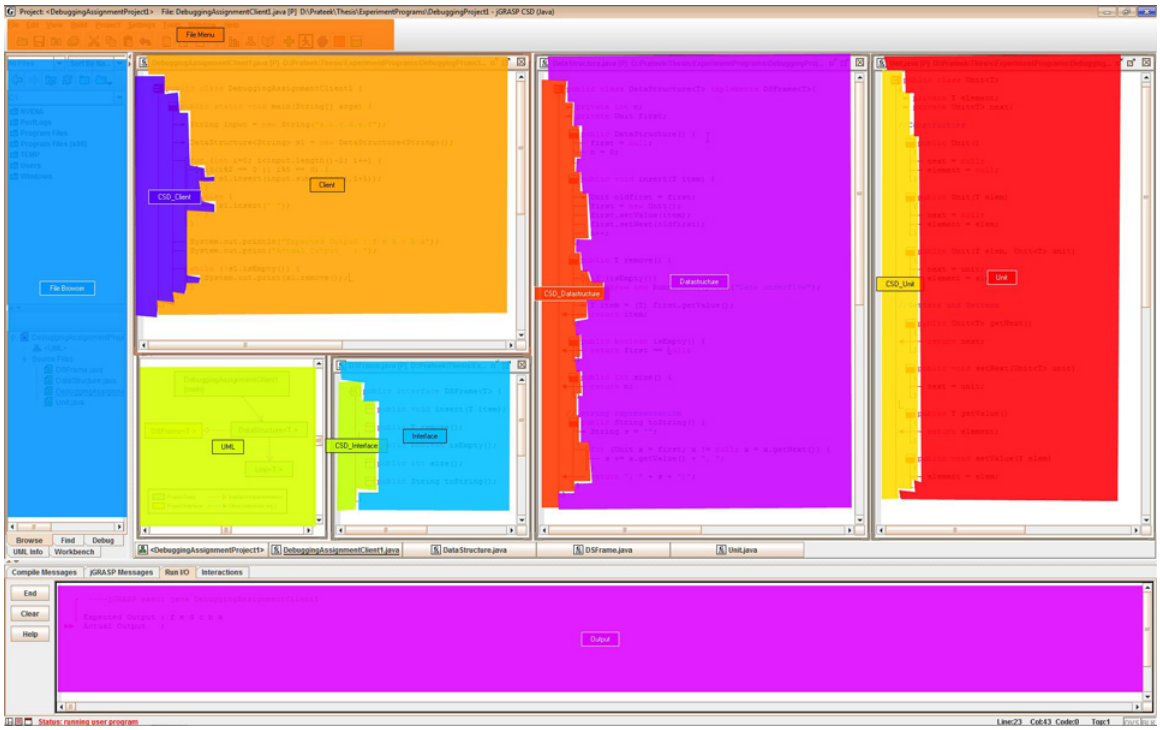


Figure 13. AOI's defined for Experiment 1 with Static Visualizations

However, for the dynamic visualization group, the AOI's changed over time due to the movement of windows by programmers and hence the complete debugging session was broken down into multiple segments for each participant (see Table 6.1). Each segment represented a single scene in which all the windows were positioned at fixed locations on screen. For each instance of a window moved and positioned at a new location on screen, a new segment was created. Figures 14 and 15 are snapshots of continuous scenes extracted from one of the debugging sessions for program one and figure 16 for program two. Gaze data from each of the segments were later combined. The same approach was taken for debugging program 2 as the windows were moved around the screen during a session. There were 11 possible AOI's of interest; Animation Control, Client Code, Client CSD, Data Structure CSD, Data Structure Code, Dynamic Window, Eval Window, File Browser, File Menu, Output, and Variable Watch.

Participant	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19
No. of Segments	7	10	7	5	5	7	7	4	8	2	4	9	11	2	8	11	10	7	11

Table 6.1 Segment wise break up for each participant

Figure 14. AOI's for Experiment 1 (no Visualization in use)

Figure 15. AOI's for Experiment 1 (dynamic visualizations in use)

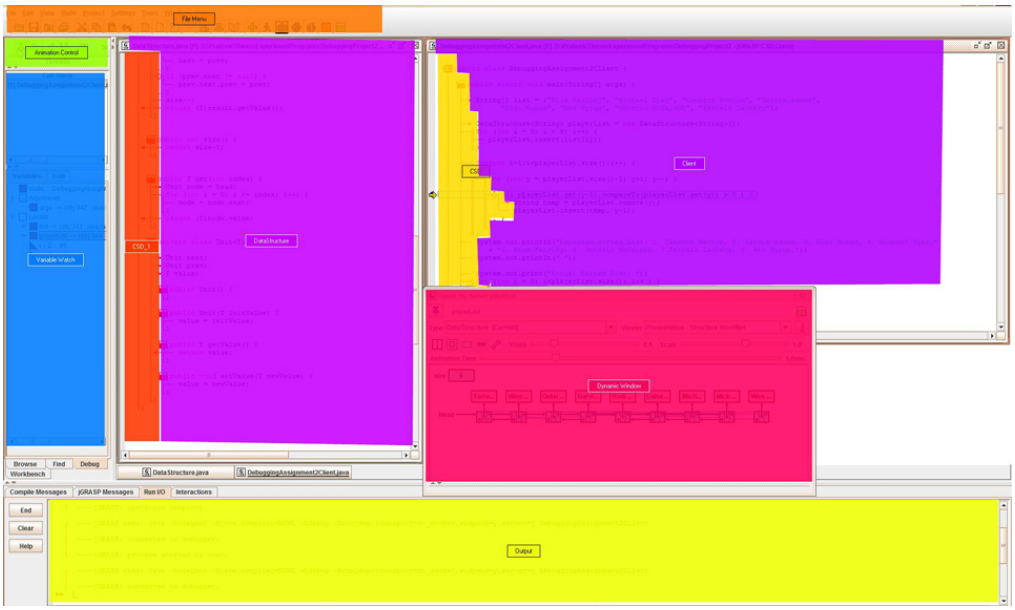


Figure 16. AOI's for Experiment 2 (dynamic viewer, CSD & variable watch in use)

For the experiment with program 1, we used a mixed design with one between-subjects factor (representations available for debugging) and four dependent variables (number of errors spotted, accumulated fixation time, mean fixation duration, and switching frequency, as measured by the eye tracker). Specifically, the within subject factor was static visualizations for one group, and dynamic visualizations for the second group. The accumulated fixation time is the total time a participant spent during a session fixating on an area of interest (AOI). For an AOI, all of the fixation durations were added, and the number was divided by the total fixation count throughout the debugging session, giving the mean fixation duration. Most of the results were analyzed by performing either ANOVAs and/or planned paired t-tests. These data analyses and their results are presented in the next chapter.

Chapter 7

RESULTS

We now discuss the results of our analyses based on data collected from the experiments described in Chapter 6.

Research Question 1: How does the depth of the mental model affect debugging performance?

For this analysis, we tabulated participant scores based on their responses to the questionnaire at the end of the debugging session, along with their corresponding performance score (based on table 5.1). Pearson correlation and Spearman correlation were used to establish the correlation between the mental model strength and the debugging performance. We first looked at the data collected from all 19 participants. According to Pearson's correlation, there was a positive correlation between the two variables, $r(17) = .56$, $p < .05$ with $R^2 = .31$.

Based on Pearson's correlation, it can be concluded that there was a moderate correlation between mental model strength and debugging performance. As the depth of the mental model increased, the debugging performance too increased. The percent of variability is relatively low with only 31% of debugging performance related to mental model strength, 69% remains unexplained. We further evaluated the correlation among these variables for the static and dynamic visualization groups separately. This correlation between mental model strength and performance was stronger for the

dynamic visualization group ($r(8) = .66$, $p < .05$) and was statistically significant. The static visualization group showed a weaker correlation that was not statistically significant ($r(7) = .48$, $p = .19$).

Research Question 2: How is the depth of the mental model built from static visualizations different from that resulting from the dynamic visualizations?

To evaluate this research question we continue using data from debugging program 1. As discussed in Chapter 5, the mental model strength was measured at 3 intervals. A repeated measures ANOVA on the difference in mental model strength for the 3 data collection points (after 5, 10 and 15 minutes) was statistically significant ($F_{0.05(2,26)}=64.52$, $p<0.001$).

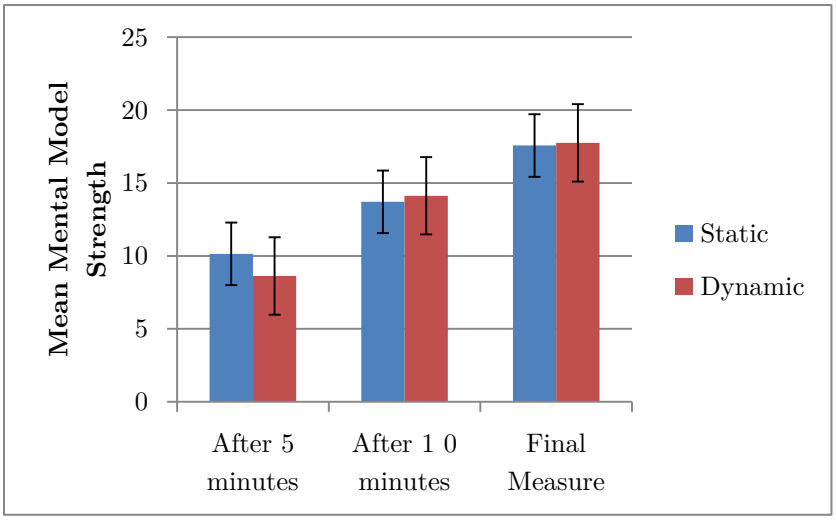


Figure 17. Mean mental model strength

The effect size was large, with $\eta^2=0.89$ and the observed power 1.00. Statistically significant differences ($p<.05$) were found between all three pair-wise comparisons of mental model strength. The interaction between the visualization type (static or dynamic) and mental model strength was not statistically significant ($F_{0.05(2,26)}=6.99$,

ns). For this analysis N=15, as 4 participants completed debugging within 10 minutes and hence their data was not included in the analysis.

Next, the final mental model strength of all the participants was evaluated. Although the mean value of mental model strength was higher for the dynamic visualization group (M=18.6, SD=2.41, N=10) when compared to static visualization group (M=17.33, SD=2.29, N=9), there was no statistically significant difference in the mean values $t(17)$=-1.17, *ns*).

Research Question 3: How do the components of the mental model (in terms of various internal representations/constructs) built from static visualizations differ from those resulting from the dynamic visualizations?

We first computed the mean values for each programming construct at the 3 stages of mental model measurements. As expected the mental model strength increased for each programming construct as summarized in figure 18.

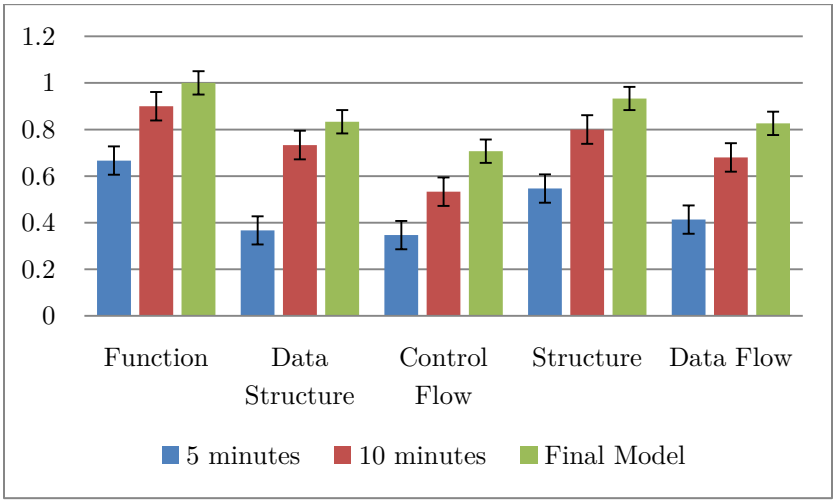


Figure 18. Mean Value of mental model strength (N = 19)

This was followed by focusing on individual constructs, leading to 5 different analyses. A two-way ANOVA with repeated measure on one factor was conducted to determine whether there was a statistically significant difference between the two different types of visualizations (static and dynamic visualization) for influencing mental model strength. This analysis was performed for each of the five programming constructs that was measured. The independent variable included a between-subjects variable, the visualization type, and within-subject variable, repeated measures of time. The dependent variable was the strength of mental model for a programming construct. An alpha level of .05 was utilized for these analyses.

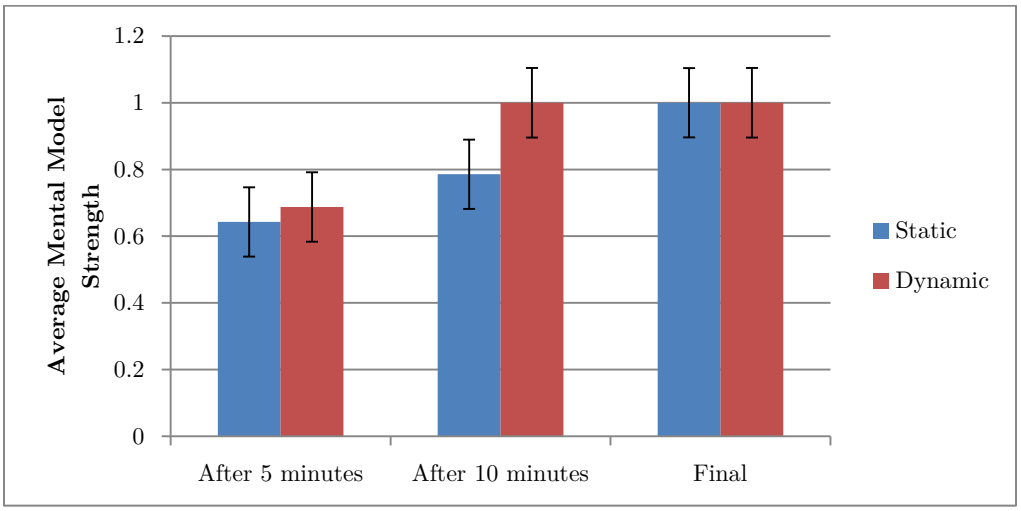


Figure 19. Average mental model strength - Function

Function – The result of main effect of three measurements of mental model for function was statistically significant (Wilk's Lambda), $F_{0.05\ (2,\ 12)}$=10.4, p<0.05, η^2 =.63 and power = 0.96. A large effect size was evident. There was no statistically significant interaction in the strength of mental model between the visualization type and the measurement time, $F_{0.05\ (2,\ 12)}$=2.47, p=.127, η^2 =.291 and power = 0.4. Although, there

is no statistically significant in mental model strength at these different stages, we suspect that there is a possibility of reaching statistical significance if N is increased. This can be derived from the fact that the power is low and Partial Eta Square is strong. Perhaps, ceiling effect is also a factor in the final stage, as the measurement reached the maximum in the second measurement for the dynamic visualization group and remained at that level through the final stage. Although the mean value of the measurement for the dynamic group was higher than the static group at all three stages, there was no statistically significant main effect in the visualization type either, $F_{0.05\ (1,\ 13)}$=1.93, p=.129, η^2 =.13 and power = 0.25, which was indicative of a moderate to large effect size.

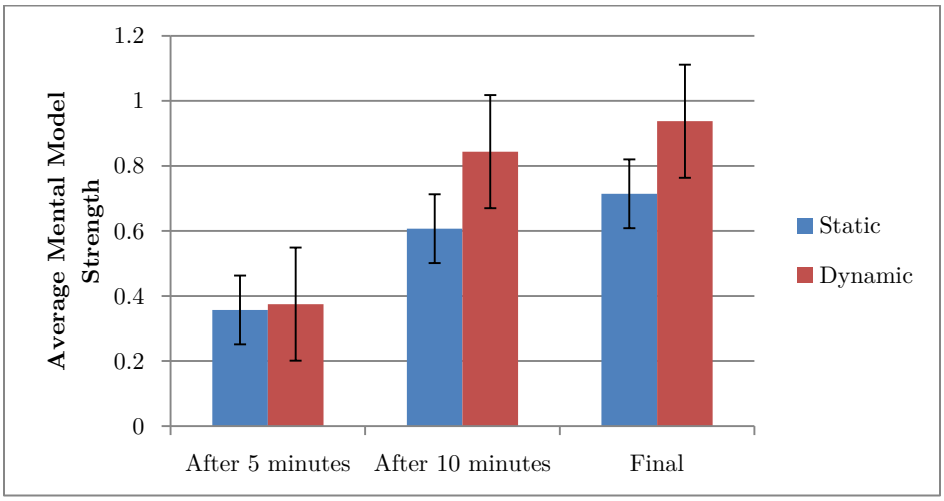


Figure 20. Average mental model strength – Data Structure

Data Structure – Here again the result of main effect of three measurements of mental model for function was statistically significant (based on Wilk's Lambda), $F_{0.05\ (2,\ 12)}$=28.35, p<0.001, η^2 =.83 and power = 1.0. A large effect size was evident. There was no statistically significant interaction in the strength of mental model between the

visualization type and the measurement time, $F_{0.05\ (2,\ 12)}$=1.98, p=.18, η^2 =.25 and power = 0.33. Here again, there is no statistically significant difference in mental model strength at these different stages, and we suspect that there is a possibility of reaching statistical significance if N is increased, given that the power is low and Partial Eta Square is strong. There was no statistically significant main effect in the visualization type either, $F_{0.05\ (1,\ 13)}$=2.8, p=.12, η^2 =.18 and power = 0.34, which was indicative of a moderate to large effect size, but it was found that the mean score for mental model strength of dynamic group was consistently higher than static group at all three stages.

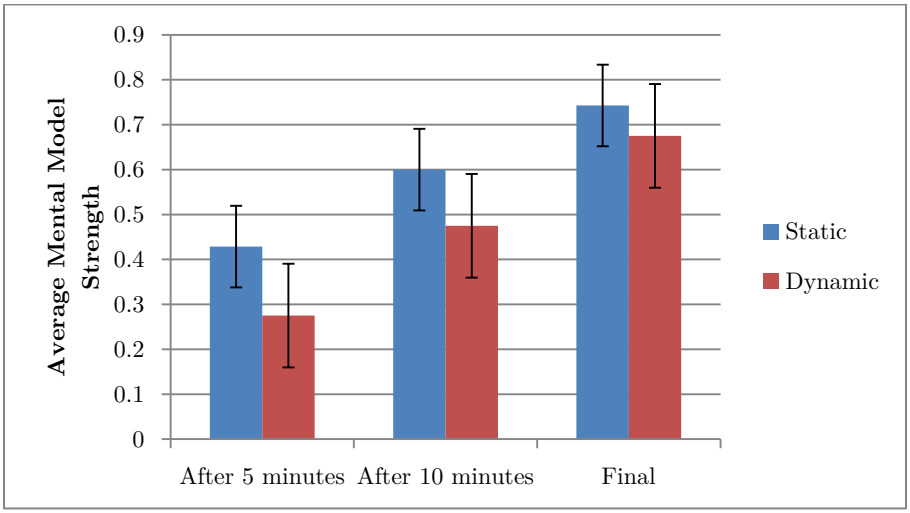


Figure 21. Average mental model strength – Control Flow

Control Flow – Here again the result of main effect of three measurements of mental model for function was statistically significant (based on Wilk's Lambda), $F_{0.05\ (2,\ 12)}$=15.85, p<0.001, η^2 =.73 and power = 0.99. A large effect size was evident. There was no statistically significant interaction in the strength of mental model between the visualization type and the measurement time, $F_{0.05\ (2,\ 12)}$=.361, p=0.7, η^2 =.05 and power = 0.1. The mental model strength for static group was consistently higher than dynamic group at all three stages, but there was no statistically significant main effect in the

visualization type either, $F_{0.05\ (1,\ 13)}$=.32, p=.58, η^2 =.02 and power = 0.08, which was indicative of a small effect size.

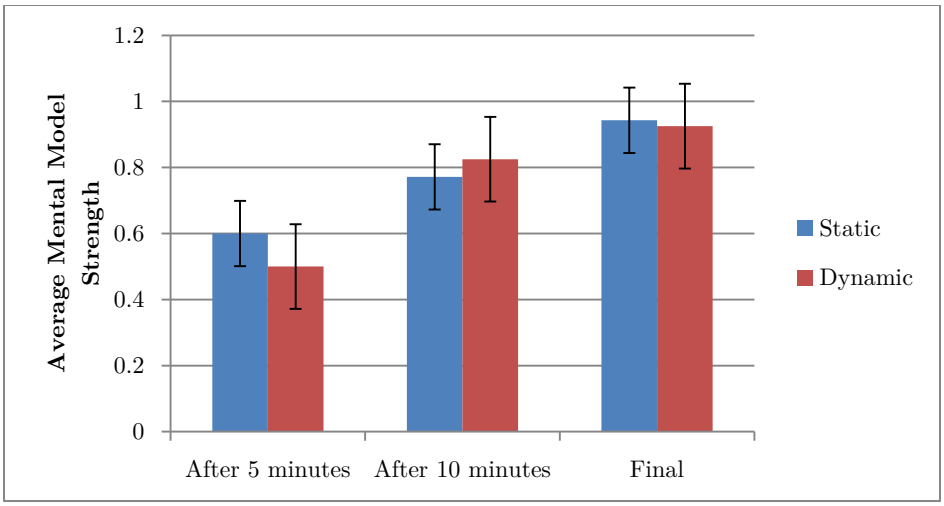


Figure 22. Average mental model strength - Structure

Structure – Here again the result of main effect of three measurements of mental model for function was statistically significant (based on Wilk's Lambda), $F_{0.05\ (2,\ 12)}$=14.93, p<0.05, η^2 =.71 and power = 0.99. A large effect size was evident. There was no statistically significant interaction in the strength of mental model between the visualization type and the measurement time, $F_{0.05\ (2,\ 12)}$=.5, p=0.62, η^2 =.08 and power = 0.11. There was no statistically significant main effect in the visualization type either, $F_{0.05\ (1,\ 13)}$=.06, p=.81, η^2 =.004 and power = 0.06. The mental model strength for the two groups fluctuated and no consistency was found.

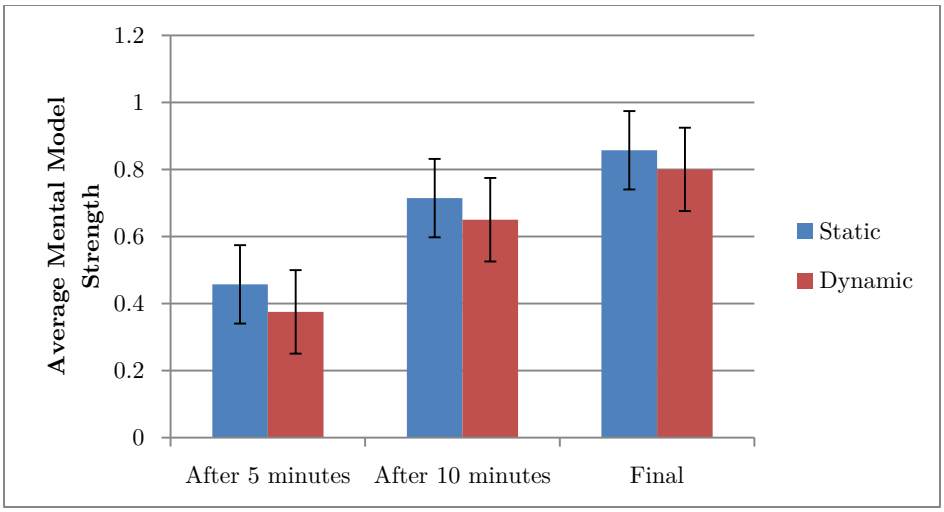


Figure 23. Average mental model strength – Data Flow

Data Flow – Here again the result of main effect of three measurements of mental model for function was statistically significant (based on Wilk's Lambda), $F_{0.05\ (2,\ 12)}$=11.6, p<0.05, η^2 =.66 and power = 0.97. A large effect size was evident. There was no statistically significant interaction in the strength of mental model between the visualization type and the measurement time, $F_{0.05\ (2,\ 12)}$=.01, p=0.99, η^2 =.002 and power = 0.05. Although the mental model for static group was consistently higher than the dynamic group at all three stages, there was no statistically significant main effect in the visualization type either, $F_{0.05\ (1,\ 13)}$=.38, p=.55, η^2 =.03 and power = 0.09, which was indicative of a small effect size.

Research Question 4: Is there a difference in the programmers' usage of static and the dynamic visualizations? Does this usage difference lead to a performance difference?

An independent-samples t-test was conducted to compare Dwell Time under static and dynamic conditions. This was performed for three different AOIs: code, visualization and output. There was no statistically significant difference in the dwell

time for code between Static Visualization (M=51.32, SD=2.82) and Dynamic visualization (M=50.13, SD=5.87) conditions; $t(17)$=.55, p = 0.59. Likewise no statistically significant difference was found for output between Static Visualization (M=3.70, SD=1.63) and Dynamic visualization (M=4.28, SD=1.63) conditions; $t(17)$=-0.75, p = 0.46.

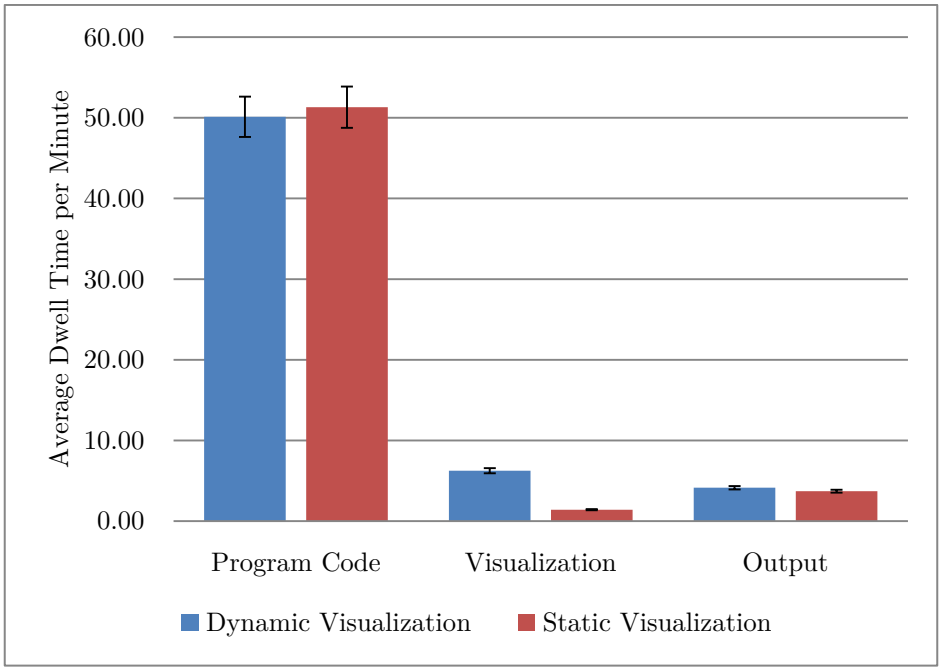


Figure 24. Average Dwell Time per minute

There was a statistically significant difference in the dwell time for visualization between Static Visualization (M=1.41, SD=.57) and Dynamic visualization (M=6.24, SD=3.02) conditions; $t(17)$=-4.72, p <.001.

An independent-samples t-test was conducted to compare Fixation counts under the static and dynamic conditions. This was performed for the same three AOIs. There was no statistically significant difference in the average fixation count for code between

Static Visualization (M=121.04, SD=9.75) and Dynamic visualization (M=122.49, SD=17.3) conditions; $t(17)$=-0.22, p = 0.83. Likewise no statistically significant difference was found for output between Static Visualization (M=9.83, SD=5.46) and Dynamic visualization (M=11.97, SD=3.92) conditions; $t(17)$=-0.96, p = 0.35.

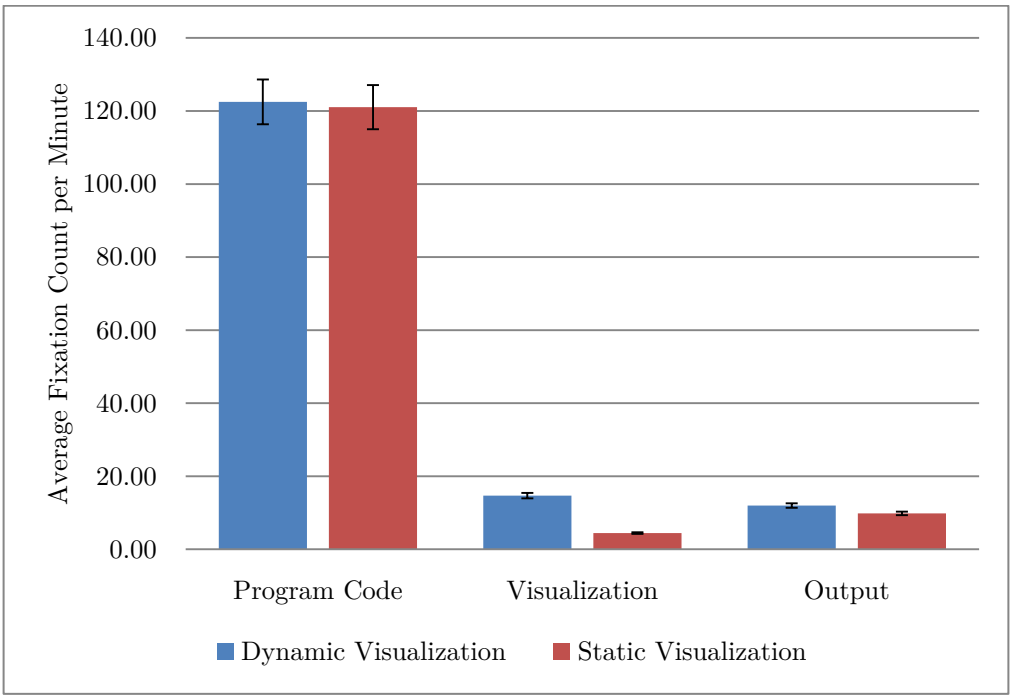


Figure 25. Average Fixation Count per minute

There was a statistically significant difference in the dwell time for visualization between Static Visualization (M=4.45, SD=1.39) and Dynamic visualization (M=14.68, SD=7.75) conditions; $t(17)$=-3.90, p <.05.

A t-test found that there was no statistically significant difference in debugging performance between Static Visualization (M=2.56, SD=1.67) and Dynamic visualization (M=2.80, SD=1.31) conditions; $t(17)$=-.357, p =.726.

Research Question 5: Is any representation (cognitive aid) preferred more than the others?

Data collected from debugging Program 2 was used for analysis here. Of the 11 AOIs possible in this experiment, 8 were of interest here as each of these 8 denoted a different kind of representation. Analysis was performed for representation use with three different visual attributes, Fixation Count, Dwell Time and Visit Count.

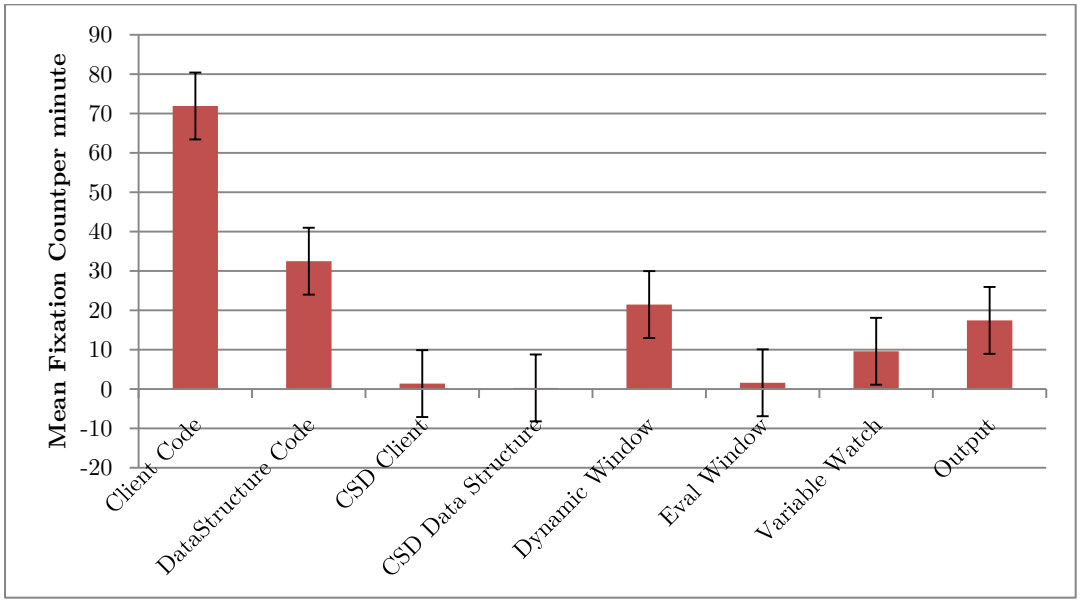


Figure 26. Mean Fixation Counts per Minute

Fixation Count

Mean fixation count per minute for each of the eight representations in use was used for analysis. The one-way ANOVA revealed that the difference in preference of representations was statistically significant across the eight AOI's, $F_{0.05\ (7,\ 144)}= 107.95$, $p<.001$.

Scheffe's post-hoc comparisons of the eight groups indicate that the client code ($M =$ 71.92, SD=11.3, 95% CI [66.45, 77.38]) received significantly higher preference ratings

than the data structure code (M = 32.47, SD=18.1, 95% CI [23.73, 41.22]), followed by dynamic viewer (M = 21.46, SD=14.14, 95% CI [14.65, 28.28]), program output (M = 17.41, SD=9.59, 95% CI [12.79, 22.04]) and the other 4 representations.

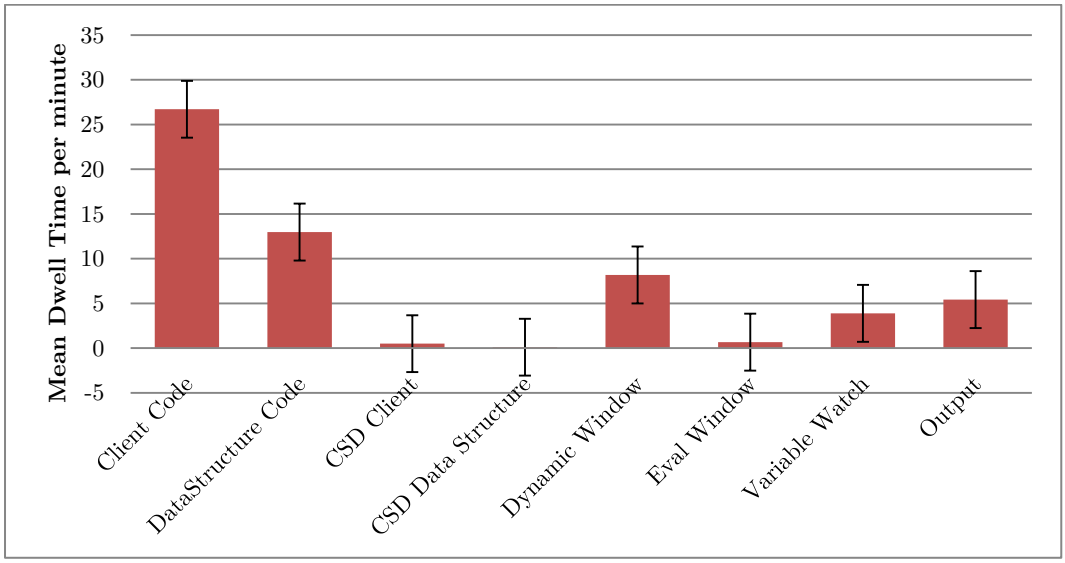


Figure 27. Mean Dwell Time per Minute

Dwell Time

Mean Dwell Time per minute for each of the eight representations in use was used for analysis. One-way ANOVA revealed that the difference in preference of representations was statistically significant across the eight AOI's, $F_{0.05\ (7,\ 144)}$= 114.93, p<.001.

Scheffe's post-hoc comparisons of the eight groups indicate that the client code (M = 44.51, SD=6.34, 95% CI [41.45, 47.56]) received significantly higher preference ratings than the data structure code (M = 21.6, SD=11.19, 95% CI [16.24, 27.02]), followed by dynamic viewer (M = 13.63, SD=8.72, 95% CI [9.42, 17.8]), program output (M = 9.05, SD=4.7, 95% CI [6.78, 11.31]) and the other 4 representations. Pair

wise comparison for data structure code too, was significantly different with all other representations.

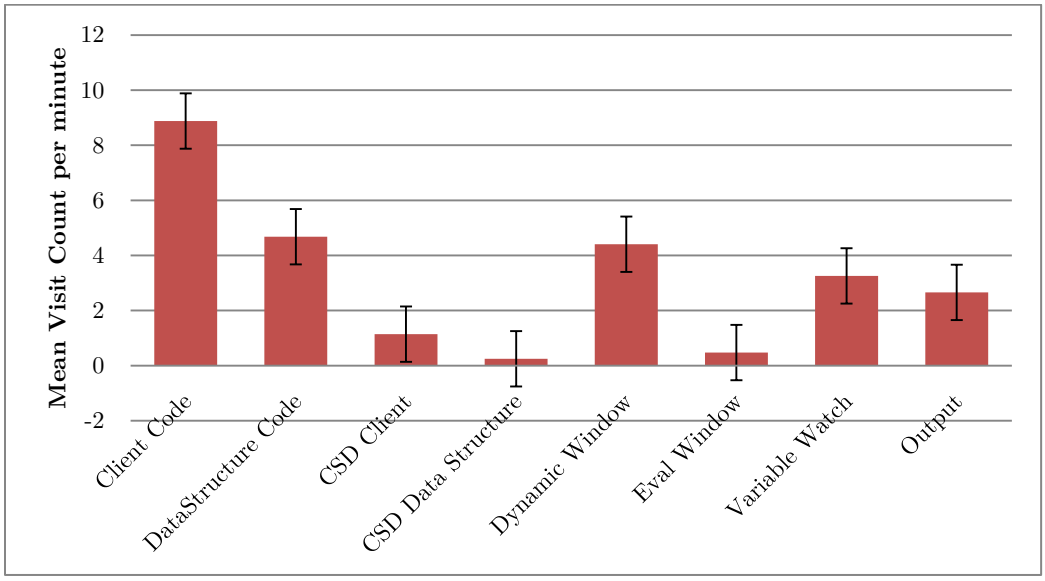


Figure 28. Mean Visit Count per Minute

Visit Count

Mean fixation count per minute for each of the eight representations in use was used for analysis. One-way ANOVA revealed that the difference in preference of representations was statistically significant across the eight AOI's, $F_{0.05\ (7,\ 144)}= 43.55$, $p<.001$.

Scheffe's post-hoc comparisons of the eight groups indicate that the client code ($M =$ 8.8, SD=2.15, 95% CI [7.84, 9.92]) received significantly higher preference ratings than the data structure code ($M = 4.67$, SD=2.56, 95% CI [3.44, 5.91]), followed by dynamic viewer ($M = 4.41$, SD=2.71, 95% CI [3.1, 5.72]), variable watch ($M = 3.26$, SD=1.91, 95% CI [2.33, 4.19]) and the other 4 representations.

In order to simplify the analysis, we combined the AOI's into four main categories (see table 5.4), ignoring the miscellaneous category. We then conducted a one way ANOVA on the four groups and found the difference to be statistically significant for all three visual attributes; fixation count($F_{0.05\ (3,\ 72)}$= 184.1, p<.001), dwell time($F_{0.05\ (3,\ 72)}$= 316.5, p<.001) and visit count($F_{0.05\ (3,\ 72)}$= 69.81, p<.001). Scheffe's post hoc pair wise comparison test too resulted in a statistically significant difference (with p<.05) between all four groups for each of the three visual attributes. The only exception was the difference in Visit Count between Static Visualization and Output with p=0.617.

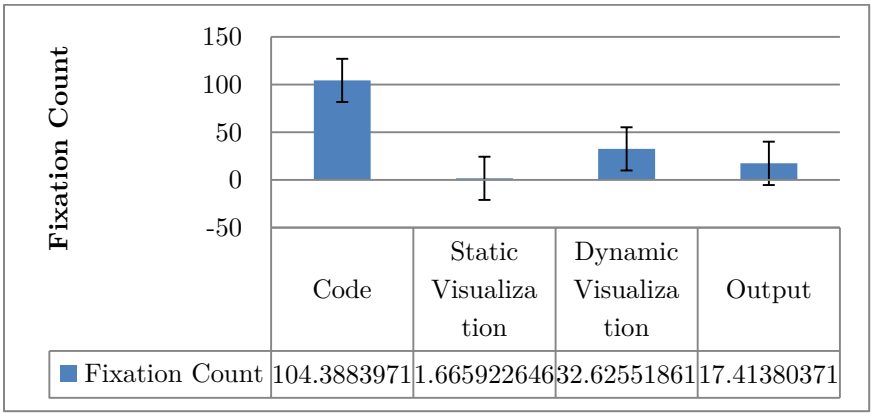


Figure 29. Mean Fixation Count (4 AOI's)

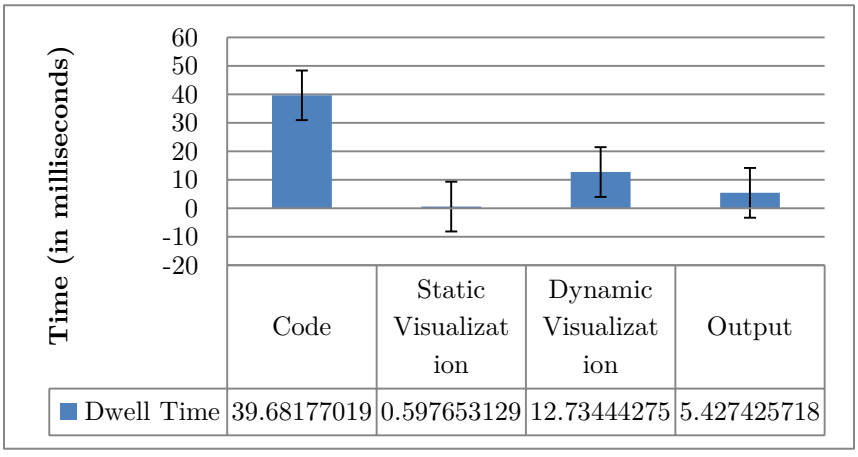


Figure 30. Mean Dwell Time (4 AOI's)

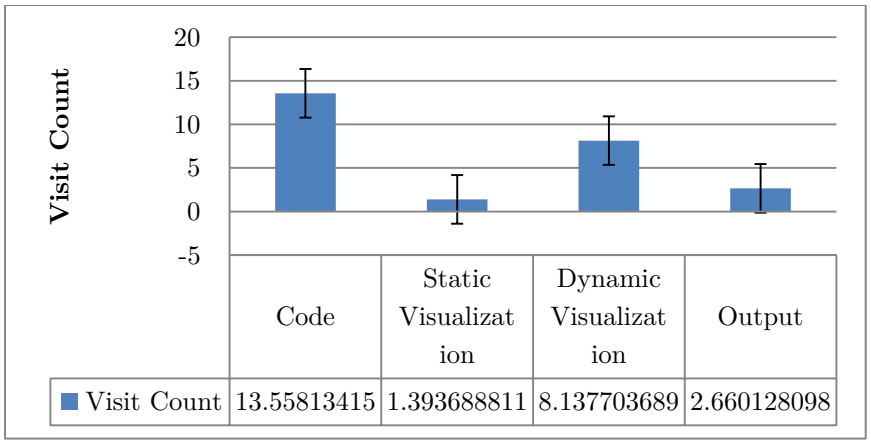


Figure 31. Mean Visit Count (4 AOI's)

Research Question 6: How do programming experience, familiarity with IDE and debugging performance influence the strategies employed in visualization use during debugging?

A utility program was developed to process the raw data collected from our experiments, and based on the desired attributes (such as gaze duration, AOI dimensions etc) the program generated a visual pattern sequence representing attention switches from one AOI to another. Once the visual pattern for all 19 participants was known, Sequential PAttern Mining (SPAM) algorithm (Ayres et al., 2002) was applied on these patterns to mine the frequently occurring visual pattern sequences. This was of importance as there are many different combinations of possible switches between the AOI's. SPAM, developed at Cornell, can be used for finding all frequent sequences within a transactional database. The algorithm is especially efficient when the sequential patterns in the database are very long. A depth-first search strategy is used to generate candidate sequences, and various pruning mechanisms are implemented to reduce the search space. The visual pattern of each participant was converted to a representation resembling a transactional database record. This format was created as a text file and

then processed by SPAM to generate recurring patterns from this data set. As the SPAM algorithm produces only the recurring patterns and not their frequency, the utility program was used to perform a frequency count for these patterns for each participant's pattern. This operation was performed for multiple configurations of AOIs that we were interested in.

In the first approach, the string based visual pattern (discussed in Chapter 5) consisting of 4 AOI's of our interest (Code, Static Visualization, Dynamic Visualization and Output) was generated. Based on the results generated from SPAM and follow up frequency count, the visual patterns listed in Table 7.1 and shown in Figure 32 were prominent. The table lists the visual patterns sorted by their frequency of appearance, with Visual Pattern 1 being the most frequently occurring pattern.

Visual Pattern 1	Visual attention to Code followed by Dynamic Visualization
Visual Pattern 2	Visual attention to Code followed by Output
Visual Pattern 3	Visual attention to Code followed by Static Visualization
Visual Pattern 4	Visual attention to Dynamic Visualization followed by Output
Visual Pattern 5	Visual attention to Code followed by Dynamic Visualization followed by Output
Visual Pattern 6	Visual attention to Static Visualization followed by Dynamic Visualization

Table 7.1 Visual Pattern sorted by frequency of appearance (4 AOI's)

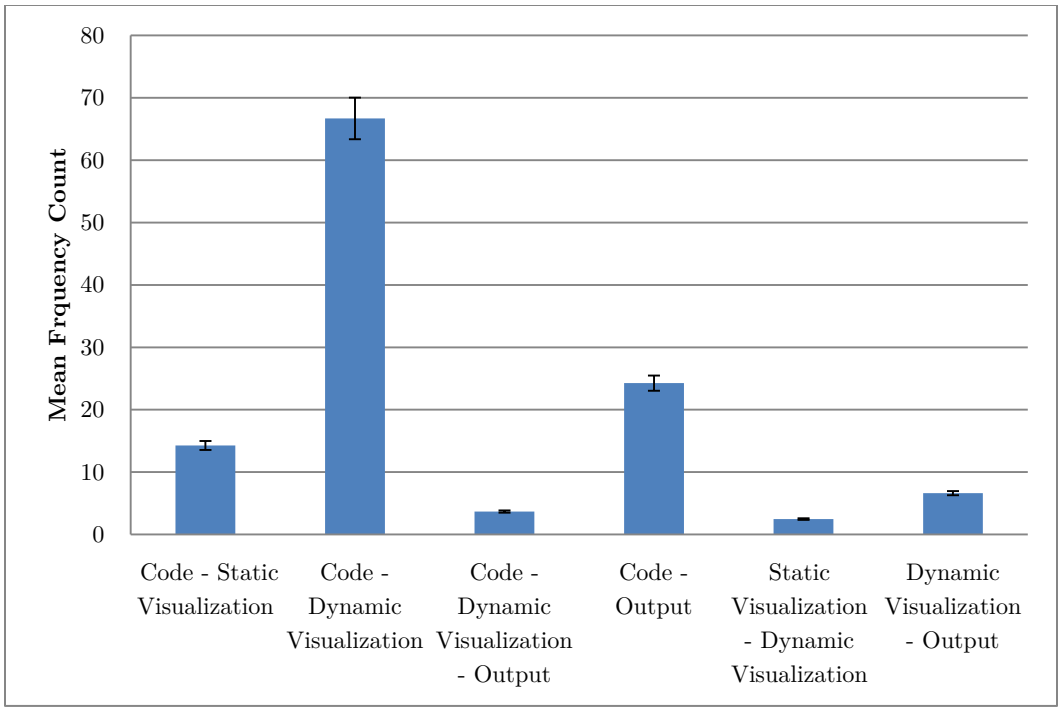


Figure 32. Mean Frequency Count of Visual Patterns (4 AOI's)

We further separated the grouped visualizations into individual AOIs, which lead to 6 AOI's of interest (Code, CSD, Variable Watch, Dynamic Viewer, Evaluation Window and Output). Static visualization was equivalent to just CSD as none of the participants used any other static visualization while debugging program two. The same procedure as the previous step was followed to generate all possible visual patterns and their frequencies. Based on the results, the visual patterns listed in Table 7.2 and illustrated in Figure 33 were prominent. The table lists out the visual patterns sorted by their frequency of appearance, with Visual Pattern 1 being the most frequently occurring pattern.

Visual Pattern 1	Visual attention to Code followed by Variable Watch
Visual Pattern 2	Visual attention to Code followed by Dynamic Window
Visual Pattern 3	Visual attention to Code followed by Output

Visual Pattern 4	Visual attention to Code followed by CSD
Visual Pattern 5	Visual attention to Variable Watch followed by Dynamic Window
Visual Pattern 6	Visual attention to Variable Watch followed by Output
Visual Pattern 7	Visual attention to Code followed by Evaluation Window
Visual Pattern 8	Visual attention to Code followed by Variable Watch followed by Dynamic Window
Visual Pattern 9	Visual attention to Output followed by Dynamic Window

Table 7.2 Visual Pattern sorted by frequency of appearance (6 AOI's)

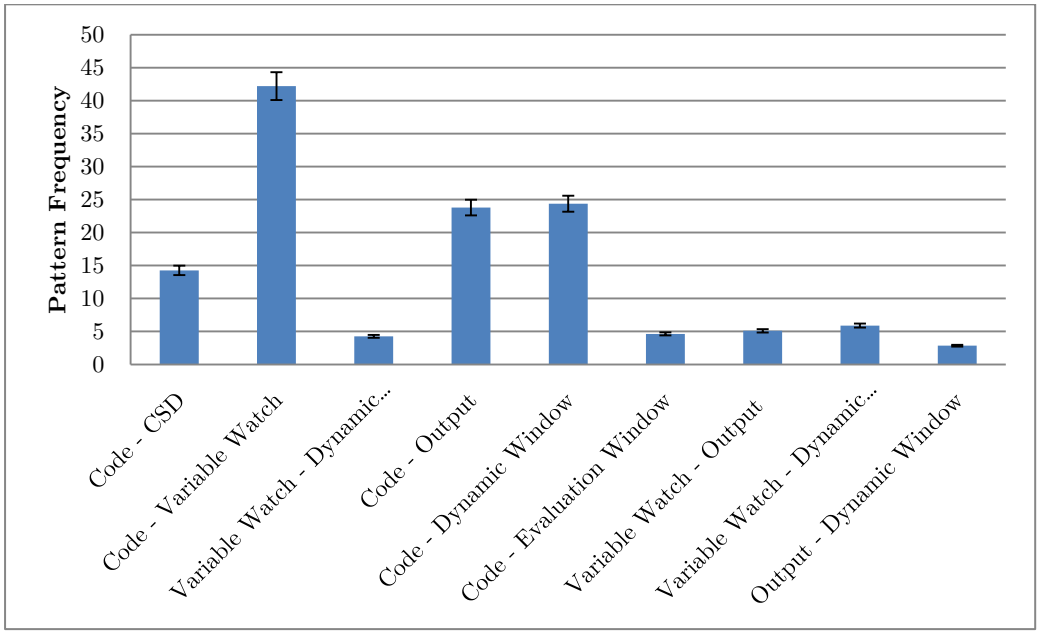


Figure 33. Mean frequency of Visual Patterns (6 AOI's)

Visual Pattern 1	Short Gaze on Code followed by Short Gaze on Static Visualization
Visual Pattern 2	Short Gaze on Code followed by Short Gaze on Static Visualization and then Short Gaze on Dynamic Visualization
Visual Pattern 3	Short Gaze on Code followed by Short Gaze on Dynamic Visualization
Visual Pattern 4	Short Gaze on Code followed by Short Gaze on Dynamic Visualization and then Short Gaze on Output
Visual Pattern 5	Short Gaze on Code followed by Short Gaze on Output

Visual Pattern 6	Long Gaze on Code followed by Short Gaze on Dynamic Visualization
Visual Pattern 7	Long Gaze on Code followed by Short Gaze on Output
Visual Pattern 8	Short Gaze on Static Visualization followed by Short Gaze on Dynamic Visualization
Visual Pattern 9	Short Gaze on Dynamic Visualization followed by Short Gaze on Output

Table 7.3 Visual Pattern sorted by frequency of appearance (8 AOI's)

This was followed by a finer analysis of patterns based on their gaze durations as discussed in Chapter 5. See Table 5.1 for more details. Patterns which emerged from this analysis were sorted by frequency and are listed in Table 7.3.

A frequency count of the patterns was performed and plotted against time. This is summarized in Appendix E 2.5. Outlined in Figure 6.6 is a representation mean frequency of three prominent patterns (Visual Pattern 1, 2 & 3) observed among all the participants over the period of the complete experiment. The vertical axis represents the frequency count of the pattern and the horizontal axis represents the time, with each data point representing a 15 second interval.

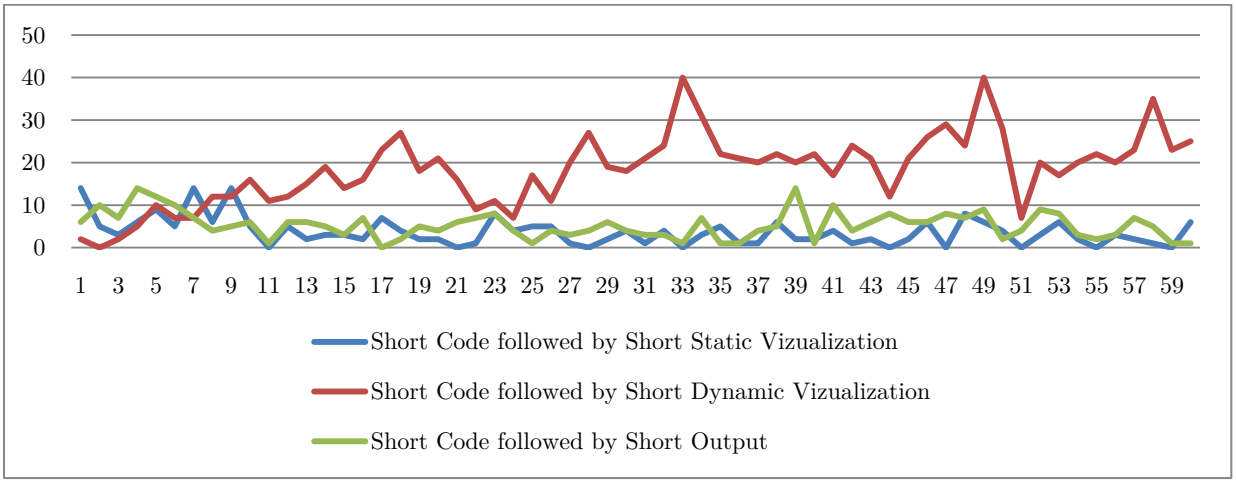


Figure 34. Timeline of 3 prominent visual patterns

In addition to this, we investigated differences in visual strategies of participants based on three Independent Variables, namely programming experience, familiarity with jGRASP and debugging performance.

Based on Programming Experience

	Pattern	Novice Programmer	Expert Programmer
15 Minute Interval	**13**	15.11	10.80
	15	59.67	71.80
	17	22.11	15.10
	25	4.00	6.20
	27	5.33	4.40
	57	7.67	6.20

		Novice Programmer			Expert Programmer		
		Int.1	Int.2	Int.3	Int.1	Int.2	Int.3
5 Minute Interval	**13**	7.33	4.89	3.556	6.3	1.9	2.9
	15	13.6	20	24.11	15.8	26.5	29.5
	157	0.67	1.11	0.667	0.4	0.9	1.1
	17	10.9	6.89	5	5	3.8	7.5
	27	0.78	1.67	0.778	0.9	1	0.8
	57	1.78	4.44	1.556	1	1.7	3.7

		Novice Programmer					Expert Programmer				
		Int.1	Int.2	Int.3	Int.4	Int.5	Int.1	Int.2	Int.3	Int.4	Int.5
3 Minute Interval	**13**	5.11	3.33	2.78	2.22	2.22	5.30	1.60	0.90	1.60	1.70
	15	5.44	10.00	12.22	17.11	12.67	6.40	12.50	19.10	14.00	19.70
	17	7.44	5.78	2.78	4.11	2.67	3.90	2.10	1.90	4.70	3.80
	27	0.22	1.00	0.89	1.00	0.22	0.50	0.50	0.40	1.00	0.20
	57	1.22	1.67	2.11	2.11	0.67	0.40	0.60	0.80	2.40	2.20

Table 7.4 Time based means of pattern frequencies – Programming Experience

The pattern frequencies were calculated for multiple time intervals (5 minute intervals and 3 minute intervals) and evaluated at the end of the experiment. A t-test

was performed to compare the mean frequency of patterns among the two groups. For the complete course of experiment, there was no statistically significant difference for any of the visual patterns. Although the visual pattern Short Code followed by Short Output ($t(17) = 2.02$, $p=.059$) was close to statistical significance, rest of the patterns were either not statistically significant or had small frequency values.

When we looked at patterns in 5 minute intervals, there was a statistically significant difference for the visual pattern Short Code followed by Short Output ($t(17) = 2.59$, $p<.05$) during the first interval. The difference for rest of the patterns were either not statistically significant different or had small frequency values. On analyzing the 10 minute intervals, the same visual pattern Short Code followed by Short Output showed statistically significant difference for both interval one ($t(17) = 2.254$, $p<.05$) and interval two ($t(17) = 2.795$, $p<.05$). Rest of the patterns were either not statistically significantly different or had small frequency values. A frequency count of the patterns for each group was performed and plotted against time. Figure 6.6 is a representation of mean frequency of three prominent patterns (Visual Pattern 1, 2 & 3) observed among all the participants over the period of the complete experiment. The vertical axis represents the average frequency count of the pattern and the horizontal axis represents the time, with each data point representing a 15 second interval.

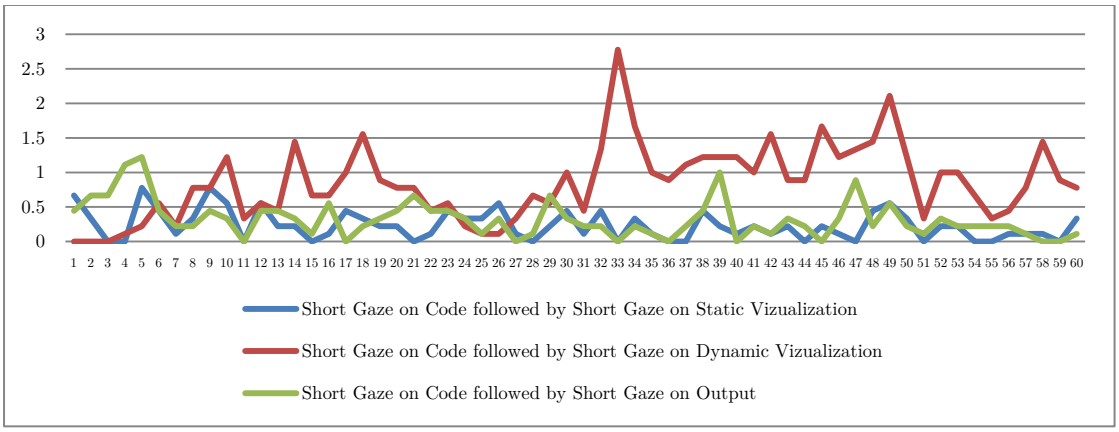


Figure 35. Timeline of 3 prominent visual patterns - Novice Programmers

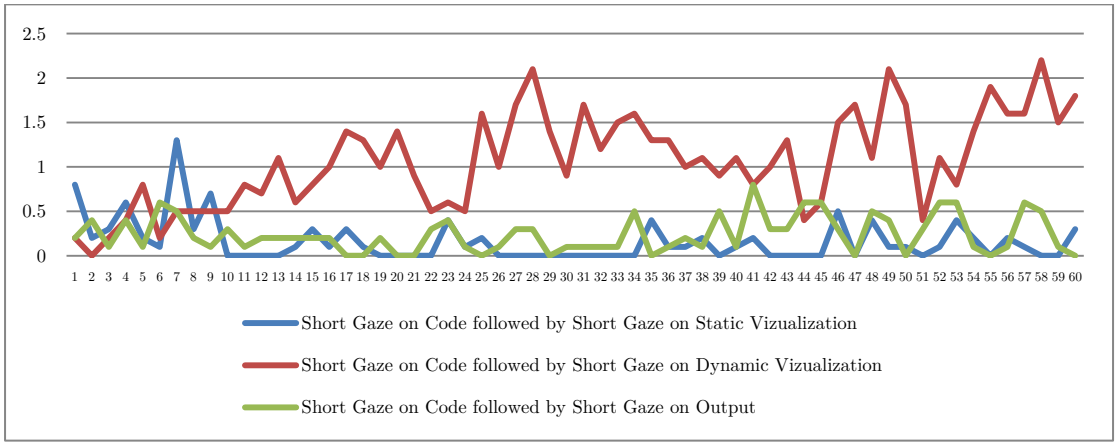


Figure 36. Timeline of 3 prominent visual patterns - Experienced Programmers

Based on familiarity with jGRASP

For the complete duration of the experiment, a statistically significant difference was found for the visual pattern Short Code followed by Short Output ($t(17) = -3.24$, $p<.05$). Statistically significant difference was also found for the visual pattern Short Code followed by Short Dynamic Visualization ($t(17) = 2.88$, $p<.05$).

When broken down to 5 minute intervals, there was statistically significant difference for all 3 intervals for the visual pattern Short Code followed by Short Output. Rest of

the patterns were either not statistically significant or had small frequency values. A frequency count of the patterns for each group was performed and plotted against time. Figure 6.6 is a representation of mean frequency of three prominent patterns (Visual Pattern 1, 2 & 3) observed among all the participants over the period of the complete experiment. The vertical axis represents the average frequency count of the pattern and the horizontal axis represents the time, with each data point representing a 15 second interval.

	Pattern	Little or No jGRASP Experience					Experienced jGRASP user				
15 Minute Interval	**13**					5.2	15.6				
	15					96.4	55.2				
	17					10.2	21.4				
	25					4.2	5.5				
	27					4.4	5.0				
	57					4.4	7.8				
		Little or No jGRASP Experience					Experienced jGRASP user				
				Int.1	Int.2	Int.3	Int.1	Int.2	Int.3		
5 Minute Interval	**13**			1.7	3.7	1.7	7.8	3.3	3.7		
	15			8.3	37.7	51.7	15.9	20.8	23.8		
	17			9.7	4.7	2.7	7.4	5.4	7.5		
	25			0.7	0.3	2.0	0.5	0.8	0.9		
	27			1.3	0.3	0.7	0.8	1.5	0.9		
	57			0.3	2.0	1.7	1.6	3.2	3.1		
		Little or No jGRASP Experience					Experienced jGRASP user				
		Int.1	Int.2	Int.3	Int.4	Int.5	Int.1	Int.2	Int.3	Int.4	Int.5
3 Minute Interval	**13**	1.33	1.7	2.0	1.0	1.0	5.94	2.6	1.9	2.2	2.4
	15	2.67	8.3	31.0	22.3	33.3	6.56	11.9	13.9	15.1	15.1
	17	5.67	6.0	1.0	3.0	1.3	5.56	3.4	2.7	5.0	4.1
	27	0.67	0.0	0.3	0.3	1.7	0.06	0.6	0.5	0.5	0.7
	57	0.33	0.0	1.3	0.7	1.7	0.88	1.3	1.5	2.7	1.6

Table 7.5 Time based means of pattern frequencies – Experience with jGRASP

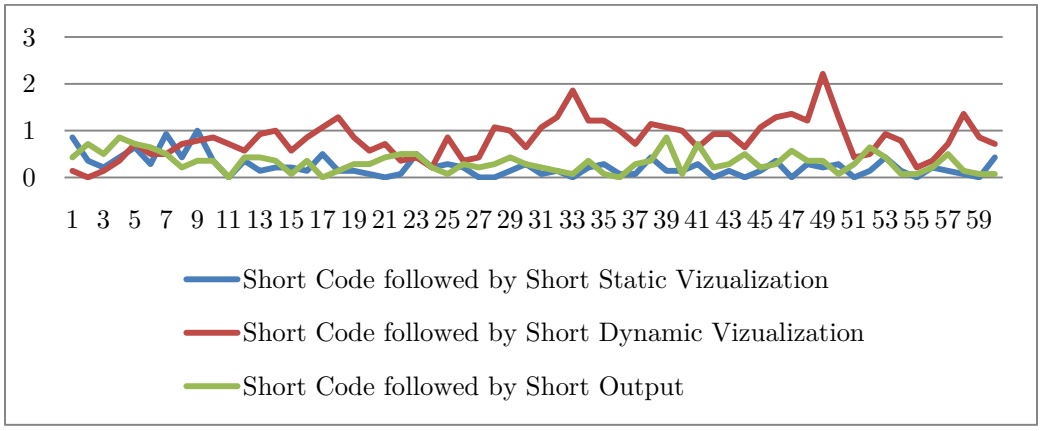


Figure 37. Timeline of 3 prominent visual patterns - Experience with jGRASP

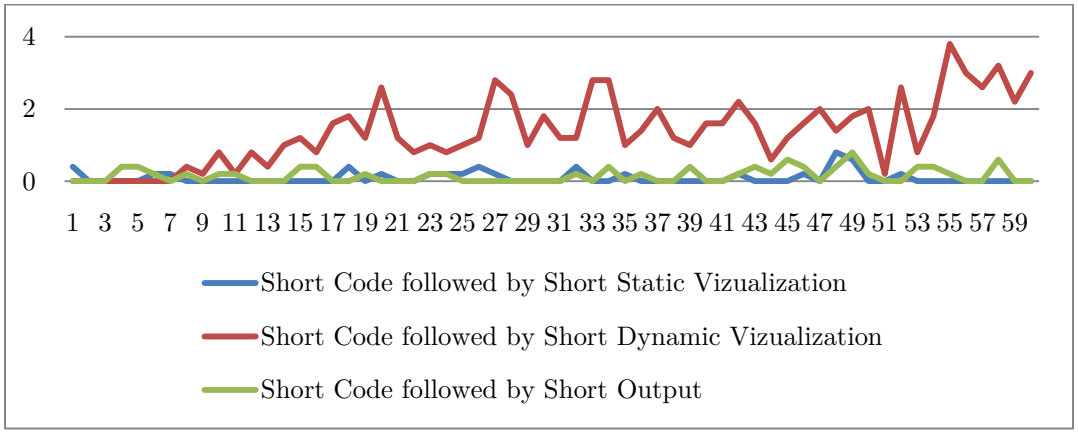


Figure 38. Timeline of 3 prominent visual patterns - No or minimal experience with jGRASP

Based on Debugging Performance

There was no statistically significant difference in the patterns between the two groups for the complete duration of the experiment. When broken down to 5 minute intervals, there was a statistically significant difference for the visual pattern Short Code followed by Short Dynamic Visualization at interval three ($t(16) = 3.14$, $p<.05$).

It was close to statistical significance at interval two ($t(17) = 1.96$, p=.066). Rest of the patterns were either not statistically significant or had small frequency values. Likewise when broken down to 3 minute intervals, there was a statistically significant difference for the same visual pattern at interval three ($t(16) = 3.12$, p<.05) and interval five ($t(15) = 2.73$, p<.05).

	Pattern	Poor Performance	Good Performance
15 Minute Interval	13	7.3	13.9
	15	99.3	59.8
	17	16.0	18.9
	25	7.7	4.7
	27	4.0	5.0
	57	4.0	7.4

		Poor Performance			Good Performance		
		Int.1	Int.2	Int.3	Int.1	Int.2	Int.3
5 Minute Interval	13	1.7	3.7	1.7	7.8	3.3	3.7
	15	8.3	37.7	51.7	15.9	20.8	23.8
	17	9.7	4.7	2.7	7.4	5.4	7.5
	25	0.7	0.3	2.0	0.5	0.8	0.9
	27	1.3	0.3	0.7	0.8	1.5	0.9
	57	0.3	2.0	1.7	1.6	3.2	3.1

		Poor Performance					Good Performance				
		Int.1	Int.2	Int.3	Int.4	Int.5	Int.1	Int.2	Int.3	Int.4	Int.5
3 Minute Interval	13	1.33	1.7	2.0	1.0	1.0	5.94	2.6	1.9	2.2	2.4
	15	2.67	8.3	31.0	22.3	33.3	6.56	11.9	13.9	15.1	15.1
	17	5.67	6.0	1.0	3.0	1.3	5.56	3.4	2.7	5.0	4.1
	25	0.67	0.0	0.3	0.3	1.7	0.06	0.6	0.5	0.5	0.7
	57	0.33	0.0	1.3	0.7	1.7	0.88	1.3	1.5	2.7	1.6

Table 7.6 Time based means of pattern frequencies – Based on performance

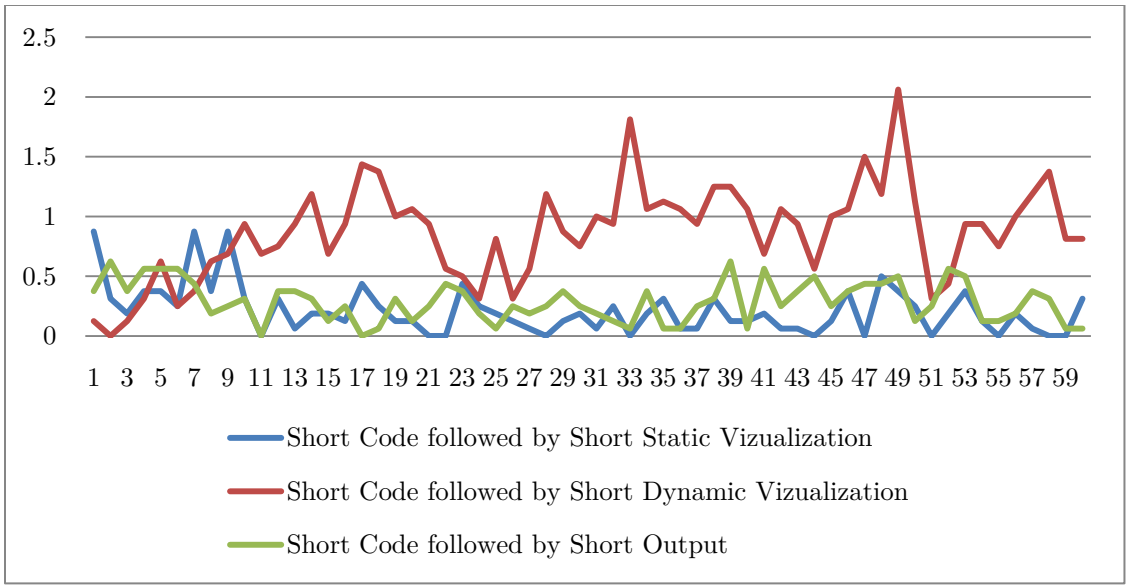


Figure 39. Timeline of 3 prominent visual patterns - Poor Performance

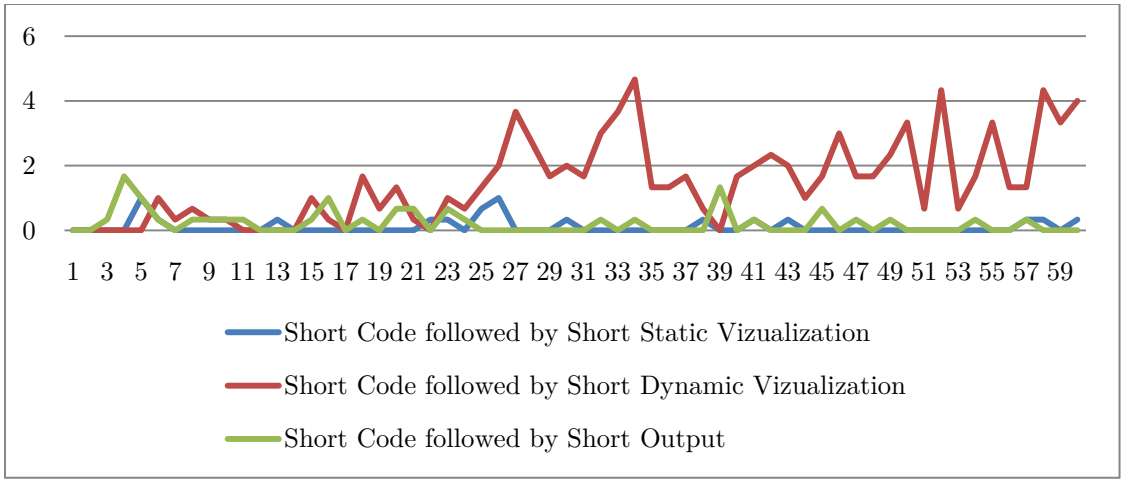


Figure 40. Timeline of 3 prominent visual patterns - Better Performance

A frequency count of the patterns for each group was performed and plotted against time. Figure 6.6 is a representation of the mean frequency of three prominent patterns (Visual Pattern 1, 2 & 3) observed among all the participants over the period of the complete experiment. The vertical axis represents the average frequency count of the pattern and the horizontal axis represents the time, with each data point representing a 15 second interval.

Chapter 8

CONCLUSIONS

The goal of experiment one (to debug program one) was to compare the differences in the usage pattern of static vs. dynamic visualizations, and to evaluate the differences in construction of mental model representation of the program between these two visualization groups. Students were provided codes with multiple logical errors and their task consisted of locating and correcting the errors using the jGRASP IDE. It was observed that the group using dynamic visualizations found more bugs on average, but the difference was not statistically significant when compared to the static visualization group. Perhaps the smaller number of subjects may have led to this result as similar studies (Cross et. al. 2009) in the past have found significant performance difference with much larger number of subjects. The final strength of the mental model for the static group was weaker than the dynamic group. Although not a statistically significant difference, this provides evidence in support of our postulate in the proposed cognitive model that the mental model created by static visualizations is not as extensive as the one created with support of dynamic visualization. There was also a strong correlation between the mental model strength and debugging performance for the dynamic visualization group, which was statistically significant when compared to static group that showed a weaker correlation and was not statistically significant. The static group used minimal visualizations and primarily depended on the program code and output, whereas

the dynamic group used a significantly higher proportion of dynamic visualization, along with program output and program code.

On further analysis pertaining to five programming constructs of program one, we found that the mental strength for Function was stronger for the group with access to dynamic visualizations when compared to static visualizations. The mean difference was not statistically significant and perhaps ceiling effect was a cause. Dynamic Data structure too was consistently higher for the dynamic visualization group at all three stages. On the contrary, strength of mental model for both control flow and data flow was consistently stronger for the static group. The comparative growth of program structure was uneven between the two groups.

The second experiment (debugging program two) gave us new insights into the visual strategies of programmers. The representations on the IDE received different dwell times and fixation counts. Consistent with previous studies (Bednarik, et. al., 2006 and Romero, et. al., 2002a), it was found that source code received the highest attention followed by dynamic viewer, output and variable watch. This order was slightly different with respect to visit counts, with variable watch receiving more visits compared to the output window. In terms of visual patterns, we found that the most common pattern was the switch between code and variable watch window, followed by code to dynamic viewer, code to static output and code to CSD.

Novice programmers tended to look at the program output more frequently than expert programmers. It was also observed that novice programmers used static visualizations more often in the first 6-9 minutes of their task. Participants with lesser experience with the jGRASP IDE looked more often at the dynamic visualizations and less at the program output when compared to participants with more experience in using jGRASP. Participants who performed better in debugging activity did not perform

frequent switches with the dynamic visualizations towards the third part of the session when compared to poor performers. Poor performers switched more between the dynamic visualizations and code when compared to participants with good performance. It is evident here that more usage of dynamic representations did not essentially lead to better performance. And, excessive usage could actually have been detrimental to the performance. However, based on the analysis performed it is not clear as to what could be an optimal usage threshold of dynamic representations for better performance at debugging.

This thesis hints at several directions that future research might take. The proposed cognitive model poses some intriguing questions, like how is the posit mental model compared to the dynamic mental model. Or, how does the strength of each individual mental model construct influence debugging performance. Some of the results arising out of this research have direct implication on the design of IDE interfaces. IDE interfaces should be designed with usability as a prime goal. In the current design of IDE's for novice programmers, all the debugging tools are not readily visible. Moreover, utility and functionality of each component is not evident at the interface. The IDE should be able to make such functionalities readily available and encourage students to employ them during program comprehension. IDEs should promote static tools in the beginning of a programming session to prime structure based knowledge, followed by promotion of dynamic visualizations in later phases to promote better comprehension. There is also a potential for designing an intelligent tutoring system for novices that works in conjunction with a gaze-tracking IDE, which could assist students during program debugging or comprehension by proposing intelligent suggestions on visualization use. These are avenues for future research.

Cumulative Bibliography

1. Aaltonen, A., Hyrskykari, A., & Räihä, K.-J. (1998). 101 spots, or how do users read menus? *ACM conference on Human factors in computing systems*, (pp. 132-13). New York, NY, USA.

2. Ainsworth, S. E., Wood, D. J., & O'Malley, C. (1998b). There's more than one way to solve a problem: Evaluating a learning environment to support the development of children's multiplication skills. *Learning and Instruction , 8* (2), 141-157.

3. Barreto, A., Gao, Y., & Adjouadi, M. (2008). Pupil diameter measurements: untapped potential to enhance computer interaction for eye tracker users? *ACM SIGACCESS conference on Computers and accessibility*, (pp. 269-270). New York, NY, USA.

4. Bednarik, R., & Tukiainen, M. (2006). An eye-tracking methodology for characterizing Program Comprehension processes. *2006 symposium on Eye tracking research & applications*, (pp. 125-132). New York, NY, USA.

5. Bednarik, R., & Tukiainen, M. (2007b). Validating the restricted focus viewer: A study using eye-movement tracking. *Behavior Research Methods , 39* (2), Behavior Research Methods.

6. Bednarik, R., Myller, N., Sutinen, E., & Tukiainen, M. (2006). Analyzing Individual Differences in Program Comprehension. *Technology, Instruction, Cognition and Learning , 3* (3-4), 205-232.

7. Bednarik, R., Myller, N., Sutinen, E., & Tukiainen, M. (2005). Effects of experience on gaze behaviour during program animation. *17th Annual Psychology of Programming Interest Group Workshop*, (pp. 49-61). Brighton, UK.

8. Biej, H.-J. (2009). Gaze-augmented manual interaction. *ACM conference on Human Factors in Computing Systems*, (pp. 3121-3124).

9. Blackwell, A., Jansen, A., & Marriott, K. (2000). Restricted Focus Viewer: A Tool for Tracking Visual Attention. In M. Anderson, P. Cheng, & V. Haarslev, *Theory and Application of Diagrams* (pp. 575-588).

10. Brooks, R. (1983). Towards a Theory of the Comprehension of Computer Programs. *International Journal Man-Machine Studies , 18*, 543-554.

11. Busemeyer, J. R., & Diederich, A. (2010). *Cognitive modeling.* Los Angeles: Sage.

12. Carney, R., & Levin, J. (2002). Pictorial Illustrations Still Improve Students' Learning from Text. *Educational Psychology Review , 14* (1), 5-26.

13. Cheng, P.-H., Lowe, R. K., & Scaife, M. (2001). Cognitive Science Approaches to Understanding Diagrammatic Representations. *Artificial Intelligence Rev. , 15*, 79-94.

14. Cox, R., & Brna, P. (1995). Analytical reasoning with external representations: Supporting the stages of selection, construction and use. *Journal of Artificail Intelligence in Education , 6* (2/3), 239-302.

15. Crane, H. D. (1994). The Purkinje image eyetracker, image stabilization, and related forms of stimulus manipulation. *Visual science and engineering: Models and applications* , 15-89.

16. Cross, J. H., Hendrix, D., Umphress, D., Barowski, L., Jain, J., & Montgomery, L. (2009). Robust Generation of Dynamic Data Structure Visualizations with Multiple Interaction Approaches. *ACM Transactions on Computing Education , 9* (2).

17. Cutrell, E., & Guan, Z. (2007). What are you looking for?: an eye-tracking study of information usage in web search. *ACM conference on Human factors in computing systems*, (pp. 407-416). New York, NY, USA.

18. Ducassé, M., & Emde, A. -M. (1988). A review of automated debugging systems: Knowledge, strategies and techniques. *International Conference of Software Engineering*, (pp. 162–171). Singapore.

19. Duchowski, A. (2007). *Eye tracking methodology: Theory and practice* (Second ed.). Springer.

20. Gentner, D. (1989). The mechanisms of analogical learning. In S. Vosniadou, & A. Ortony, *Similarity and Analogical Reasoning* (pp. 197-241). Cambridge: Cambridge University Press, England.

21. Gernsbacher, M. A., Varner, K. R., & Faust, M. (1990). Investigating differences in general comprehension skill. *Journal of Experimental Psychology: Learning, Memory, and Cognition* (16), 430-445.

22. Gilmore, D. J. (1991). Models of debugging. *Acta Psychologica , 78* (1-3), 151-172.

23. Goldberg, H. J., & Kotval, X. P. (1999). Computer interface evaluation using eye movements: Methods and constructs. *International Journal of Industrial Ergonomics , 24*, 631-645.

24. Goldberg, J. H., & Kotval, X. P. (1998). Eye-movement based evaluation of the computer interface. *Advances in Occupational Ergonomics and Safety* , 529-532.

25. Gould, J. (1975). Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies , 7* (1), 151-182.

26. Graesser, A. C., Millis, K. K., & Zwaan, R. A. (1997). Discourse comprehension. *Annu. Rev.Psychol.* (48), 163–189.

27. Green, T. R. (1989). Cognitive dimensions of notations. *People and Computers* .

28. Grubb, P., & Takang, A. (2003). *Software Maintenance: Concepts and Practice.* Singapore: World Scientific Publishing.

29. Jacob, R. J., & Karn, K. S. (2003). Eye tracking in Human-Computer Interaction and usability research: Ready to deliver the promises. *The mind's eye: Cognitive and applied aspects of eye movement research* , 573-605.

30. Johnson-Laird, P. N. (1983). *Mental Models: towards a cognitive science of language, inferences and consciousness.* Cambridge: Cambridge University Press.

31. Just, M. A., & Carpenter, P. A. (1992). A capacity theory of comprehension:. *Psychological Review* (98), 122–149.

32. Katz, I., & Anderson, J. (1987). Debugging: An analysis of bug location strategies. *Human- Computer Interaction , 3* (4), 351–399.

33. Kintsch, W. (1998). *Comprehension: A paradigm for cognition.* Cambridge: Cambridge University Press, UK.

34. Kintsch, W. (1988). The Role of Knowledge in Discourse Comprehension: A Construction-Integration Model. *Psychological Review* (95), 163-182.

35. Kintsch, W., & Van Dijk, T. A. (1975). Comment on se rappelle et on resume des histoires,. *In Langage* (40), 98-116.

36. Kumar, M., Paepcke, A., & Winograd, T. (2007). EyePoint: practical pointing and selection using gaze and keyboard. *ACM conference on Human factors in computing systems*, (pp. 421-430). New York, NY, USA.

37. Letovsky, S. (1986). Cognitive Processes in Program Comprehension. *Proc. First Workshop Empirical Studies of Programmers* , 58-79.

38. Levie, H. W., & Lentz, R. (1982). Effects of text illustrations: A review of research. *Educ.Commun. Technol. J* (30), 195–232.

39. Lewandowsky, S., & Behrens, J. T. (1999). Statistical graphs and maps. *Handbook of applied cognition* , 513-549.

40. Littman, D., Pinto, J., Letovsky, S., & Soloway, E. (1987). Mental models and software maintenance. *Jouranal of Systems and Software , 7* (4), 341-355.

41. Loman, N. L., & Mayer, R. E. (1983). Signaling techniques that increase the understandability of expository prose. *Journal of Educational Psychology , 75*, 402-412.

42. M., C., & J., S. (1989). Subject Differences in the Reading of Computer Algorithms. *Designing and Using Human-Computer Interfaces and Knowledge-Based Systems* , 137-144.

43. Mautone, P. D., & Mayer, R. E. (2007). Cognitive aids for guiding graph comprehension. *Journal of Educational Psychology , 99* (3), 640-652.

44. Mayer, R. E. (1996). Learning strategies for making sense out of expository text: The SOI model for guiding three cognitive processes in knowledge construction. *Educational Psychology Review* (8), 357-371.

45. Mayer, R. E. (2001). *Multi-media Learning.* Cambridge, UK: Cambridge University Press.

46. Mayer, R. E. (1997). Multimedia learning: Are we asking the right questions? *Education Psychology* (32), 1-19.

47. Mayer, R. E. (1983). The elusive search for teachable aspects of problem solving. (J. A. Glover, & R. R. Ronning, Eds.) *Historical foundations of educational psychology* , 327–348.

48. Mayer, R. E., & Moreno, R. (2003). Nine ways to reduce cognitive load in multimedia learning. *Educational Psychologist* (38), 43–52.

49. McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., et al. (2008). Debugging: A review of the literature from an educational perspective. *Computer Science Education , 18* (2), 67-92.

50. Narayanan, H., & Hegarty, M. (1998). On designing comprehensible interactive hypermedia manuals. *International Journal of Human Computer Studies , 48* (2).

51. Narayanan, N. H., & Hegarty, M. (2002). Multimedia design for communication of dynamic information. *International Journal of Human Computer Studies. , 57*, 279-315.

52. Nathan, M. J., Kintsch, W., & Young, E. (1992). A Theory of Algebra-Word-Problem Comprehension and Its Implications for the Design of Learning Environments. *Cognition & Instruction , 9* (4), 329.

53. Nevalainen, S., & Sajaniemi, J. (2004). Comparison of three eye tracking devices in psychology of programming research. *6th Annual Psychology of Programming Interest Group*, (pp. 170-184). Carlow, Ireland.

54. Nevalainen, S., & Sajaniemi, J. (2005). Short-term effects of graphical versus textual visualisation of variables on program perception. *17th Annual Psychology of Programming Interest Group Worskhop*, (pp. 77-91).

55. Oberlander, J., Stenning, K., & Cox, R. (1999). Hyperproof: Abstraction, Visual preference and Modality. *Logic, Language and Computation , II*, 222-236.

56. Pennington, N. (1987a). Comprehension strategies in programming. *Comprehension strategies in programming* , 100-113.

57. Pennington, N. (1987b). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology , 19*, 295-341.

58. Rayner, K. (1998). Eye movements in reading and information processing: 20 years of research. *Psychological Bulletin , 124*, 372-422.

59. Rickards, J. P., Fajen, B. R., Sullivan, J. F., & Gillespie, G. (1997). Signaling, notetaking, and field independence-dependence in text comprehension and recall. *Journal of Educational Psychology , 89* (3), 508-517.

60. Romero, P., Cox, R., du Boulay, B., & Lutz, R. (2003a). A survey of representations employed in object-oriented programming environments. *Journal of Visual Languages and Computing , 14* (5), 387-419.

61. Romero, P., Cox, R., du Boulay, B., & Lutz, R. (2002a). Visual attention and representation switching during Java program debugging: a study using the Restricted Focus Viewer. *Diagrammatic Representation and Inference : Second International Conference, Diagrams* (pp. 221-235). Callaway Gardens, GA, USA: Springer Verlag.

62. Romero, P., du Boulay, B., Cox, R., & Lutz, R. (2003b). Java debugging strategies in multirepresentational environments. *15th Annual Workshop of the Psychology of Programming Interest Group (PPIG).* Keele University, UK.

63. Romero, P., Lutz, R., Cox, R., & Du Boulay, B. (2002b). Co-ordination of multiple external representations during Java program debugging. *Empirical Studies of Programmers symposium of the IEEE Human Centric Computing Languages and Environments Symposia.* Arlington, VA.

64. Schnotz, W., & Bannert, M. (2003). Construction and interference in learning from multiple representation. *Learning and Instruction.* (13), 141–156.

65. Schnotz, W., & Bannert, M. (1999). Support and interference effects in learning from multiple representations. *European Conference on Cognitive Science* , 447-452.

66. Shah, P., & Carpenter, P. (1995). Conceptual limitations in comprehending line graphs. *Journal of Experimental Psychology , 124*, 337-370.

67. Shah, P., & Hoeffner, J. (2002). Review of graph comprehension research: Implications for instruction. *Educational Psychology Review , 14* (1), 47-69.

68. Shah, P., Mayer, R. E., & Hegarty, M. (1999). Graphs as aids to knowledge construction. *Journal of Educational Psychology , 91*, 690-702.

69. Shneiderman, B., & Mayer, R. (1979). Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer and Information Sciences , 8* (3), 219-238.

70. Sime, J. (1996). An investigation into teaching and assesment of qualitative knowledge in engineering. *European Conference on Artificial Intelligence on Education* , 240-246.

71. Soloway, E., Adelson, B., & Ehrlich, K. (1988). Knowledge and Processes in the Comprehension of Computer Programs. *The Nature of Expertise* , 129-152.

72. Soloway, E., Lampert, R., Letovsky, S., Littman, D., & Pinto, J. (1988, November 31). Designing documentation to compensate for delocalized plans. *Communications ACM* , pp. 1259-1267.

73. Stasko, J. T., Domingue, J. B., Brown, M. H., & Price, B. A. (1998). *Software Visualization.* MIT Press.

74. Steptoe, W., Wolff, R., Murgia, A., Guimaraes, E., Rae, J., Sharkey, P., et al. (2008). Eye-tracking for avatar eye-gaze and interactional analysis in immersive collaborative virtual environments. *ACM conference on Computer supported cooperative work*, (pp. 197-200). New York, NY, USA.

75. Storey, M.-A. D., Fracchia, F. D., & Müller, H. A. (1999). Cognitive design elements to support the construction of a mental model during software exploration. *Journal of System Software , 44* (3), 171-185.

76. Trabasso, T., & Van den Broek, P. (1985). Causal Thinking and the Representation of Narrative Events. *Journal of Memory and Language* (24), 612-630.

77. Uwano, H., Nakamura, M., Monden, A., & Matsumoto, K. (2006). Analyzing individual performance of source code review. *Eye tracking research & applications*, (pp. 133-140). San Diego, California.

78. Uwano, H., Nakamura, M., Monden, A., & Matsumoto, K. (2006). Analyzing individual performance of source code review using reviewers' eye movement. *2006 symposium on Eye tracking research & applications*, (pp. 133–140). New York, NY, USA.

79. Van Oostendorp, H., & Goldman, S. R. (1998). *The Construction of Mental Representations During Reading.* Mahwah, N.J.: L. Erlbaum Associates.

80. Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man–Machine Studies , 23*, 459–494.

81. Xu, S., Jiang, H., & Lau, F. (2009). User-oriented document summarization through vision-based eye-tracking. *ACM conference on Intelligent user interfaces*, (pp. 7-16). New York, NY, USA.

82. Young, L. R., & Sheena, D. (1975). Survey of eye movement recording methods. *Behavior Research Methods* , 397-429.

Appendix A

DEMOGRAPHIC DATA

	Gender	Level	Programming Experience	Java Experience	Prior experience with jGRASP	Experience with jGRASP	Group
P01	Male	Undergraduate	1 - 2 years	6 - 12 months	Yes	6 - 12 months	SV
P02	Male	Undergraduate	1 - 2 years	6 - 12 months	Yes	6 - 12 months	SV
P03	Male	Undergraduate	2 - 5 years	1 - 2 years	Yes	1 - 2 years	SV
P04	Female	Undergraduate	6 -12 months	6 - 12 months	Yes	6 - 12 months	SV
P05	Male	Undergraduate	1 - 2 years	6 - 12 months	Yes	6 - 12 months	SV
P06	Male	Undergraduate	6 - 12 months	6 - 12 months	Yes	6 - 12 months	SV
P07	Male	Undergraduate	5 + years	2 - 5 years	Yes	2 + years	SV
P08	Male	Undergraduate	2 - 5 years	2 - 5 years	No	NA	DV
P09	Male	Undergraduate	5 + years	2 - 5 years	No	NA	SV
P10	Male	Graduate	5 + years	5 + years	Yes	0 - 6 months	DV
P11	Male	Undergraduate	6 - 12 months	6 - 12 months	Yes	6 - 12 months	DV
P12	Male	Undergraduate	2 - 5 years	2 - 5 years	Yes	2 + years	DV
P13	Female	Graduate	5 + years	2 - 5 years	No	NA	DV
P14	Male	Undergraduate	2 - 5 years	6 - 12 months	Yes	6 - 12 months	DV
P15	Male	Undergraduate	1 - 2 years	1 - 2 years	Yes	1 - 2 years	DV
P16	Male	Graduate	1 - 2 years	1 - 2 years	Yes	1 - 2 years	DV
P17	Male	Graduate	5 + years	2 - 5 years	No	NA	DV
P18	Male	Undergraduate	1 - 2 years	6 - 12 months	Yes	6 - 12 months	DV
P19	Male	Undergraduate	1 - 2 years	6 - 12 months	Yes	6 - 12 months	SV

Appendix B

EXPERIMENT PROGRAMS

B1. EXPERIMENT 1

This Program consisted of 3 Java Classes and an Interface. The total lines of code in this programwas 134.

Client Class

```
public class DebuggingAssignmentClient1 {

       public static void main(String[] args) {

         String input = new String("a.b.c.d.e.f");

         DataStructure<String> s1 = new DataStructure<String>();

         for (int i=1; i<input.length() ; i++) {
            if(i%2 == 0 && i%5 != 0) {
               s1.insert(input.substring(i,i+1) +" ");
            }
            else {
               s1.insert(" ");
            }
         }

         System.out.println("Expected Output : f e d c b a");
         System.out.print("Actual Output   : ");
         while (!s1.isEmpty()) {
            System.out.print(s1.remove());
         }
      }
   }
```

DSFrame Interface

```
    public interface DSFrame<T> {

       public void insert(T item);
```

```
        public T remove();
        public boolean isEmpty();
        public int size();
        public String toString();

    }
```

DataStructure Class

```
public class DataStructure<T> implements DSFrame<T>{

      private int n;
      private Unit first;

       public DataStructure() {
         first = null;
         n = 0;
      }
       @SuppressWarnings("unchecked")

       public void insert(T item) {

         Unit oldfirst = first;
         first = new Unit();
         first.setValue(item);
         first.setNext(oldfirst);
         n++;
      }

       @SuppressWarnings("unchecked")

       public T remove() {

         if (isEmpty())
            throw new RuntimeException("Data underflow");

         T item = (T) first.getValue();
         first = first.getNext();
         n--;
         return item;
      }
```

```
        public boolean isEmpty() {
          return first == null;
        }

        public int size() {
          return n;
        }

// string representation
        public String toString() {
          String s = "";

          for (Unit x = first; x != null; x = x.getNext()) {
             s += x.getValue() + ", ";
          }
          return "[ " + s + "]";
        }

   }
```

Unit Class

```
public class Unit<T>
   {
      private T element;
      private Unit<T> next;

   //Constructors
        public Unit() {
          next = null;
          element = null;
        }

        public Unit(T elem) {
          next = null;
          element = elem;
        }

        public Unit(T elem, Unit<T> unit) {
          next = unit;
          element = elem;
        }
```

```
//Getters and Setters

    public Unit<T> getNext() {

       return next;
   }

    public void setNext(Unit<T> unit) {
       next = unit;
   }

    public T getValue() {

       return element;
   }

    public void setValue(T elem) {

       element = elem;
   }
}
```

B3. EXPERIMENT 2

This Program consisted of 2 Java classes; a client class and a datastructure class.

Client Class

```
public class DebuggingAssignment2Client {

       public static void main(String[] args) {

         String[] list = {"Nick Fairley", "Zac Etheridge", "Michael Dyer",
"Cameron Newton", "Darvin Adams", "Demond Washington", "Kodi Burns", "Wes
Byrum", "Onterio McCalebb", "Philip Lutzenkirchen",
               "Lee Ziemba", "Terrell Zachery"};

         DataStructure<String> playerList = new DataStructure <String>();
         for (int i = 0; i < 12; i++) {
            playerList.insert(list[i]);
         }

         for(int i=0;i<playerList.size();i++) {

            for (int y = playerList.size(); y>i; y--) {

               if(playerList.get(y-1).compareTo(playerList.get(y))>0) {
                  String temp = playerList.remove(y);
                  playerList.insert(temp, y-1);
               }
            }
         }

         System.out.println("Sorted List:");
         for (int i = 0; i<playerList.size(); i++ ) {
            System.out.println(i+1+". "+playerList.get(i));
         }

      }
   }
```

DataStructure Class

```
public class DataStructure<T> {

      private int size;
      private Unit head;
      private Unit last;
       public DataStructure() {
      }

       public void insert(T value) {
         Unit node = new Unit(value);
         if (head != null) {
            node.prev = head.prev;
            node.next = head;
            head.prev = node;
            last.next = node;
         }
         else {
            node.next = node;
            node.prev = node;
            last = node;
         }
         head = node;
         size++;
      }


       public void insert(T value, int index) {

         if (index == 0) {
            insert(value);
            return;
         }
         Unit node = new Unit(value);

         Unit prev = head;

         for (int i = 1; i < index; i++) {
            prev = prev.next;
         }
```

```
    node.next = prev.next;
    node.prev = prev;
    node.next.prev = node;

    if (last == prev) {
        last = node;
    }
    prev.next = node;

    size++;
}

 public T remove(int index) {
    if (index == 0) {
        Unit result = head;
        head = head.next;
        if (head == result) {
            head = null;
            last = null;
        }
        else {
            head.prev = last;
            last.next = head;
        }
        size--;
        return (T)head.getValue();
    }

    Unit prev = head;
    for (int i = 1; i < index; i++) {
        prev = prev.next;
    }
    Unit result = prev.next;
    prev.next = prev.next.next;

    if (prev.next == last) {
        last = prev;
    }
    if (prev.next != null) {
        prev.next.prev = prev;
```

```
      }
      size--;
      return (T)result.getValue();
   }

    public int size() {
      return size;
   }

    public T get(int index) {
      Unit node = head;
      for (int i = 0; i <= index; i++) {
         node = node.next;
      }
      return (T)node.value;
   }

private class Unit<T> {

      Unit next;
      Unit prev;
      T value;

       public Unit() {
      }

       public Unit(T initValue) {
         value = initValue;
      }

       public T getValue() {
         return value;
      }

       public void setValue(T newValue) {
         value = newValue;
      }
   }
}
```

Appendix C

MENTAL MODEL QUESTIONNAIRE

Function

- What is the function of the program?
- Can you briefly tell how does the program achieve this?

Static/Dynamic Slice

- The following code snippet was picked from the program,

```
if(……) {
    s1.insert(input.substring(i,i+1) + " ");
}
else {
    s1.insert(" ");
}
```

  As the conditional statement currently stands in your program, do you think instance variable 's1' can be modified by both the statements?

Static/Dynamic Data Structure

- Which known data structure does the class 'Datastructure' represent?
- Is this data structure static or dynamic in nature?
- How many elements can an instance of the 'Datastructure' class hold?
- How many elements can character array 'inputTest' hold?

Control flow

- Which is the first method from DataStructure class that is invoked by the Client?

- When is the toString() method of the 'DataStructure' class invoked?
    1. Before remove() method
    2. After remove() method
    3. Never invoked
- In the program, when is the 'Datastructure' instance s1 checked for it being empty?
    1. Before invoking remove
    2. After invoking remove
    3. Both
    4. Neither of the above
- Describe the order in which the four classes in this project are invoked/processed?
- In what order are elements picked up from the 'input' String?

Structure

- How is the 'Unit' class related to the 'DataStructure' class?
- How is 'DebuggingAssignment1Client' class related to the 'Unit' class?
- Does the Client class create an instance of Unit class?
- How is the 'DSFrame' related to the 'DataStructure'?
- How is 'DebuggingAssignment1Client' class related to 'DataStructure' class?

Data Flow

- How does the insert() method affect the functioning of the size() method in the DataStructure Class?
- When the remove() method is invoked, which element is removed from the data structure?
- When the insert() method is invoked, which position is the new element inserted in the data structure?
- How does the remove() method affect the functioning of the size() method in the DataStructure Class?
- In the Datastructure class, how does the insert() method affect the functioning of the isEmpty() method?

Appendix D

INTERVIEW QUESTIONS

- What was your strategy while debugging the first program?
- How about the strategy with the second program?
- How did you overcome the challenges posed by the program?
- Which representation shown by the IDE was the most helpful?
- How would you rate you debugging experience with the IDE?
- How important were the following for your decision to use the representations?
    - Visual appeal, data values, correlation to program execution
- In general, how often do you debug using a representation?
- What could be done to improve the IDE and its debugging experience?
- Why the Viewer was used, why did he/she think it was appropriate?
- Was it helpful in the end?
- Does the fact that jGRASP viewer shows you real time manipulations help you?
- Why did you (not) use the UML diagram? Did it help?
- Why did you choose to use jGRASP viewer over the variable window to debug?

Appendix E

DEBUGGING PERFORMANCE

E1.1 Program One

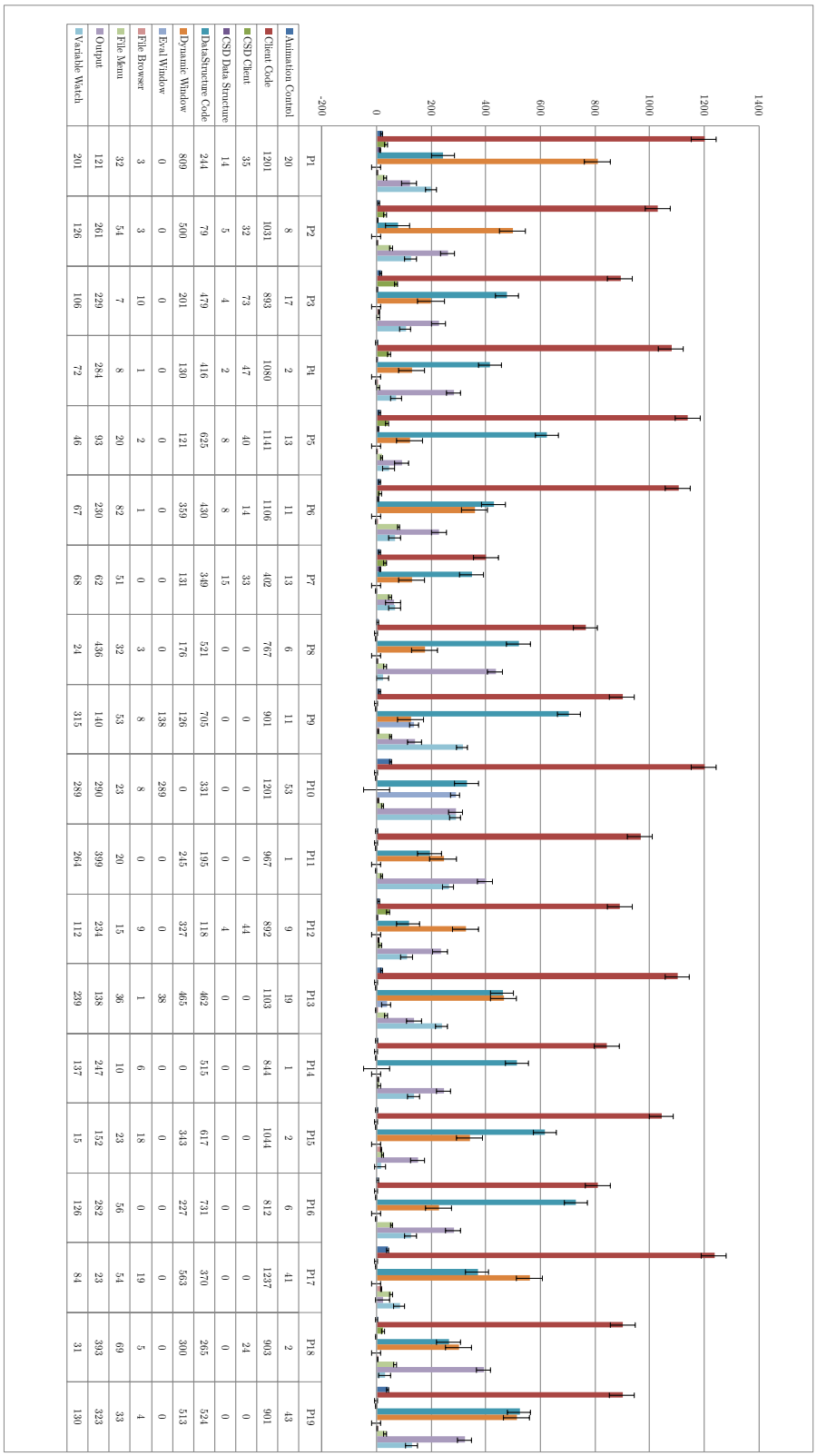
		Bug Type					
	Time Taken	**index value**	**loop condition**	**remove n--**	**remove first**	**Group***	**Bugs Fixed**
P1	8:35	3:58	8:35	6:19	6:19	SV	4
P2	15:00					SV	0
P3	15:00			5:47	12:32	SV	2
P4	14:27	5:28	12:49	8:08	14:27	SV	4
P5	15:00	12:41	16:04	6:01		SV	3
P6	11:52	6:19	11:52	5:46	5:01	SV	4
P7	12:20	4:00	12:20	7:01	7:01	SV	4
P8	15:00	6:55				DV	1
P9	14:30	3:01		14:25		SV	2
P10	15:00	7:11	13:29			DV	2
P11	15:00	14:47	15:00			DV	2
P12	15:06				15:06	DV	1
P13	15:05	13:09			15:05	DV	2
P14	8:20	8:16	8:18	6:25	6:24	DV	4
P15	14:25	10:30	12:34	14:23	14:20	DV	4
P16	15:00	12:42	15:00	13:29	13:26	DV	4
P17	9:40	4:44	8:04	9:40	9:20	DV	4
P18	14:41	8:07	8:58	14:09	14:41	DV	4
P19	15:00					SV	0

* SV – Static Visualizations, DV – Dynamic Visualizations

E2.1 Program Two

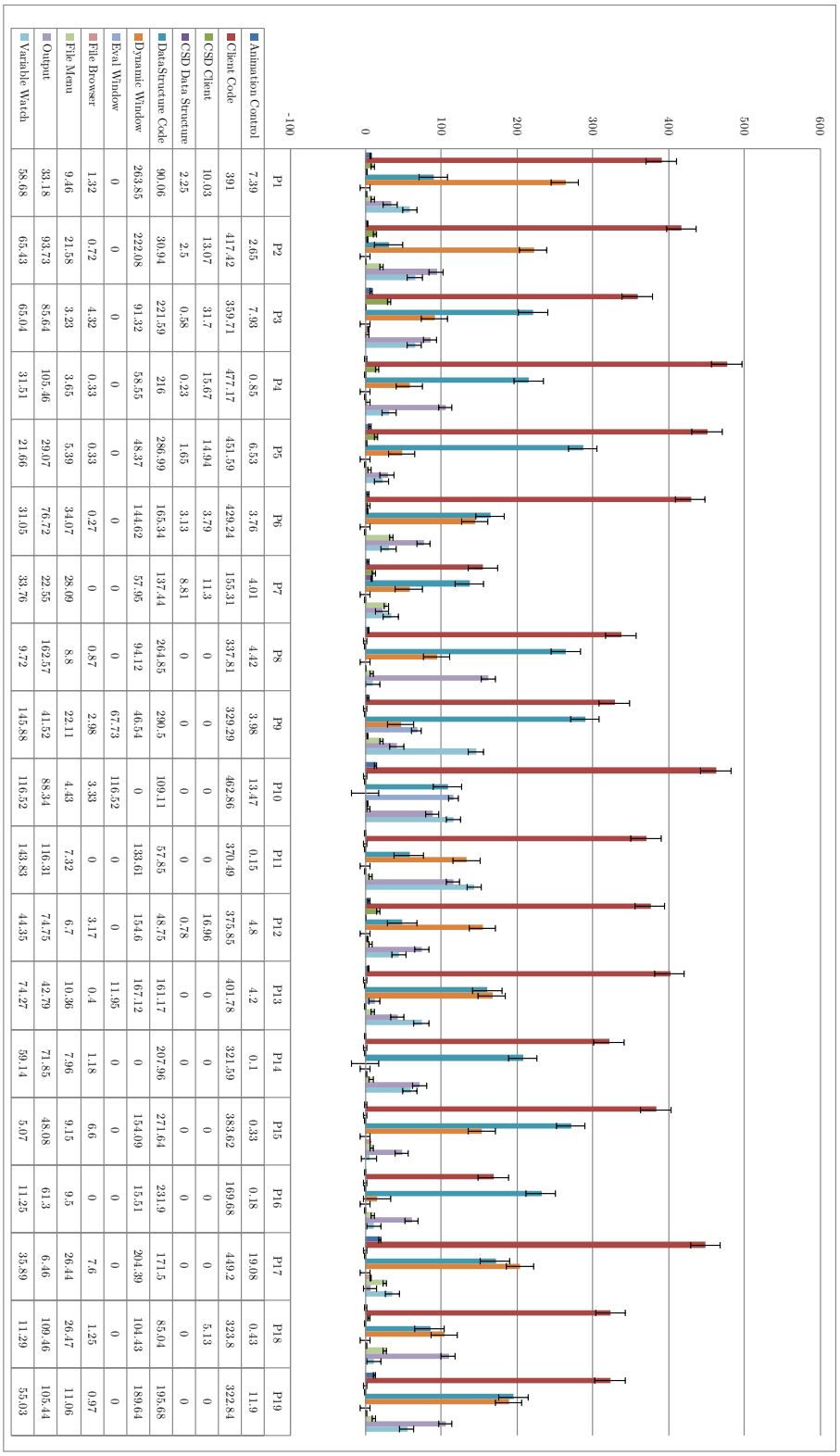
		Bug Type			
	Total Time	**size**	**get() method**	**index**	**Bugs Fixed**
Participant1	15:00			13:05	1
Participant2	15:00				0
Participant3	14:53	11:09	5:26	14:53	3
Participant4	15:00	7:03			1
Participant5	15:00			5:31	1
Participant6	14:55	2:45	14:55		2
Participant7	5:49	2:16	4:46	5:49	3
Participant8	15:00			2:23	1
Participant9	15:00				0
Participant10	15:00	7:51			1
Participant11	15:00			12:13	1
Participant12	13:10			10:03	1
Participant13	15:00				0
Participant14	11:34	4:47	9:05	11:34	3
Participant15	15:00	6:26		4:20	2
Participant16	15:00	9:01	13:39		2
Participant17	15:00	15:00	5:56	9:15	3
Participant18	15:00	2:59		1:19	2
Participant19	14:55	14:55	12:01		2

E2.2 Debugging Experiment Two (Total Fixation Count)



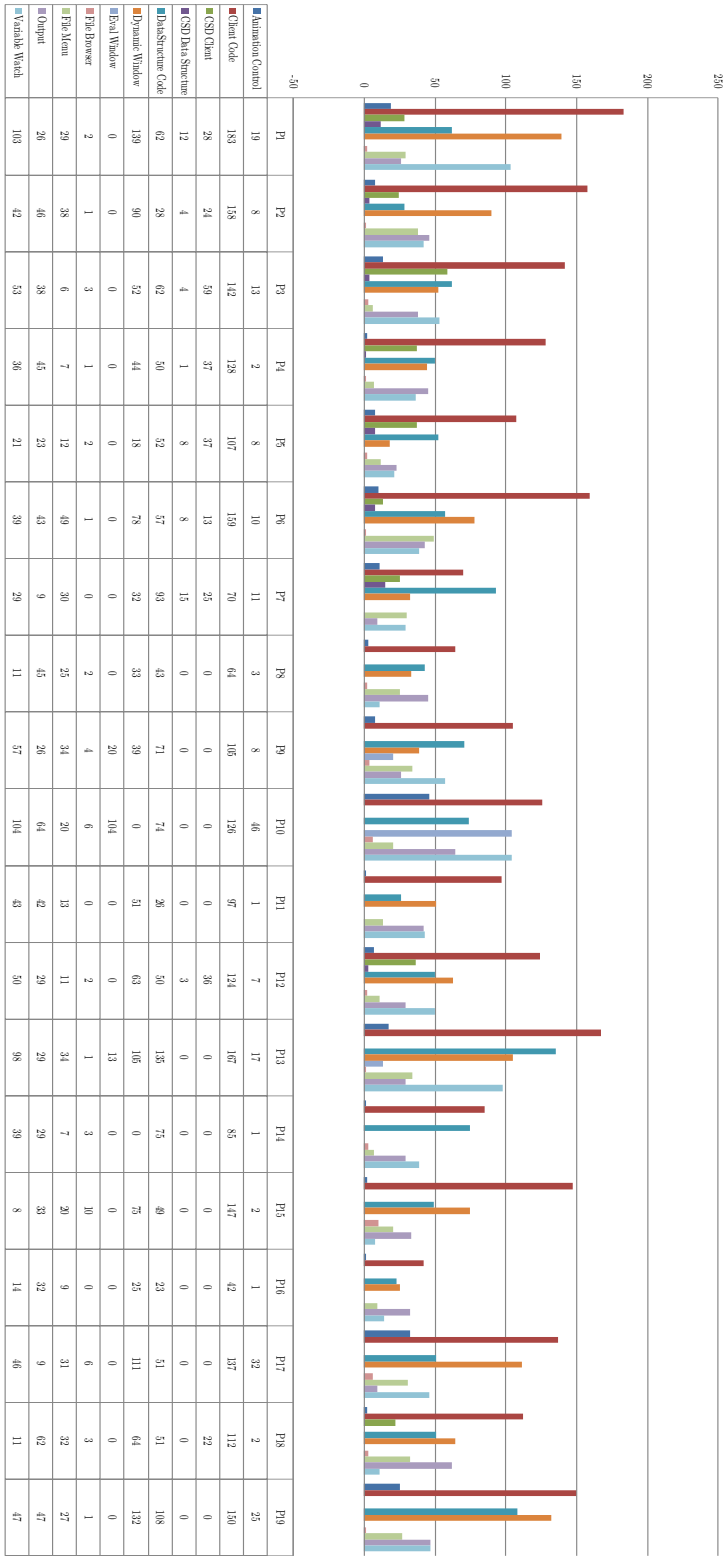
	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19
Animation Control	20	8	17	2	13	11	13	6	11	53	1	9	19	1	2	6	41	2	43
Client Code	1201	1031	893	1080	1141	1106	402	767	901	1201	967	892	1103	844	1044	812	1237	903	901
CSD Client	35	32	73	47	40	14	33	0	0	0	0	44	0	0	0	0	0	24	0
CSD Data Structure	14	5	4	2	8	8	15	0	0	0	0	4	0	0	0	0	0	0	0
DataStructure Code	244	79	479	416	625	430	349	521	705	331	195	118	462	515	617	731	370	265	524
Dynamic Window	809	500	201	130	121	359	131	176	126	0	245	327	465	0	343	227	563	300	513
Eval Window	0	0	0	0	0	0	0	0	138	289	0	0	38	0	0	0	0	0	0
File Browser	3	3	10	1	2	1	0	3	8	8	0	9	1	6	18	0	19	5	4
File Menu	32	54	7	8	20	82	51	32	53	23	20	15	36	10	23	56	54	69	33
Output	121	261	229	284	93	230	62	436	140	290	399	234	138	247	152	282	23	393	323
Variable Watch	201	126	106	72	46	67	68	24	315	289	264	112	239	137	15	126	84	31	130

E2.3 Debugging Experiment Two (Total Dwell Time)



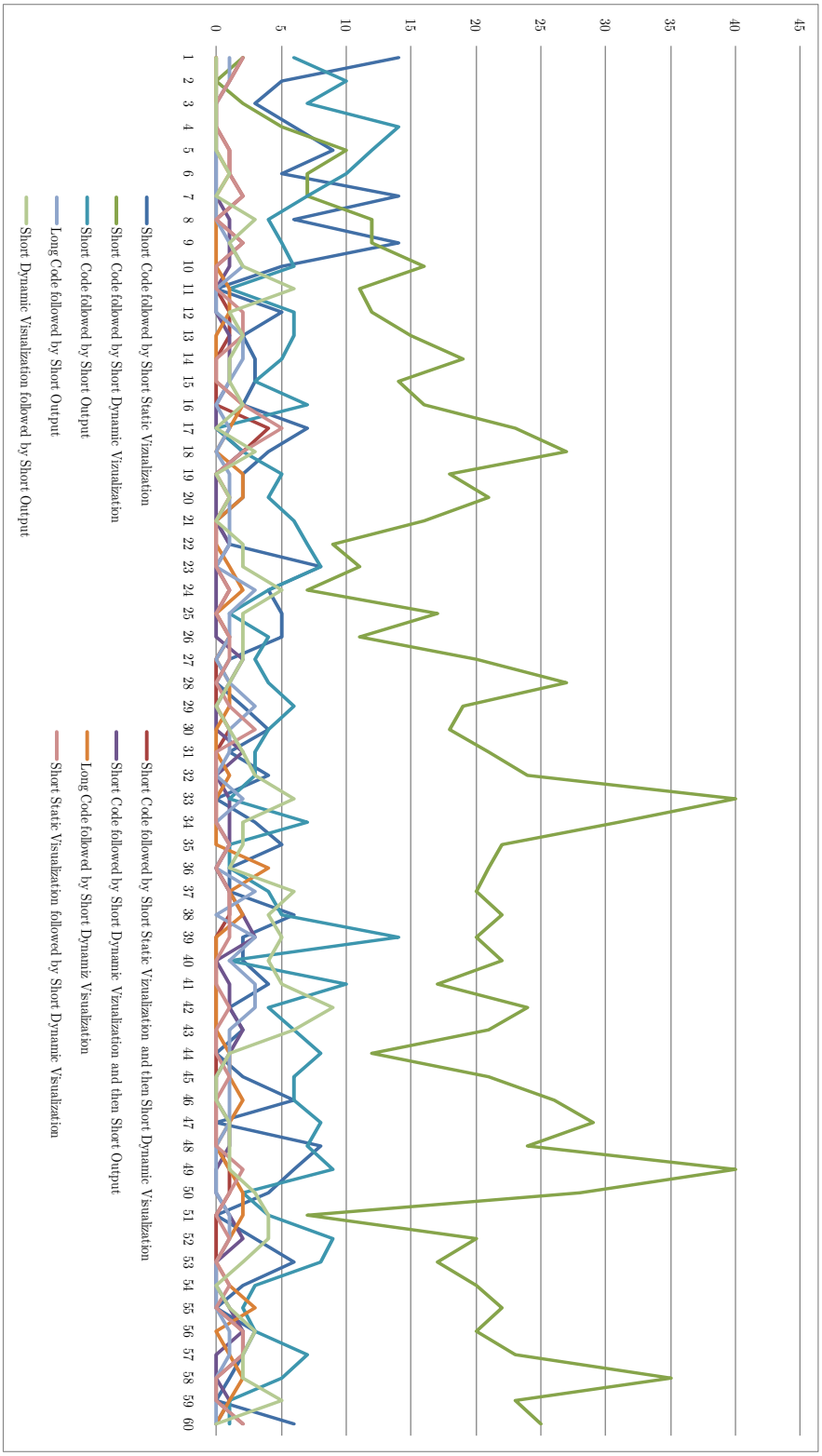
	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19
Animation Control	7.39	2.65	7.93	0.85	6.53	3.76	4.01	4.42	3.98	13.47	0.15	4.8	4.2	0.1	0.33	0.18	19.08	0.43	11.9
Client Code	391	417.42	359.71	477.17	451.59	429.24	155.31	337.81	329.29	462.86	370.49	375.85	401.78	321.59	383.62	169.68	449.2	323.8	322.84
CSD Client	10.03	13.07	31.7	15.67	14.94	3.79	11.3	0	0	0	0	16.96	0	0	0	0	0	5.13	0
CSD Data Structure	2.25	2.5	0.58	0.23	1.65	3.13	8.81	0	0	0	0	0.78	0	0	0	0	0	0	0
DataStructure Code	90.06	30.94	221.59	216	286.99	165.34	137.44	264.85	290.5	109.11	57.85	48.75	161.17	207.96	271.64	231.9	171.5	85.04	195.68
Dynamic Window	263.85	222.08	91.32	58.55	48.37	144.62	57.95	94.12	46.54	0	133.61	154.6	167.12	0	154.09	15.51	204.39	104.43	189.64
Eval Window	0	0	0	0	0	0	0	0	67.73	116.52	0	0	11.95	0	0	0	0	0	0
File Browser	1.32	0.72	4.32	0.33	0.33	0.27	0	0.87	2.98	3.33	0	3.17	0.4	1.18	6.6	0	7.6	1.25	0.97
File Menu	9.46	21.58	3.23	3.65	5.39	34.07	28.09	8.8	22.11	4.43	7.32	6.7	10.36	7.96	9.15	9.5	26.44	26.47	11.06
Output	33.18	93.73	85.64	105.46	29.07	76.72	22.55	162.57	41.52	88.34	116.31	74.75	42.79	71.85	48.08	61.3	6.46	109.46	105.44
Variable Watch	58.68	65.43	65.04	31.51	21.66	31.05	33.76	9.72	145.88	116.52	143.83	44.35	74.27	59.14	5.07	11.25	35.89	11.29	55.03

E2.4 Debugging Experiment Two (Total Visit Count)



	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19
Animation Control	19	8	13	2	8	10	11	3	8	46	1	7	17	1	2	1	32	2	25
Client Code	183	158	142	128	107	159	70	64	105	126	97	124	167	85	147	42	137	112	150
CSD Client	28	24	59	37	37	13	25	0	0	0	0	36	0	0	0	0	0	22	0
CSD Data Structure	12	4	4	1	8	8	15	0	0	0	0	3	0	0	0	0	0	0	0
DataStructure Code	62	28	62	50	52	57	93	43	71	74	26	50	135	75	49	23	51	51	108
Dynamic Window	139	90	52	44	18	78	32	33	39	0	51	63	105	0	75	25	111	64	132
Eval Window	0	0	0	0	0	0	0	0	20	104	0	0	13	0	0	0	0	0	0
File Browser	2	1	3	1	2	1	0	2	4	6	0	2	1	3	10	0	6	3	1
File Menu	29	38	6	7	12	49	30	25	34	20	13	11	34	7	20	9	31	32	27
Output	26	46	38	45	23	43	9	45	26	64	42	29	29	29	33	32	9	62	47
Variable Watch	103	42	53	36	21	39	29	11	57	104	43	50	98	39	8	14	46	11	47

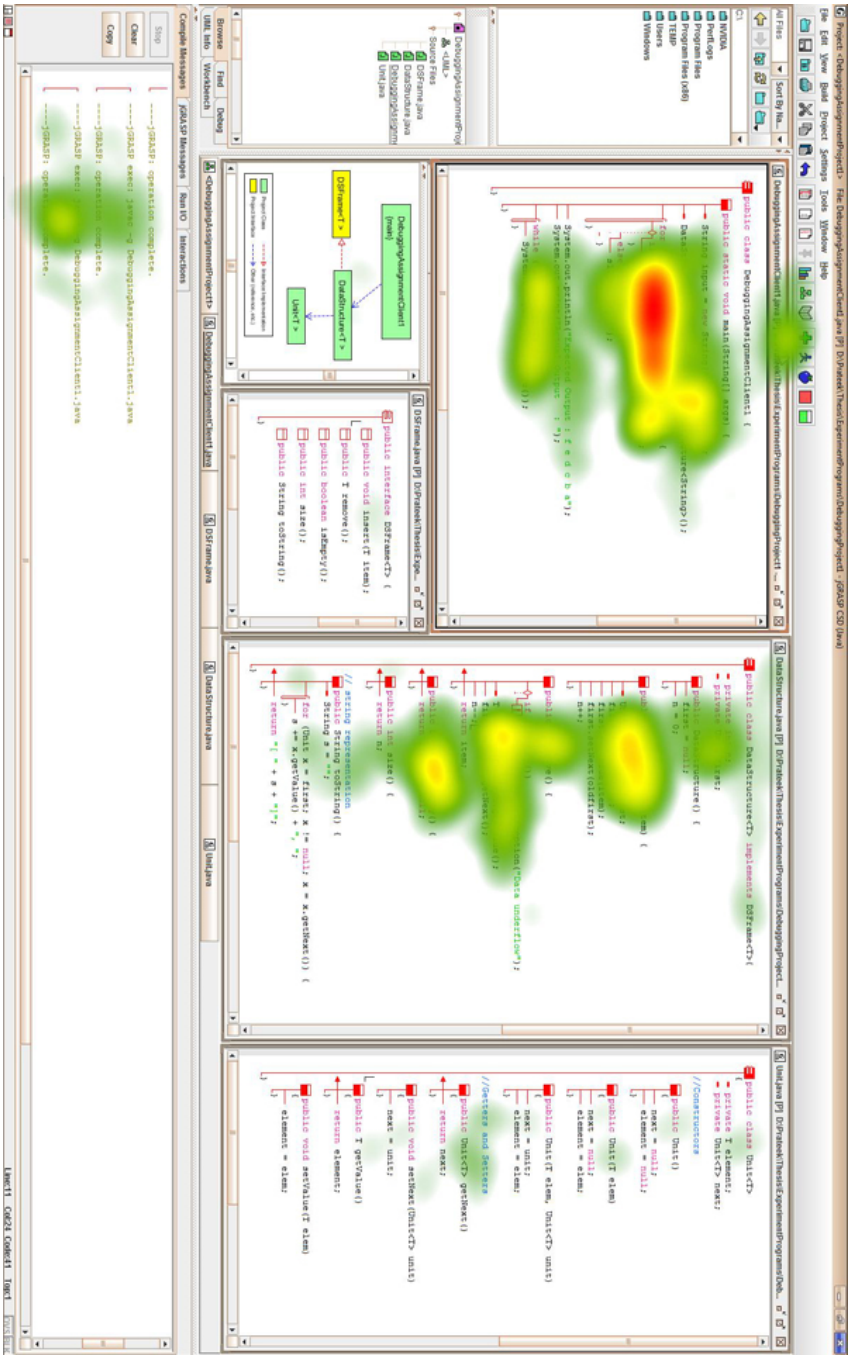
E2.5 Debugging Experiment Two (Visual Patterns)



Appendix F

HEAT MAPS AND GAZE PLOTS

Heatmap based on fixation duration for participant nine and account for a section where dynamic representation was used



Heatmap based on fixation duration for participant three's debugging session