UNIVERSIDAD
NACIONAL
DE COLOMBIA

Empirical Evaluation of the Usefulness of Graph-based Visualization Techniques to Support Software Understanding

Germán Oswaldo Cárdenas Caro

Universidad Nacional de Colombia

Facultad de Ingeniería, Departamento de Ingeniería de Sistemas e Industrial

Bogotá, Colombia

2016

Empirical Evaluation of the Usefulness of Graph-based Visualization Techniques to Support Software Understanding

Germán Oswaldo Cárdenas Caro

A Thesis presented in partial fulfillment for the degree of Master in Engineering - Systems and Computing

Advisor
Ph.D., McS., Jairo Hernán Aponte Melo

Research Line:
Software Engineering

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería de Sistemas e Industrial.
Bogotá, Colombia
2016

Dedication

A mi familia entera

Acknowledgement

I would like to thank my advisor professor Jairo Hernán Aponte for their constant support in this thesis.

I would like to thank to the people of Colombia and to the National University of Colombia. I am really happy to give back this small retribution to everything I have received.

Resumen

Muchos investigadores han señalado la falta de estudios empíricos que sistemáticamente examinen las ventajas y desventajas del uso de técnicas de visualización para soportar la comprensión del software. Estos estudios son indispensables para recolectar y analizar evidencia objetiva y cuantificable acerca de la utilidad de las técnicas de visualización y herramientas propuestas, y más aún, para servir como guía de la investigación en visualización de software. En este estudio, 6 tareas típicas de comprensión de software fueron realizadas por 20 estudiantes de ingeniería de software. Se midió el tiempo de respuesta y se calificó la exactitud en las respuestas de los participantes. Los resultados indican que, por una parte, el uso de la técnica de visualización basada en grafos mejoró la exactitud en las respuestas de los estudiantes (21.45% en promedio), por otra parte, no se encontró evidencia de reducción en el tiempo gastado por los estudiantes para resolver las tareas de comprensión de software.

Palabras clave: visualización de software; experimento controlado, comprensión de software.

Abstract

Many researchers have highlighted the scarcity of empirical studies that systematically examine the advantages and disadvantages of the use of visualization techniques for software understanding activities. Such studies are crucial for gathering and analyzing objective and quantifiable evidence about the usefulness of proposed visualization techniques and tools, and ultimately, for guiding the research in software visualization. This paper presents a controlled experiment aimed at assessing the impact of a graph-based visualization technique on comprehension tasks. Six common comprehension tasks were performed by 20 undergraduate software engineering students. The completion time and the accuracy of the participants' responses were measured. The results indicate that on one hand the use of the graph-based visualization increases the correctness (by 21.45% in average) but on the other hand it does not reduce the completion time in program comprehension tasks.

Keywords— software visualization; controlled experiment; software comprehension

Content

Lista de figuras

Pág.

Lista de tablas

Pág.

Introduction

In order to analyze and understand large-scale software several techniques have been developed and one of the most interesting is software visualization. This approach takes advantage of the human's brain ability to recognize and understand graphic patterns and images and has been extensively used by many researchers to propose a wide variety of techniques and supporting tools. Unfortunately, not many of them have been empirically evaluated, which indicates that the systematic, disciplined, and controlled method for evaluating visualization techniques provided by experimentation has hardly been used [1, 2, 3]. As a consequence, several primary studies have highlighted the need for an objective evaluation of the proposed visualization techniques that allows researchers and practitioners to identify the pros and cons of applying them for performing typical software engineering activities. Among the existing empirical evaluation of software visualization tools, most controlled experiments are dedicated to the validation of the tools developed by the authors of those studies.

This thesis presents a controlled experiment aimed at evaluating an independent graph-based visualization technique, not developed by the authors. Graph-based visualizations are the most popular techniques used to represent software architectures [2] and explicitly depict software organization and its key aspects [4]. Our purpose is to evaluate the efficiency and effectiveness of this visualization technique at supporting typical software comprehension tasks. Source-code-based exploration technique is our chosen baseline, since it is the common way to perform software understanding [1, 5].

Goals

The general goal of this work is to provide empirical evidence of the effectiveness and efficiency of graph-based visualization techniques to support software understanding. The following are the specific goals to achieve in the current study:

1. To build the current state of the art on empirical evaluation of graph-based software visualization techniques.
2. To select a tool and a set of software systems in order to generate the software visualizations for the assessment.
3. To design and perform an empirical assessment of the visualization technique in which we will collect qualitative and quantitative data.
4. To analyze the data collected in order to determine the strengths and weaknesses of graph-based visualization techniques used to support software understanding.
5. To present results of the empirical assessment so that the conducted experiments can be completely replicated by other researchers.

Contributions

This work makes the following main contributions:

1. A state of the art of empirical evaluations of graph-based visualization techniques.
2. A well-designed controlled experiment to determine benefits and drawbacks of the graph-based visualization technique to support software comprehension.
3. Article: "An Empirical Assessment of the Graph-based Visualization Technique" submitted and accepted in the International Conference on Information Systems and Computer Science (INCISCOS 2016).
4. A complete set of study materials for reviewing and replication purposes[1].

Thesis Outline

The document is structured as follows:

[1] https://drive.google.com/open?id=0B6lbY7sU2RMIMW1LclRhTGc5OTQ

1. **Chapter 1** shows the background and related work which was used as starting point for the research.
2. **Chapter 2** delineates all the considerations taking into account to design the controlled experiment.
3. **Chapter 3** presents overall and per task results of the experiment.
4. Chapter 4 draws some conclusions and presents the future work.

1. Background

In the current section the background and related work used for this thesis is presented. This include relevant concepts and definitions, representative tools and techniques, and the related work performed by researchers in order to empirically evaluate the usefulness of software visualization to support different aspects of software comprehension through controlled experiments.

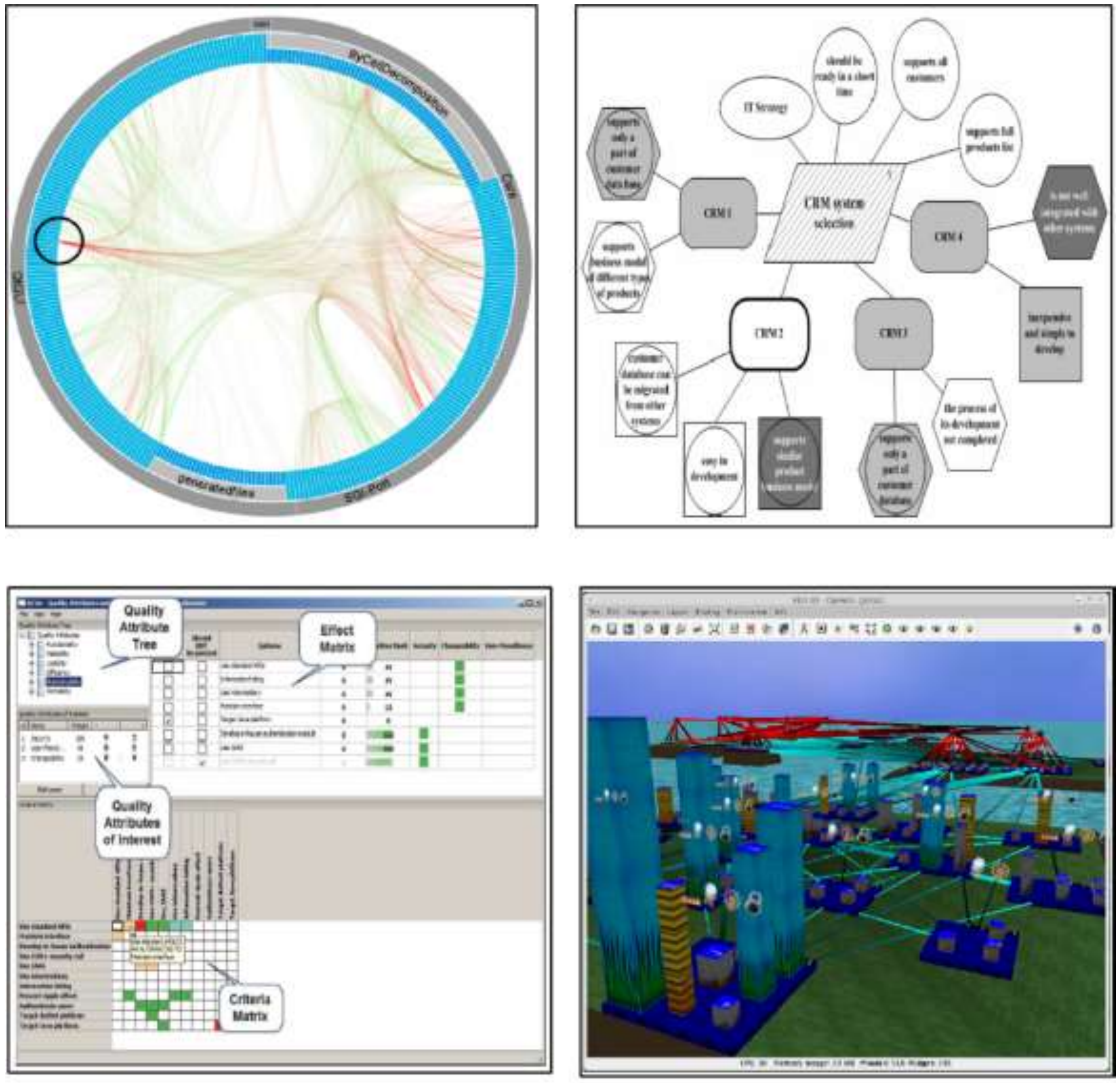
1.1 Software Visualization

Since the software system increasingly complex, the associated task to their development, maintenance, and evolution become more complex too. The more complex the system, the more difficult its understanding. It is precisely here where arise the necessity to use techniques facilitate software comprehension. Software visualization is a technique that support this task, and it is defined as "a representation of computer programs, associated documentation and data that enhances, simplifies, and clarifies the mental representation the software engineer has of the operation of a computer system" [17].

1.2 Types of Software Visualization Techniques

For the current research we have adopted the categorization made by Shahin et al. [2] whom employed thematic analysis, which is a qualitative method to identify, analyze and report patterns form a given set of data [6]. They define 4 kinds of visualization techniques, namely graph-based, notation-based, matrix-based, and metaphor-based. Since the current work only evaluate one of them (graph-based), the techniques are described at glance, and the graph-based visualization technique is presented deeper in a separate section. An example of the 4 techniques are shown in Figure 1-1.

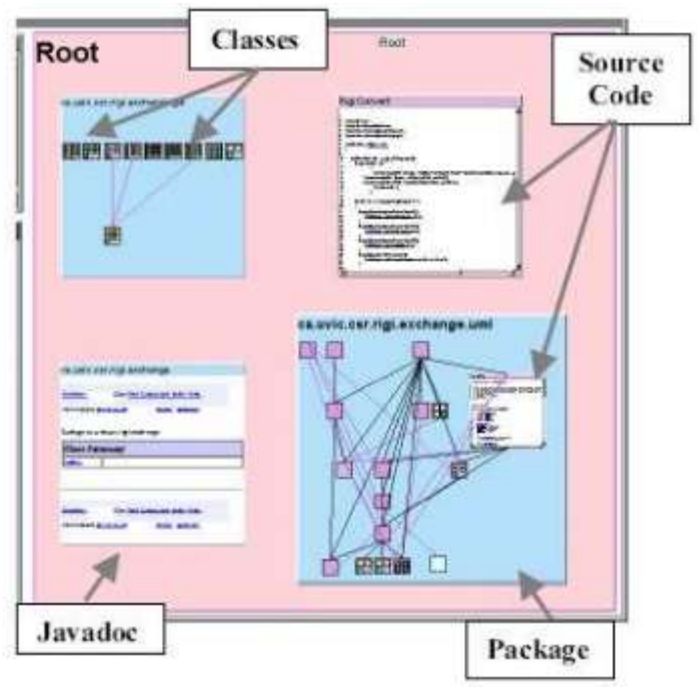
Figure 1-1: Software Visualization techniques [2].



1.2.1 Graph-based Visualization Technique

This type of visualization technique is the most used in order to represent the software structure [2]. It uses nodes to represents different software entities and edges to visualize relationships between them [7]. The early work of Storey et al. [11, 13] in which their tool Rigi is presented and evaluated is one of the most representative of the technique. Sim et al. [45] presents their graph-based tool the Searchable Bookshelf for information retrieval and software comprehension. Synytskyy et al. [46] presents their tool LSEdit, and also reports some success stories. Figure 1-2 presents a view of SHriMP [13], a tool used for visualize static aspects of software.

Figure 1-2: SHriMP's multiples views of a Java program [12].



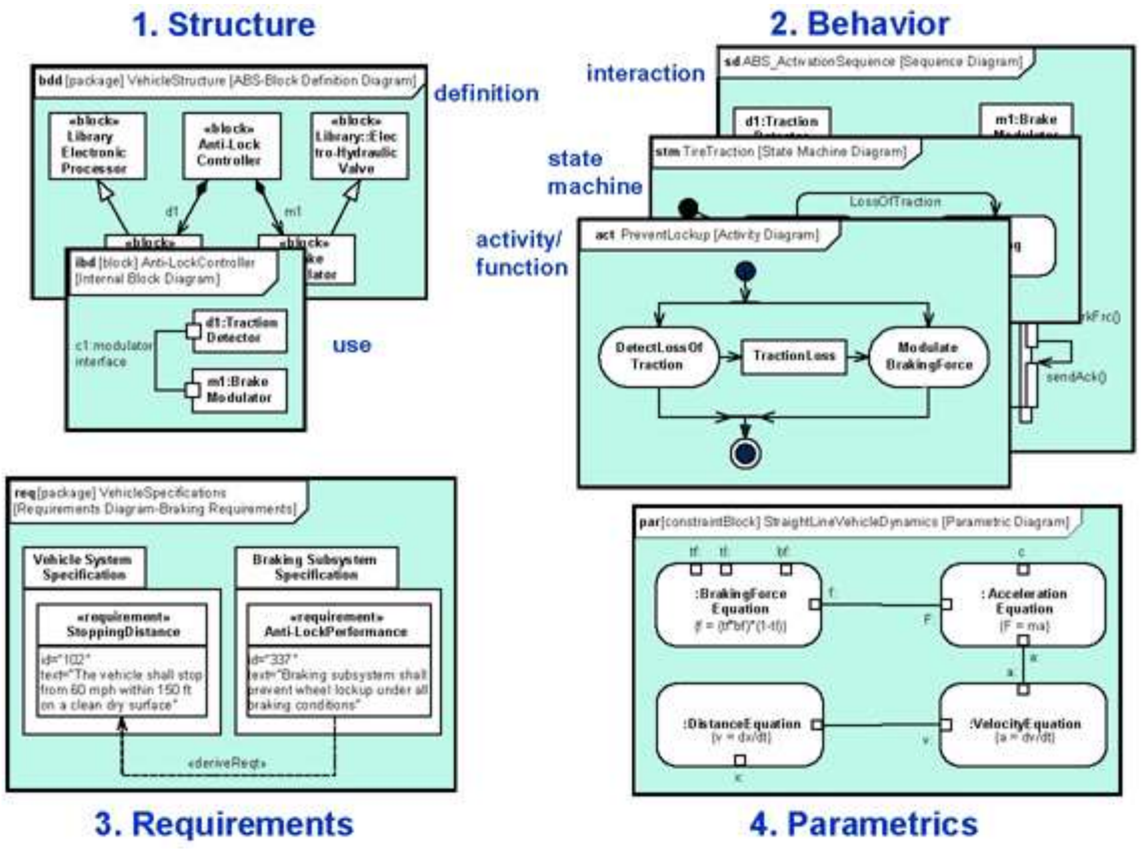
1.2.2 Notation-based Visualization Technique

Classical graphic languages to represent software such as UML (Unified Modeling Language)[2], SysML[3] (System Modeling Language), and E-R Diagrams fit in this category [2, 7]. UML is a well-known general-purpose modeling language developed and maintained by OMG (Object Management Group)[4], and is widely used in both Industry and Academia. SysML is also developed by the OMG consortium. It is defined as a dialect of UML standard,

[2] http://uml.org/
[3] http://sysml.org/
[4] http://www.omg.org/

and is used to support different source of processes such as: analysis, design, verification and validation [8]. Stratton et al. [47] reports an automatic construction of UML models form source code. Telea et al. reference Enterprise Architect, a tool for reverse engineering. Figure 1-3 shows the key diagrams of SysML.

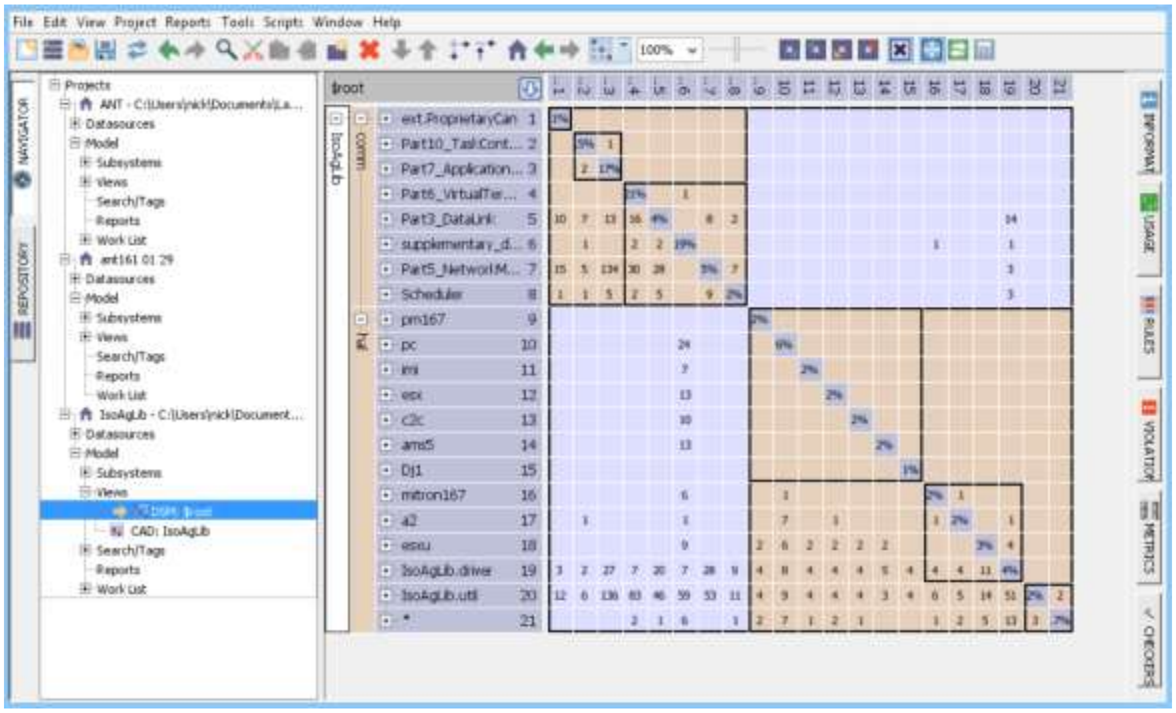
Figure 1-3: Example of SysML core diagrams.


1. Structure
2. Behavior
3. Requirements
4. Parametrics
definition
use
interaction
state machine
activity/ function
bdd [package] VehicleStructure [ABS-Block Definition Diagram]
ibd [block] Anti-LockController [Internal Block Diagram]
sd ABS_ActivationSequence [Sequence Diagram]
stm TireTraction [State Machine Diagram]
act PreventLockup [Activity Diagram]
req [package] VehicleSpecifications [Requirements Diagram-Braking Requirements]
par [constraintBlock] StraightLineVehicleDynamics [Parametric Diagram]
«block» Library Electronic Processor
«block» Anti-Lock Controller
«block» Library::Electro-Hydraulic Valve
d1:Traction Detector
m1:Brake Modulator
c1:modulator interface
DetectLossOf Traction
TractionLoss
Modulate BrakingForce
LossOfTraction
sendAck()
Vehicle System Specification
Braking Subsystem Specification
«requirement» StoppingDistance
id="102" text="The vehicle shall stop from 60 mph within 150 ft on a clean dry surface"
«requirement» Anti-LockPerformance
id="337" text="Braking subsystem shall prevent wheel lockup under all braking conditions"
«deriveReqt»
:BrakingForce Equation [f = (tf*bf)*(1-tl)]
:Acceleration Equation [F = ma]
:DistanceEquation [v = dx/dt]
:VelocityEquation [a = dv/dt]

1.2.3 Matrix-based Visualization Technique

Since the visualization of large systems may become complex and hard to follow using current visualization approaches mostly based on node-link diagrams [9], the matrix-based visualization techniques emerges as a necessary and very useful complement to support software understanding. It is less intuitive in comparison with graph-based visualization techniques, but it is able to show additional details when the graph is large or dense [2]. Beck and Diehl [49] use matrix to compare software architecture descriptions. Also, Lungu and Lanza [50] use matrix to display detailed dependency between two modules. Figure 1-4 shows a matrix-based representation of a .NET software system.

Figure 1-4: View of Lattix[5].

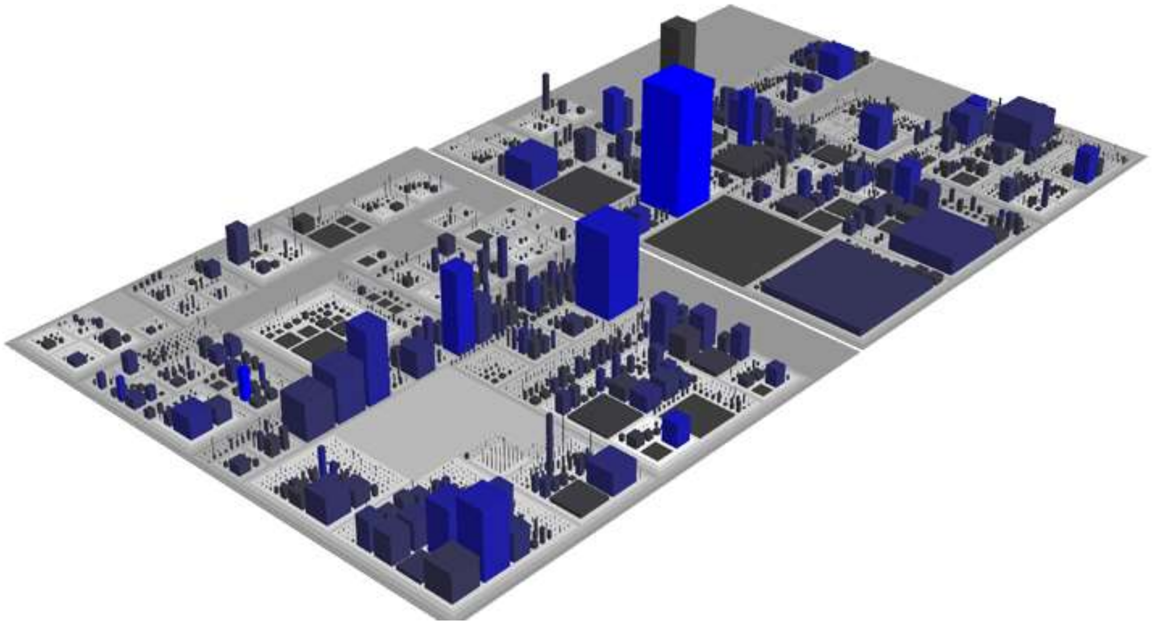


1.2.4 Metaphor-based Visualization Technique

This visualization technique is particularly effective and easy to understand due to its representation are based in real-world elements [2]. Software systems are visualized in wide variety of actual objects such as cities, landscapes, and solar system. City-metaphor typically represents software artifacts as entities that commonly make up urban environments [10]. For example, Wettel et al. [1] presents CodeCity a metaphor that represents classes as buildings, and packages as districts. Figure 1-5 shows a visualization of JDK (Java Development Kit)1.5. Balzer et al. [51] use 3D landscapes to represent the static structure of object-oriented programs.

[5] http://lattix.com/

Figure 1-5: View of CodeCity



1.3 Graph-based Visualization Technique Concepts

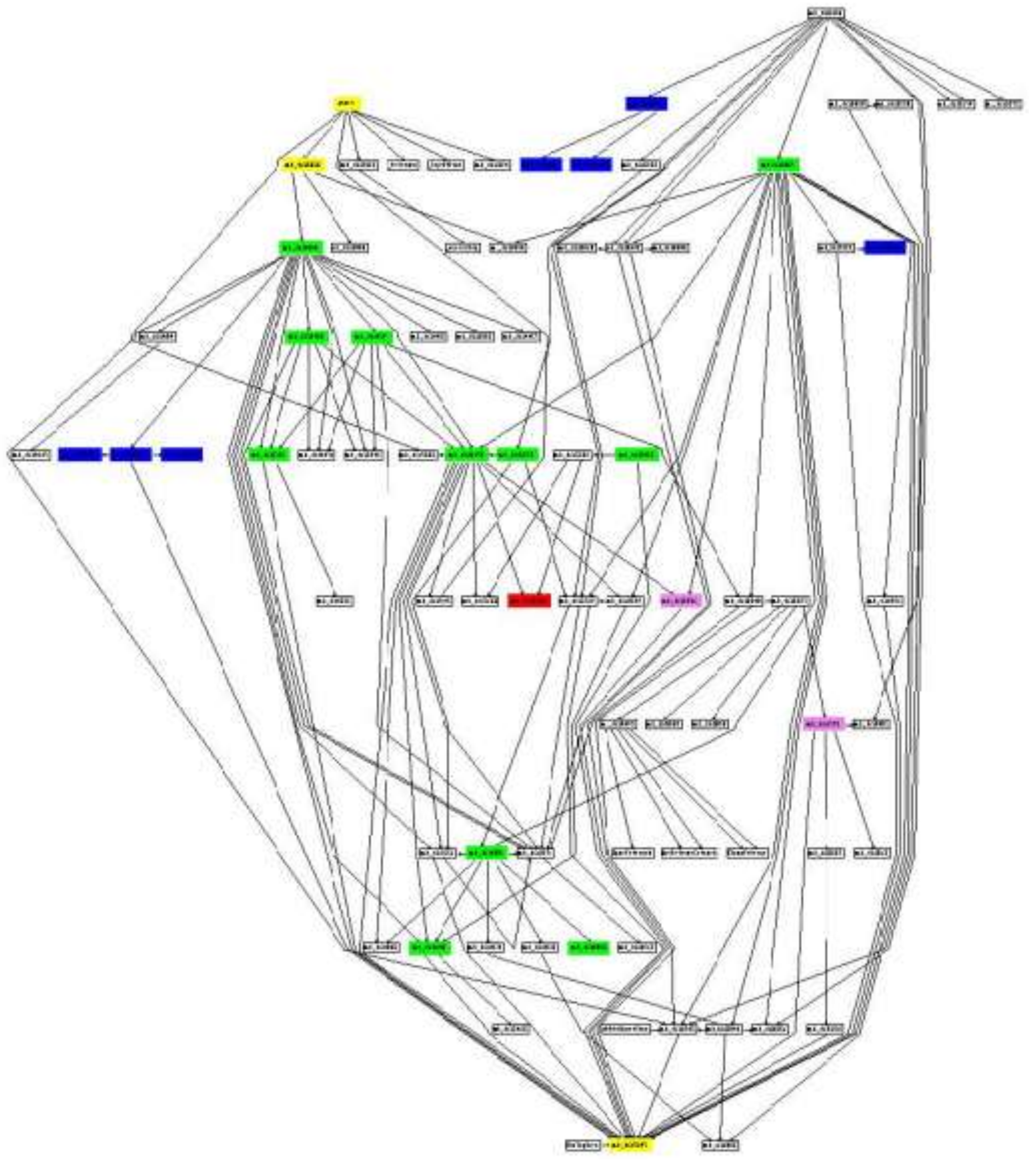
Since the current study aims to empirically assess how useful is the graph-based visualization technique to support software understanding, we consider necessary to describe deeply the concepts and definitions involved in its representation and use.

1.3.1 Graph Visualization

Graphs have emerged as a great concept to represent a huge range of types of information due to their natural capability to represent objects and relationships between objects [14]. Since graph are the fundamental structural representation of data [15], and software is inherently structured, software graph-based representation is a very obvious result. Software graph visualization represents entities as nodes and the relationships between these entities as edges. Nodes could vary depending of the desired granularity. They may represent methods, classes, packages, modules, components, subsystems, and even entire systems. Edges can visualize either static or dynamic aspects, such as inheritance, implementations, aggregation-composition or calls between methods [14].

Depending of their application context graph can be drawn in different ways. Nodes may be dots, circles, or simply labels. Edges may be straight lines, orthogonal or arbitrary polygonal paths, or curves [14]. Figure 1-6 visualize functions and the calls between them. Colors represents hot-spots.

Figure 1-6: Graph-based software representations.



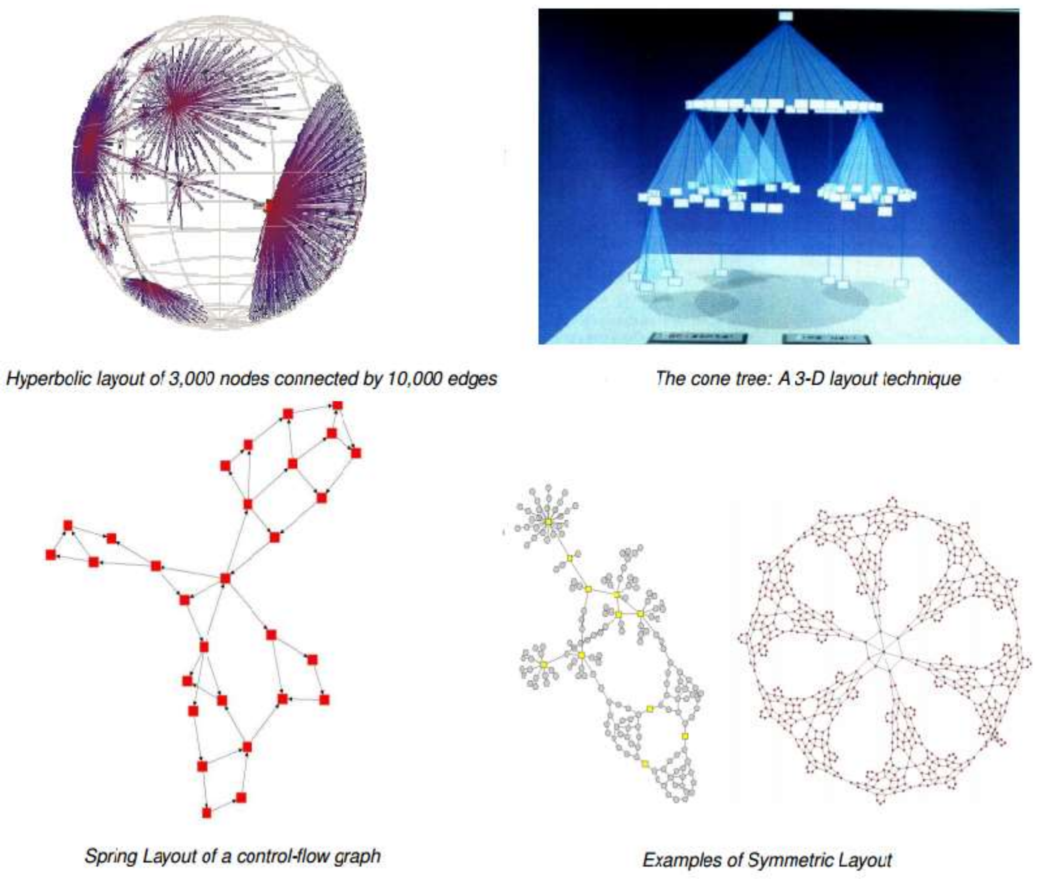
1.3.2 Graph Layouts

In this section a brief review of graph layout concepts is presented. The works of Lemieux et al. [12], Herman et al. [15], and Kaufmann et al. [16] give more detailed insights into the graph layout theory. The classification given in [12] was taken as basis to develop this section, since it is focused towards software comprehension, which is one of the key concepts of our study.

- **Tree Layout:** Tree layout has an ancestor-children composition. All children are placed immediately below of their ancestor. In general, this representation models hierarchical information using space-filling methods [10]. The most common structured- positioning are top-down, left-to-right, concentric circles, Information Cubes, and 3D cone-tree.
- **Hierarchical Layout:** This technique is the typical method to visualize directed graphs [12]. Hierarchical layouts drawn vertices in rows or levels, and edges are driven from the top to the bottom.
- **Orthogonal Layout:** In this sort of layout nodes are located both horizontally and vertically, so the edges are blended in 90-degree angles. Orthogonal layout representations have the advantage of minimize the amount of edge crossing [12].
- **Force-based Layout:** It places the nodes according to a system of forces using physical concepts. The most used algorithms are spring embedder and molecular mechanics. It combines repulsive and attractive forces and the optimal graph is shaped when the forces have been minimized [12].
- **Hyperbolic Layout:** This type of layout was developed with graph visualization and interaction in mind [15]. It reduces the amount of space that is necessary to display representations, providing a distorted view of graph structure.
- **Layout of Cluster Graphs:** This is based in the abstraction and reduction techniques [12], it has the advantage of reducing the number of visible elements improving the clarity of the diagram [15].
- **Nested Graph Hierarchy:** This technique is an expansion of graphs where each node can contain a nested graph, and it is possible to navigate through the hierarchy going to arbitrary depth.
- **Symmetric Layout:** Symmetric layout grouped nodes in symmetrical patterns by applying transformation that not to modify information presented by the graph. It is a very practical way to reduce edge crossings [12].

Figure 1-7 shows different examples of graphs layouts

Figure 1-7: Graph-based layouts.


Hyperbolic layout of 3,000 nodes connected by 10,000 edges
The cone tree: A 3-D layout technique
Spring Layout of a control-flow graph
Examples of Symmetric Layout

1.4 Empirical Software Engineering Concepts

Since main contributions of this work were obtained via empirical strategies, we considered essential to present the more important concepts used in empirical software engineering going deeper in those related to the current thesis.

1.4.1 Empirical Methods

Empirical software engineering provides a set of methods to solve any research problem, and in a great number of cases is necessary to apply a combination of them to fully fulfill concluding results [18]. Moreover, the results and their validity have a strong relationship with the method chosen.

There are two types of approaches in to empirical studies, namely exploratory research and exploratory research [19].

Exploratory research is focused to study a phenomenon is its natural setting. It is often referred to as qualitative research. Explanatory research which is focused in obtain quantitative measures of cause-effect relationships [19]. The current thesis aims for gathering information using both approaches.

The most important empirical methods' definitions are given below.

- **Controlled Experiments:** A method that seek to test a hypothesis manipulating one or more independent variables in order to measure the effect on one or more dependent variables [18].
- **Case Studies:** Empirical method which investigate a phenomenon within its real-life context. It is especially useful when are not clearly distinguishable the boundary between context and phenomenon [20].
- **Survey Research:** It is used to identify characteristics within a well-defined population by mean of the analysis of a representative sample. It is often associated with questionnaires, but it could be carry out through interviews, or data-login techniques [18].
- **Ethnographies:** This is a technique that look for understand how communities build their social interactions. This kind of technique allow researchers to investigate communities culture [21].
- **Action Research:** In this method the researcher attempts to investigate a real-life problem while tries to solve it [22]. Action research attempt to improve de researched situation instead of only to observe and measure.

1.4.2 Controlled Experiments Concepts

As we mentioned above, an experiment in software engineering is an inquiry which modify one or more for measuring the effect on one or more independent variables. It is mostly performed in controlled environments. According with Wohlin et al. [19] is the experiment is not randomized, it is termed quasi-experiment. Quasi-experiments are empirical methods similar to “typical” experiments, where treatments are not assigned to subjects randomly, they are assigned according with some criteria that arise from the subjects themselves. Our empirical assessment belongs to this category because subjects were allocated in the experimental treatment based on their performance in the software engineering course.

In order to understand the experimental process, it is necessary to introduce some basic concepts.

- **Independent Variables:** Variables in the process that are manipulated and controlled.
- **Dependent Variables:** Variables which value is measured to know the effect caused in response to a change in the independent variable.
- **Subjects:** Often called participants. The term is referred to the people who apply the treatment.
- **Objects:** They are the artifacts over the treatment is applied by the subjects.
- **Treatment:** It is a combination of values of the independent variables. The treatments are applied to a combination of objects and subjects.

1.4.3 Experimental Process

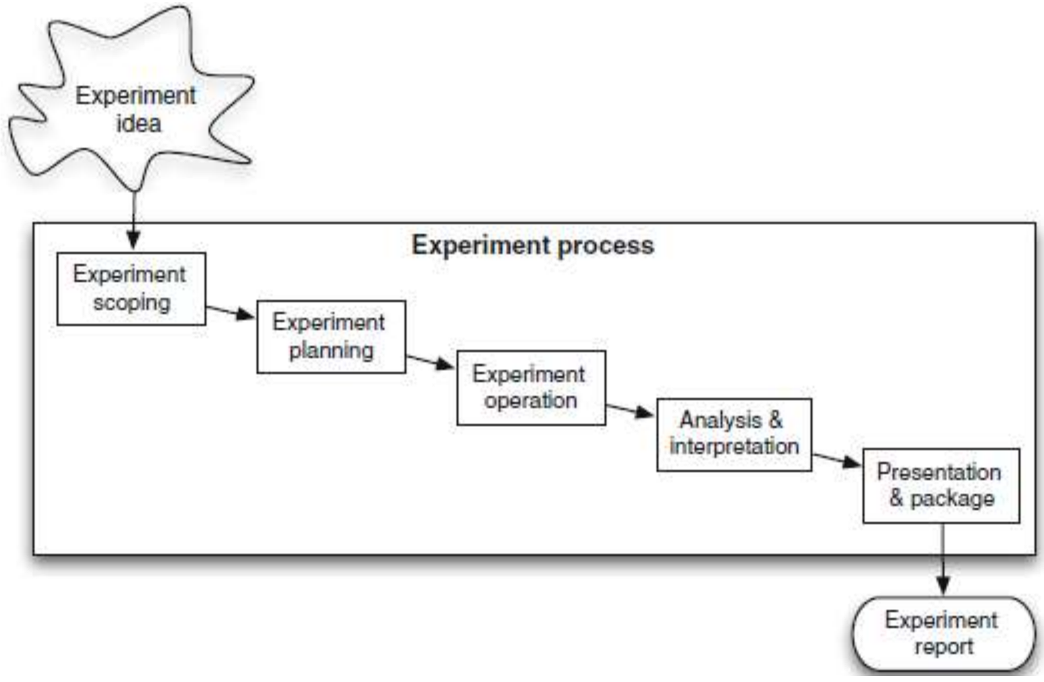
The experimental process is made up by several and well-defined series of steps. In order to building this section, we took the activities structure presented by Wohlin et al. [19].

- **Scoping:** This is the first activity of the experimental process. The hypothesis has to be clear, and both objectives and goals must be defined.
- **Planning:** The context of the experiment is determined in the planning step. It includes define participants and environment of the experiment, and formally state null and alternative hypothesis. Even more, the independent and dependent variables are selected and their possible values and scales. Threats to validity are also determined in this step. The most important product of this step is the experimental design.
- **Operation:** In this stage the preparation, execution and data validation of the experiment take place. In the preparation the subjects and material are readied for the experiment. The execution step is related with the accomplishment of the experimental design and the data collection. Data validation is concerned of ensuring that the data is correct and representative.

- **Analysis and Interpretation:** It is the phase where the collected data in the previous stage is analyzed and interpreted. Data is understood by mean of descriptive statistics. Strange measures are removing by using data removing methods. Finally, it is determined if the hypothesis is rejected by using statistical tests.
- **Presentation and package:** This is the final activity of the experimental process, results of the experiment could be presented by means of a technical paper, and a lab package can be elaborated for either validation or validation purposes.

The experimental process is described in figure 1- 8.

Figure 1-8: Overview of the experimental process [19].



1.5 Software Comprehension Concepts

Software or program comprehension is the activity by means programmers understand how a software system or a part of it works [23]. It is an intensive activity in which developers spend about 50% of their time [24, 25]. This section presents an overview of the different concepts and strategies in software comprehension. We decided to take the approach given by Storey [26].

1.5.1 Definitions

- **Mental Model:** It portrays the developer’s mental representation of the software system.
- **Cognitive Model:** It depicts the cognitive process that take place in the developer’s mind to form the mental model.
- **Cognitive Support:** It assists cognitive tasks such as thinking or reasoning [27].
- **Beacons:** They are familiar patterns or features that work as clue to identify program structures.
- **Rules of programming:** Rules referred to coding standards and implementations [26].

1.5.2 Cognitive Theories

- **Top-down comprehension:** This theory claims that developers understand programs from top to de down, i.e. mapping domain knowledge into source code. It is used especially when the code is familiar for the developer [26].
- **Bottom-up comprehension:** In this theory software understanding is gained firstly reading pieces of code and then putting together this gained knowledge in high-level abstractions.
- **Opportunistic and systematic strategies:** This strategy states that programmers either acquire general understanding of the software by using code-reading, control and data-flow abstractions, or taking an as-needed approach focusing on pieces of code related to a particular task [26].
- **The Integrated Metamodel:** This theory consists of 4 components. The first three components describe the process to create the mental model by using the firsts three approaches (top-down, bottom-up, and opportunistic), and the last one called knowledge base describes the information needed to build the previous models. It is represented by the developer’s current knowledge, which is used to gain new understanding of the software.

1.6 ISPACE

The current section presents ISPACE tool[6], the selected tool to implement the technique under assessment. Since ISPACE has a large number of characteristics, we limit our description to those which were needed to perform our controlled experiment. The work presented in [28, 29] gives a more detailed tool description. Also, we describe briefly the selection criteria.

1.6.1 Basic Concepts and Operations

The main factor being analyzed is the use of the graph-based visualization technique, which in this case is implemented in the ISPACE tool. The selection of the graph-based visualization tool was strongly influenced by the easiness of use and installation, the proven usefulness of the generated visualizations, and the ability to visualize packages which on one hand is the natural decomposition mechanism of a java system, and on the other, is essential for understanding non-trivial programs [30].

This Eclipse plug-in allows the users to explore the structure of a software system, i.e., its components and their dependencies using a nested-labeled graph. Nodes are called container boxes and can contain other nodes. Edges represent relationships between boxes, and the number of dependences is mapped onto the weight of the arrows [28].

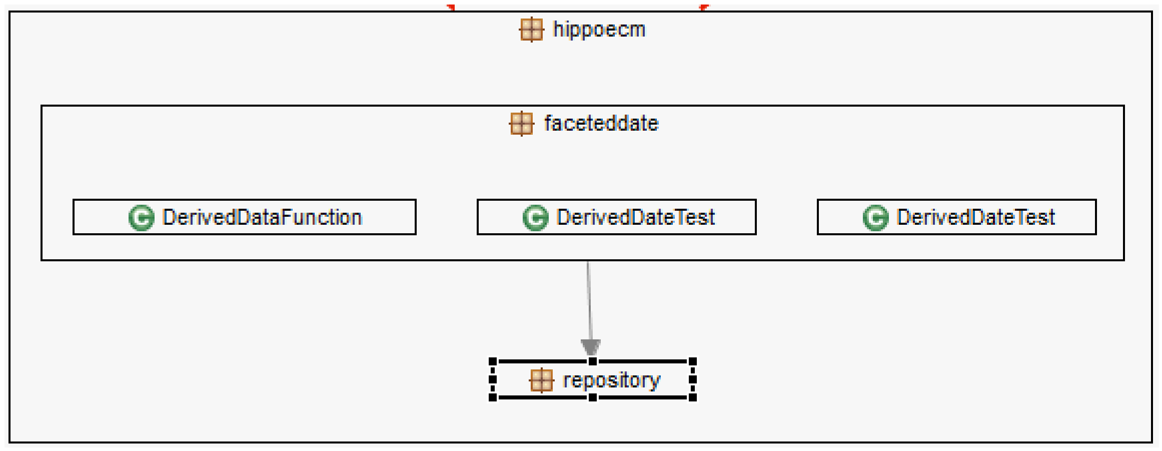
- **Expand/Collapse:** The elements nested in any composed node can be visualize by calling the “expand group” functionality. The inverse operation is collapse. Also, these operations can be performed using double-click. This functionality allowed subjects to explore hierarchy, and to focused in the part of the system of their interest.
- **Hide/Unhide:** This feature permits to hide irrelevant nodes and their edges. This feature allowed subjects to improve their visualizations.
- **Move:** All elements can be moved within the ISPACE layout, even among container boxes.

[6] http://web.archive.org/web/20111201180259/http://ispace.stribor.de/index.php?

- **Nodes and relationships:** ISPACE is able to represents all the sorts of elements of a Java system as a node, from packages to variables and methods. Furthermore, it is capable to describe relationships such as: implements, inherits, and uses.

Figure 1-9 shows a screenshot of the ISPACE user interface. It represents the packages hierarchy hippoecm.faceteddate and its classes, and its relationship with hippoecm.repository package. This hierarchical-recursive organization is the typical structure of object-oriented software [31].

Figure 1-9: ISPACE view.



1.7 Related Work

Although there have been several works on evaluation of visualization techniques, we restrict ourselves to briefly report on controlled experiments aimed at assessing graph-based software visualization tools and approaches used to support program comprehension.

Knodel et al. [32] evaluated the impact of changing the configuration of graphical elements in their tool for software architecture visualization and evaluation SAVE. The experiment compares the influence of two configuration utilizing Tomcat web server as object system. 29 academic subjects performed 10 architecture analyses tasks. Their findings show an improvement of 63% in effectiveness by changing the configuration of graphical elements.

Cornelissen et al. [33] carried out an experiment to assess EXTRAVIS, their tool for visualizing execution traces. The experiment compares EXTRAVIS + Eclipse IDE against Eclipse IDE. The results show a time decrease of 22% in time, and an increase of 43% in correctness. They used CHECKSTYLE as object system, and a group of 23 academic subjects and one participant from industry.

Storey et al. [11] evaluated the two different views approaches in their tool Rigi, namely Multi-Win and SHriMP, and vi/grep Unix command line tool. 3 different programs written in C were used as objects system. 12 subjects with academic background participated in the controlled experiment. Quantitative results of this experiment is not reported.

Haitzer and Zdun [34] conducted a controlled experiment to determine the usefulness of component diagrams to support understanding architectural level using Freecol computer game as object, and 60 students as subjects. The results indicate that architectural component diagram are useful to understand architectural connections that are hard to see by exploring code.

Quante [35] performed a controlled experiment to evaluate whether their approach DOPG (Dynamic Object Process Graphs) is useful to support program understanding. The controlled experiment was carried out with 25 participants, all of them students, and three object systems, namely Jetris, GanttProject, and ArgoUML. They found that the usefulness of the approach depends of the system, it was only beneficial for ArgoUML, the biggest system.

Finally, Fittkau et al. [36] conducted two controlled experiments; the primary study and its replication in order to compare their tools EXRAVIS and ExploreViz in typical comprehension tasks. They performed their assessment by using Babsi and PMD as object systems. In the experiment 30 students participated, and 50 in its replication. The findings show that subjects spent similar time for the small-sized system (Babsi), and a time reduction of 28% for the large-sized (PMD) system in favor of ExplorViz. Also, results show a significant improvement for both small and large-sized systems by using ExploreViz; 39 and 61% respectively.

There are two major differences between the related work and our study. First at all, most of the studies evaluate their own tools or approaches, while our study is completely independent. The assessment presented by Haitzer and Zdun [34] is the only exception, but they used predefined diagrams in their controlled experiment. In our approach we provided subjects with a tool, which allows them to interact directly with the object system.

The second principal difference is the system object's size. We decided to utilize a software system of 303828 LOC. The experiments cited in this section, mainly present software systems not representatives of actual industrial systems. Only in the Quante's experiment [35] the largest system has 319917 LOC, which we consider a reasonable and representative size.

2. Experimental Design

Research Design is the process of selecting a method for a particular research problem, tapping into its strengths, while mitigating its weaknesses [18]. The purpose of the experiment is to provide a quantitative evaluation of the effectiveness and efficiency of the graph-based visualization technique when compared to a common code-based exploration technique. Throughout this chapter the complete experiment's design is explained in detail.

2.1 Research Questions

The following research questions were formulated:

RQ1: Does the use of graph-based VT increase the correctness, when performing software understanding tasks?

RQ2: Does the use of graph-based VT reduce the completion time, when performing software understanding tasks?

Thus, there are only two treatments in our experiment, one that provides subjects with a graph-based visualization tool for performing the tasks (that is, ISPACE), and another that provides subjects only with the Eclipse IDE. A between-subjects design was used, so that each participant was tested on only one treatment.

2.2 Hypotheses

Based on the formulated research questions, the null and alternative hypotheses are given in the table 2-1.

Table 2-1: Hypotheses

Null Hypothesis	Alternative Hypothesis
H10: There are no significant differences in correctness between both techniques for software understanding tasks.	**H1**: There are significant differences in correctness between both techniques for software understanding tasks.
H20: There are no significant differences in completion time between both techniques for software understanding tasks.	**H2**: There are significant differences in completion time between both techniques for software understanding tasks.

2.3 Object System

A real system was chosen for our assessment. This selection was made based on two criteria: 1) it has to be as large as typical industrial software, so that it is possible to extend and generalize the results, and 2) its application domain has to be familiar to the subjects, in order to prevent unnecessary confusion among participants. The selected system was Hippo CMS[7], a friendly and popular Content Management System written in Java and used by a variety and well-known range of clients such as: Autodesk, ACM, and the University of Amsterdam among many others. It has 145 packages, 928 classes, and 303828 lines of code.

2.4 Tasks

For choosing experimental tasks, three strategies were adopted. The first one was to survey practitioners for identifying activities they consider important in the software understanding process in industry; the second one was to look for typical tasks in previous and related studies [1, 11, 32, 33, 36, 37, 38, 39, 40]; and the last one was to make sure that each task requires a reasonable amount of time to be completed, yet the set of tasks does not require more than 120 minutes.

[7] https://www.onehippo.org/

Eight tasks were initially chosen, but it was necessary to exclude two original tasks. One task was redundant because it was based on names recognition and coupling analysis which we consider covered by tasks 3, 4 and 6. The second task was dismissed since it was biased towards a particular architectural style, so we would not able to generalize results.

Finally, six tasks[8] were selected in order to cover the most important and common software understanding activities [40]. It was decided on the final set of tasks, described below, after a pilot was ran with two subjects to determine the approximate amount of time needed to complete the tasks, and to remove ambiguities in the writing of questions.

Task 1. From a structural viewpoint, what is the most important package or set of packages in the system? How does it interact with the others? How did you identify it?

Rationale: Assessing high-level structure/architecture of the software system and how its components interact is a key comprehension activity to understand the domain of the system [40].

Task 2. Describe the class structure of the package P. That is, relationships among entities. How did you identify that structure?

Rationale: Investigating the internal structure of an artefact is a typical comprehension task [40].

Task 3. Which is the class in the package P with the strongest coupling to package Q? How did you identify that class? [33]

Rationale: Coupling and cohesion are two of the most important design concepts and help to determine how the system works and how easy it is to maintain and evolve.

[8] Complete questionnaire is available in the experimental package.

Task 4. Which is the class in the package P with the highest fan-in (incoming calls)? Which is the class with the lowest fan-out (outgoing calls)? How did you identify them? [36]
Rationale: Understanding dependencies between artefacts and assessing the quality of the system's design is one of the principal activities in software comprehension [40].

Task 5. Look for the class C in the package P. Evaluate its change impact considering its caller classes. The assessment is done in terms of both intensity (number of potentially affected classes) and dispersion (how these classes are distributed in the packages structure). How did you do this task?

Rationale: Impact analysis provides the means to estimate how a change to a restricted part of the system would impact the rest of the system. Although extensively used in maintenance activities, impact analysis may also be performed by developers when estimating the effort needed to perform a change. It also gives an idea of the quality of the system. A part of the system which requires a large effort to change may be a good candidate for refactoring [1].

Task 6. Describe the purpose of package P. How did you determine the purpose of that package? [11]

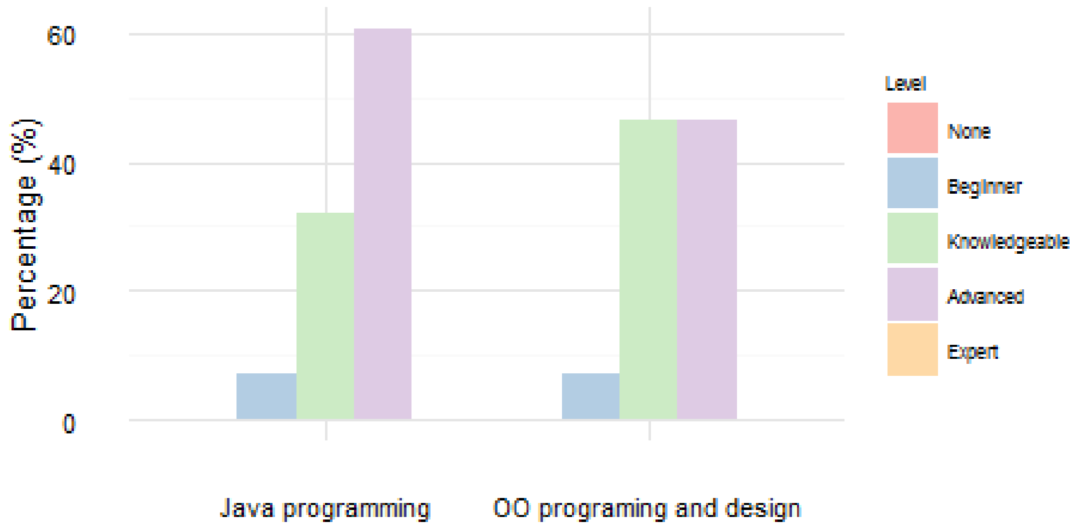
Rationale: Investigating the functionality of (a part of) the system and understanding its domain is one of the main and useful activities in software comprehension for practitioners and researchers [40].

2.5 Subjects

Subjects in this controlled experiment were undergraduate students from a Software Engineering course. As we see in Figure 2-1, all subjects had knowledge of the fundamentals of OO programing and design, and Java programing experience, which it was estimated as an adequate background for performing the experiment.

Since motivation is an essential element of software visualization evaluations [41], to recruit participants it was offered a reward and 21 students from the course decided to accept the invitation. However, it was necessary to discard the responses of one of the participants, since the participant did the tasks in just a few minutes, which indicated lack of serious effort.

Figure 2-1: Subjects' Expertise.



To assign the two treatments, participants were divided into two groups, maintaining a balance between the groups with respect to course performance as much as possible. To do that, the students were grouped in three categories namely, A, B, and C, according to their grades and the quality of their contributions to the course software project. After that, the students of each category were evenly distributed among the two groups using the blocking technique (Table 2-2).

Table 2-2: Subjects' Distribution.

Number of Subjects	Treatments		
Blocks	Graph-based	Source-code-based	Total
C	6	7	13
B	1	2	3
A	3	1	5

Total	10	10	20

2.6 Data collection

All subjects answered the six tasks via an online questionnaire designed using the Qualtrics[9] system. The tasks were presented in the order they appear in Section 2.4. Each task was timed and the subjects were asked to write the answer in the space provided in the online forms. They were not allowed to go back to questions already answered or skip a question without answered it.

2.7 Independent variables

Since the current experiment has the purpose of measuring the effectiveness and the efficiency of graph-based VT at supporting software understanding tasks, we consider the type of technique our independent variable. This variable has two levels, i.e., graph-based visualization and source-code-based exploration technique.

2.8 Dependent variables

We assessed the correctness and the completion time of each task to measure effectiveness and efficiency, respectively. The correctness of each answer was scored by one of the authors with a number between 0 and 10 (See grading scale in the experimental package). All answers were previously mixed, so the grader did not know which treatment was rating. The time spent by a subject on answering a task was recorded by Qualtrics, the online survey tool used.

2.9 Controlled variables

We identified the participants' course performance as an influential factor over the experiment's results. To mitigate this potential influence, we used blocking technique based

[9] https://www.qualtrics.com/

on the criteria and categories described in Section 2.5. Table 2-3 shows the assignment of individuals to groups, broken down by categories.

Table 2-3: Block design

Treatments	**A**	**B**	**C**	**Total**
Source Code-based group	2	2	7	11
Graph-based group	3	1	6	10

2.10 Study procedure and instrumentation

We ran a pilot study that allowed us to verify the feasibility of the tasks, calculate the approximate time required by each task, and improve the wording and clarity of the questions and instructions for the participants. It was performed by two students, each one of them resolved tasks for one treatment.

Before the study, one of the authors made a brief presentation of ISPACE to the entire course. At the same meeting, the presenter did a review of the main features provided by Eclipse for exploring the structure of a system. In addition, when starting the experiment, the ISPACE group was given a one-page description of the visualization tool and a brief tutorial of the most relevant Eclipse functionalities for performing the tasks of the study; and the other group was provided only with the Eclipse tutorial.

The study was conducted in a laboratory where each participant used a laptop computer previously set up with the software required. Participants in the ISPACE group were allowed to use any Eclipse feature they thought are essential for doing the task at hand, in addition to the ISPACE plug-in. At the end of the study, they filled out a short post-questionnaire. The complete set of study materials is available for reviewing and replication purposes.

2.11 Threats to Validity

2.11.1 External Validity

External validity refers to the degree to which the results of our study can be generalized to other populations and contexts. First of all, the subjects were undergraduate students with acceptable knowledge of Java and Eclipse, but with no knowledge of the software architecture subject. It is likely that using graduate students or industry developers, the results may have been different. Secondly, the six tasks chosen refer to a single medium sized system. Thus, the representativeness of the tasks and subject system selected is another threat that reduces our ability to extrapolate our results to other Java systems and other types of comprehension tasks.

2.11.2 Internal Validity

Internal validity refers to unrelated factors that may compete with the independent variable in explaining the study results. First of all, to reduce the threat that the subjects may not be competent enough to perform the tasks proposed, we chose them from a software engineering course, ensuring that participants had basic knowledge of Java programming, and OO programming. In addition, all subjects attended a brief presentation about the main functionality of ISPACE and the Eclipse IDE, and also received short tutorial of both tools. Secondly, based on the performance of the students in the course, we grouped the subjects such that both groups would have participants with fairly similar programming skills and software engineering knowledge. In this way we mitigated the threat of an unbalanced distribution of the subjects’ expertise across the two groups. Third, since we are conducting an independent assessment, the choice of the tasks was not biased toward any technique, and participants in the ISPACE group were allowed to use any Eclipse feature they thought are essential for doing the task at hand, in addition to the ISPACE plug-in. Finally, the participants were recruited on a voluntary basis, all of them received a reward simply for participation, they were assured of the anonymity of their answers, they did not know neither the study goal, nor which group they were before performing the study.

3. Results

We used the two-tailed Student's t-test for our analysis as it is the most suitable for our experimental design. This test requires our data meet normal distribution and depend on equal or unequal variances. To test normal distribution, Shapiro-Wilk test was used and to test homogeneity of the variances, Levene's test was conducted. Both test succeeded assuming a significant level of 0.05 (α=0.05). The complete statistics related to the experiment's result are presented in Table 3-1.

Table 3-1: Descriptive Statistics of the Experimental Results.

Correctness										
Treatment	*Mean*	*Min*	*Max*	*Median*	*Stdev*	*Shapir-Wilk Test*	*Cohen's d*	*Levene's Test*	*t*	*p-value*
Code-based	5.78	4.17	7.08	6.21	0.98	0.1934	1.0034	0.4782	-2.2587	0.0366
Graph-based	7.01	4.87	9.58	7.08	1.43	0.8272				
Completion Time										
Treatment	*Mean*	*Min*	*Max*	*Median*	*Stdev*	*Shapir-Wilk Test*	*Cohen's d*	*Levene's Test*	*t*	*p-value*
Code-based	40.00	20.57	58.48	38.73	12.00	0.8241	-	0.7236	0.0602	0.9530
Graph-based	39.64	28.11	60.83	37.83	10.60	0.1847				

We performed the analysis for correctness and completion time using RStudio[10] Statistical Software. All scripts and results are available as part of our experimental package.

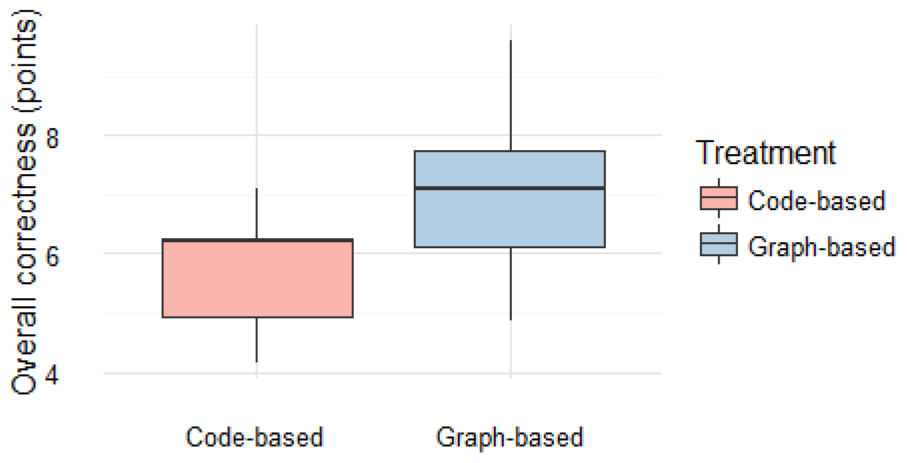
3.1 Analysis Overall Results on Correctness

Student's t-test revealed a p-value 0.0366. In consequence, we reject H10 in favor of the alternative hypothesis H1 indicating that the mean correctness in the graph-based treatment was significantly higher than the one for code-based treatment. This means that the data show evidence that the graph-based visualization increases the correctness in program comprehension tasks (21.45%).

[10] https://www.rstudio.com/

To understand better de magnitude of the difference between the two treatments [42, 43], we decided to compute the effect size using Cohen's d measure. The found value was 1.0034, which means a large effect due to the support of the graph-based visualization technique. The effect of the technique on correctness is illustrated in Figure 3-1.

Figure 3-1: Boxplots for overall correctness.

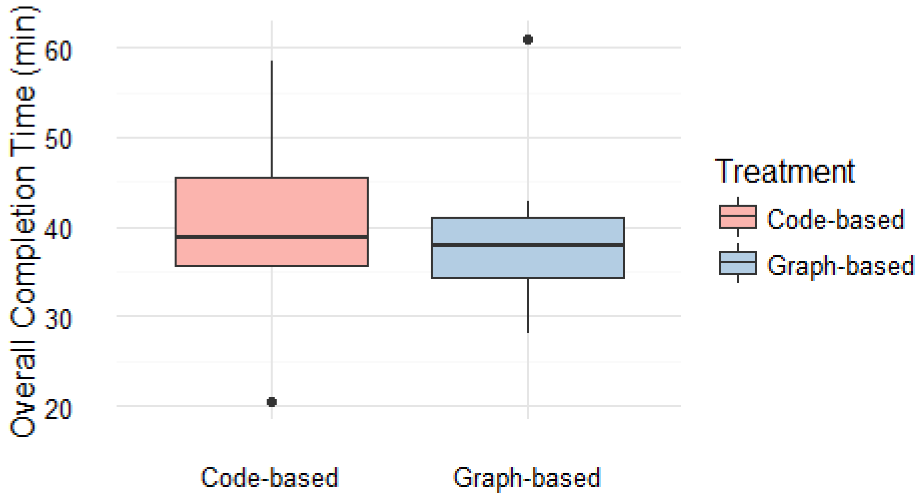


3.2 Analysis Overall Results on Completion Time

For the completion time analysis, we decided to remove results from both task T1 and T4 due to their low correctness, especially in the code-based treatment, which makes impossible the comparison. Even more, we decided to perform an extra grading including only results from the remaining tasks (T2, T3, T5, and T6), and removed all subjects with a total score less than 7.5 in the same way that the evaluation performed by Fittkau et al. [36]. It ended up in elimination of 3 subjects per treatment.

Student's t-test revealed a p-value 0.9530. Thus, there is no evidence to rejected H20 in favor of the alternative hypothesis H2 indicating that the difference between the mean completion time in both graph-based and code-based treatments was not significant. This means that the data show no evidence that there are significant differences in completion time between both techniques for software understanding tasks. The effect of the technique on completion time is illustrated in Figure 3-2.

Figure 3-2: Boxplots for overall completion time.



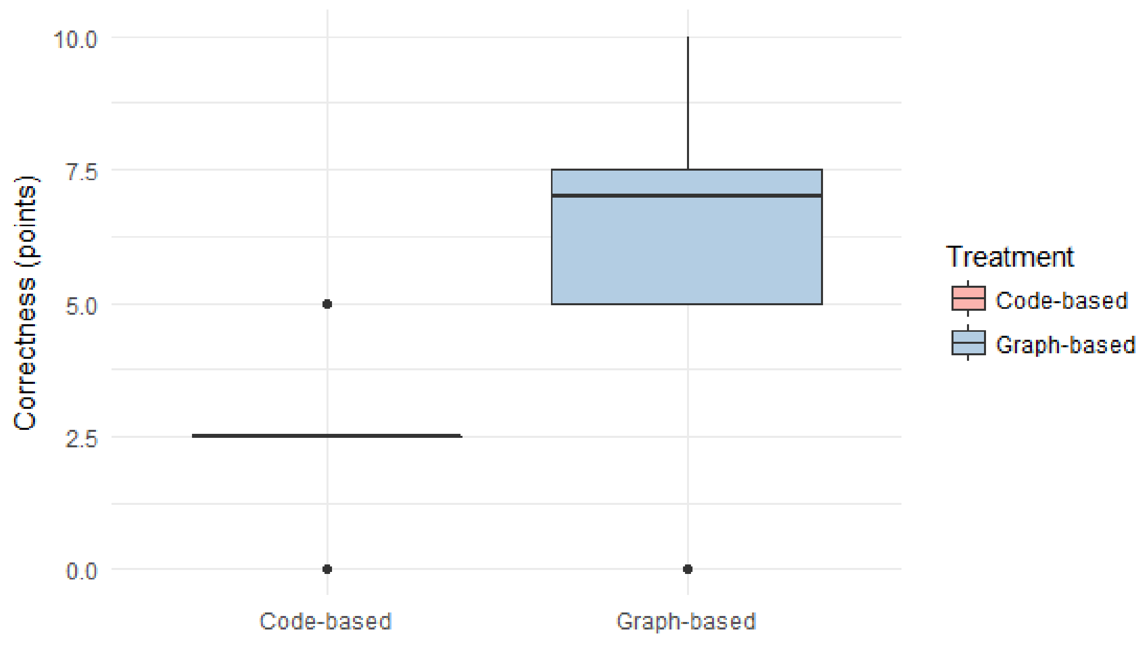
3.3 Tasks Analysis

We provide a deeper analysis of the results, for each task. Both task T1 and T4 was excluded from completion time analysis, as the same way as results below 7.5 for remaining tasks (T2, T3, T5, and T6). All results were taken into account for the correctness analysis.

3.3.1 Task 1

The graph-based group performed better than the code-based group in terms of correctness (6.15 vs. 2.50). Subjects in code-based treatment had a very poor performance, while for the graph-based one it was acceptable. The subject who had an acceptable performance used strategies such as look for hierarchical relations and amount of classes per package. That was the approach used by most of the participants in graph-based treatment, which suggests that the VT leads participants through a better strategy. The effect of the technique on correctness is shown in Figure 3-3.

Figure 3-3: Boxplots for Task 1 correctness.



3.3.2 Task 2

The graph-based group performed better than code-based in terms of correctness (8.19 vs. 5.90). Description of entities and relationships was more detailed in the graph-based treatment than in the code-based one, which suggests that VT gives an extra degree of accurateness. There is a difference in terms of completion time in favor of the graph-based treatment (16.63 vs. 18.43). The effect of the technique on correctness and completion time is shown in Figure 3-4 and 3-5 respectively.

Figure 3-4: Boxplots for Task 2 correctness.

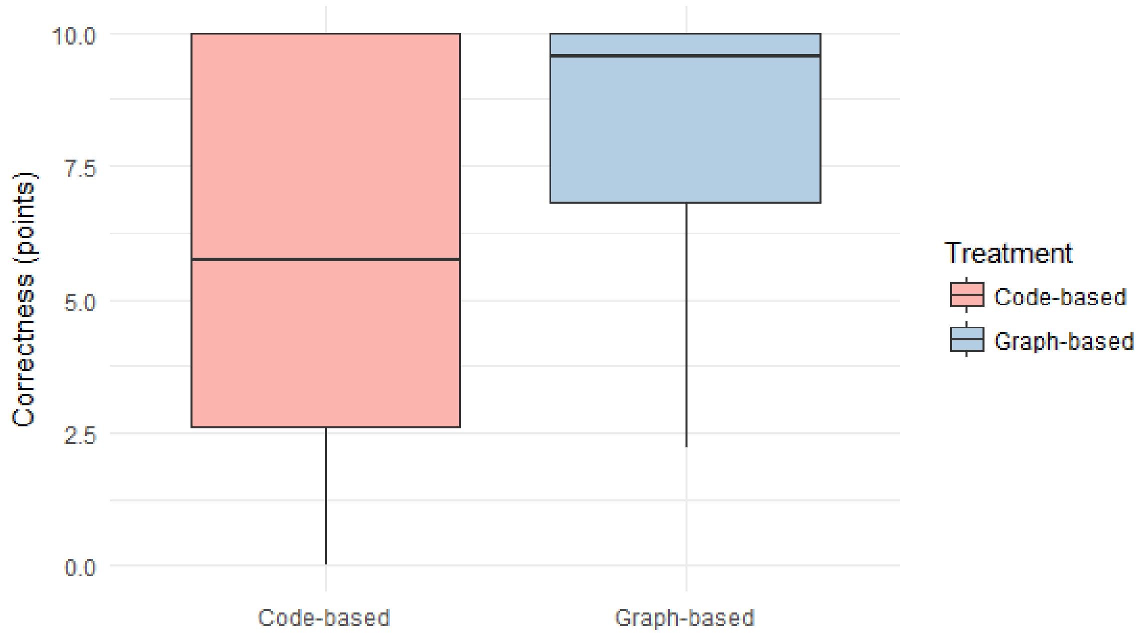
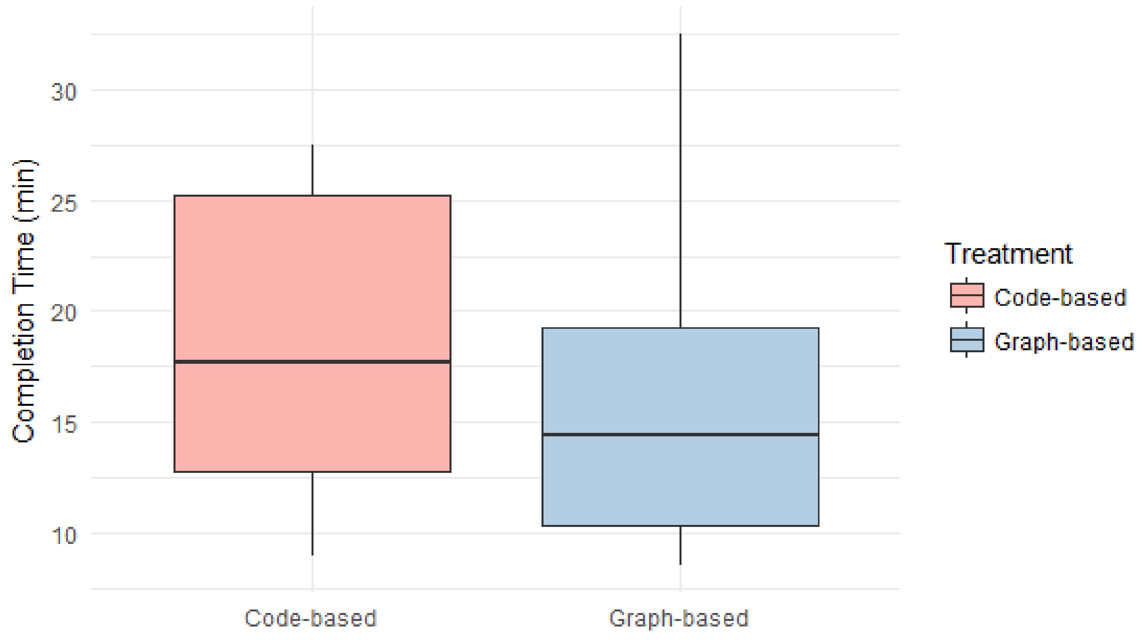


Figure 3-5: Boxplots for Task 2 completion time.



3.3.3 Task 3

The code-based group performed better than graph-based group in terms of correctness (10.00 vs. 8.00). Code-based exploration seems to be the best strategy here; even participants in graph-based treatment used it to carry out this task. There is a slight difference in favor of graph-based technique in completion time (7.27 vs. 8.25). The effect of the technique on correctness and completion time is shown in Figure 3-6 and 3-7 respectively.

Figure 3-6: Boxplots for Task 3 correctness.

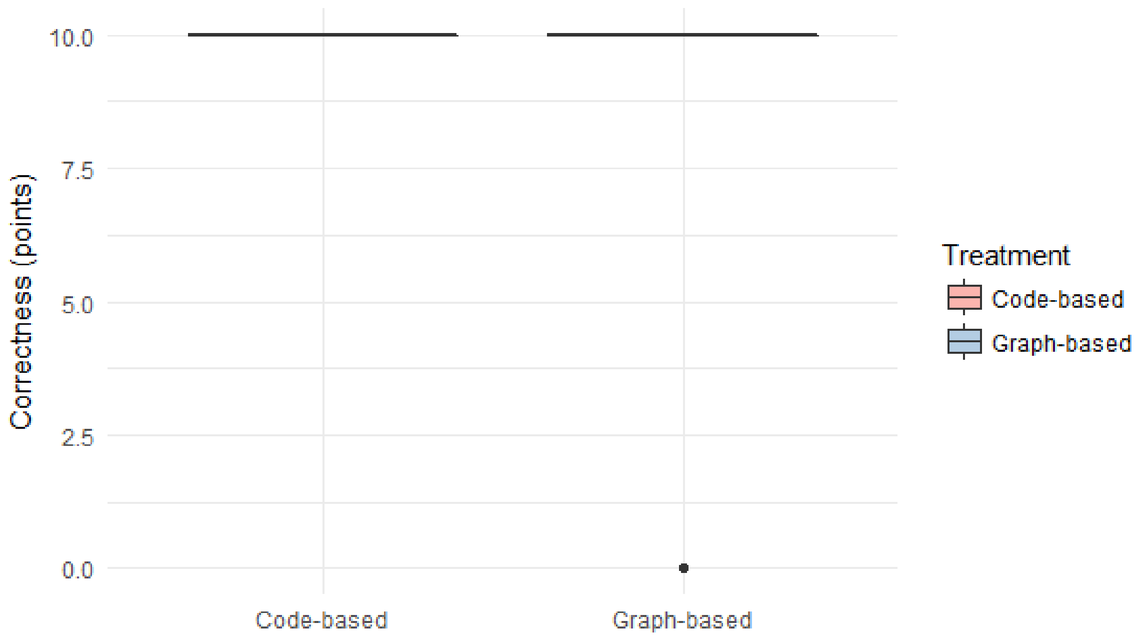
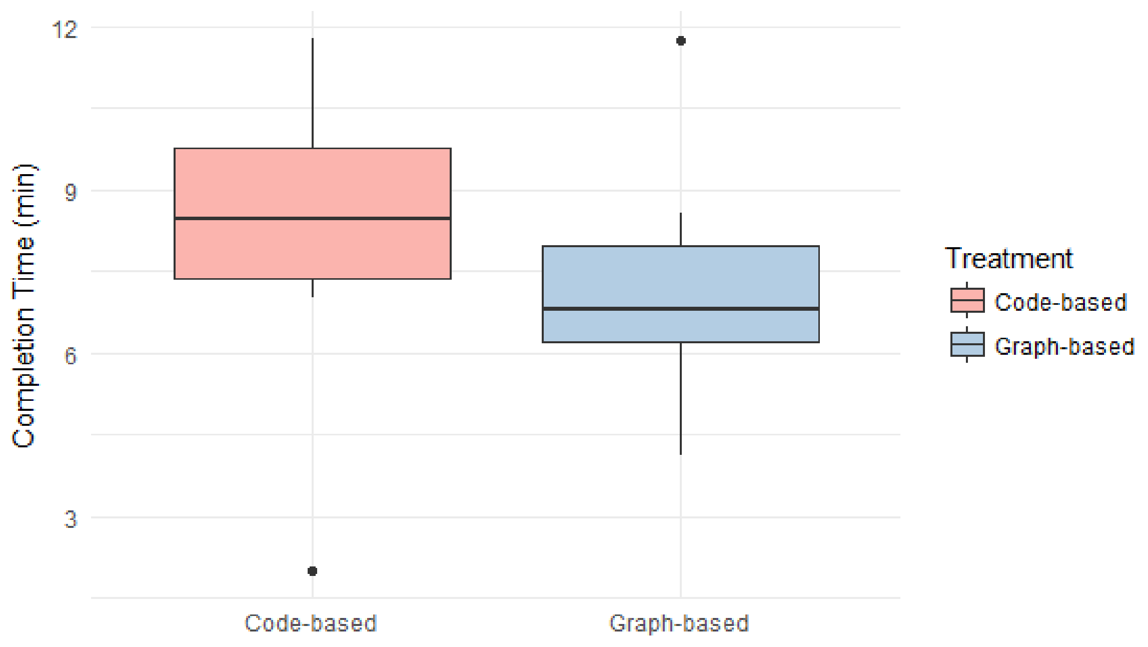


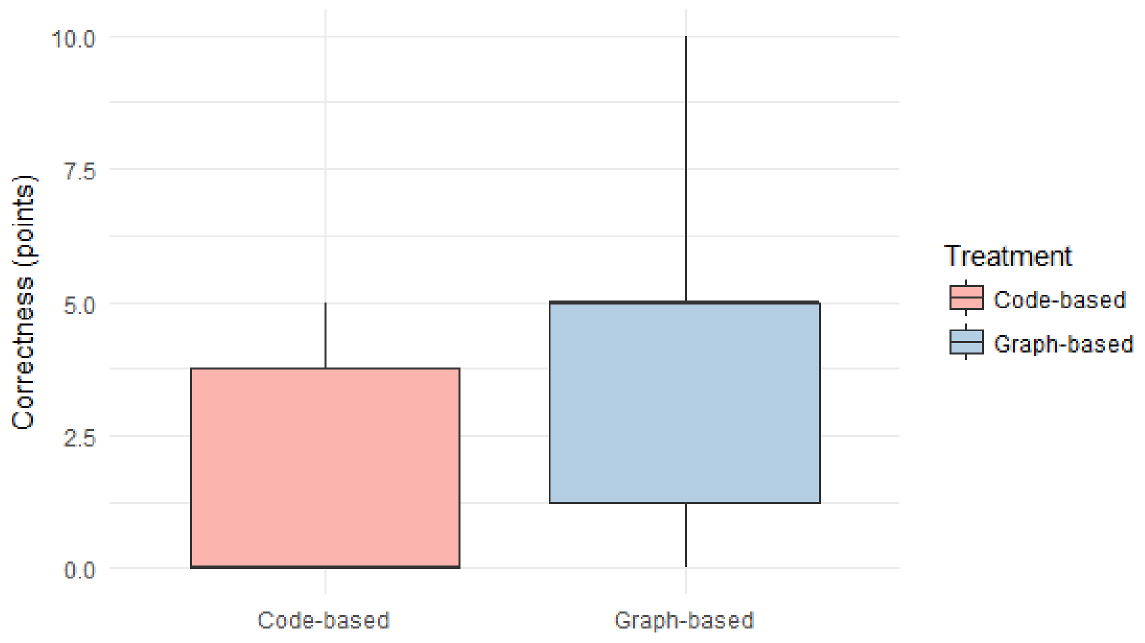
Figure 3-7: Boxplots for Task 3 completion time.



3.3.4 Task 4

The graph-based group performed better than code-based group in terms of correctness (4.50 vs. 1.50). Exploring calls via metrics (number of dependencies) and caller classes was the strategy used by participants with perfect score in the graph-based treatment. In the code-based treatment no one has a perfect score, and seven out of the eleven subjects scored 0 in this task. This suggests that the use of graph-based conventions substantially improves correctness in this task. The effect of the technique on correctness is shown in Figure 3-8.

Figure 3-8: Boxplots for Task 4 correctness.



3.3.5 Task 5

For this task the graph-based performed better than code-based in terms of correctness (5.25 vs. 4.75). However, both groups had a low performance. Evidence suggests similar results, and similar strategies (searching relationships). On the other hand, the graph-based performed better than code-based in terms of completion time (9.25 vs. 10.34). The effect of the technique on correctness and completion time is shown in Figure 3-9 and 3-10 respectively.

Figure 3-9: Boxplots for Task 5 correctness.

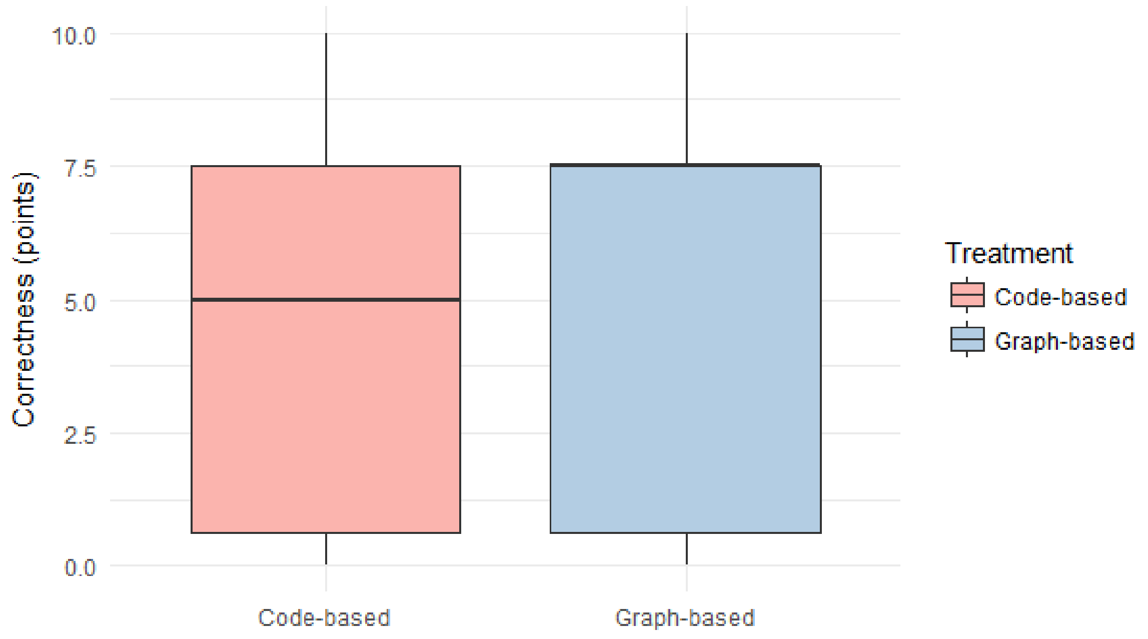
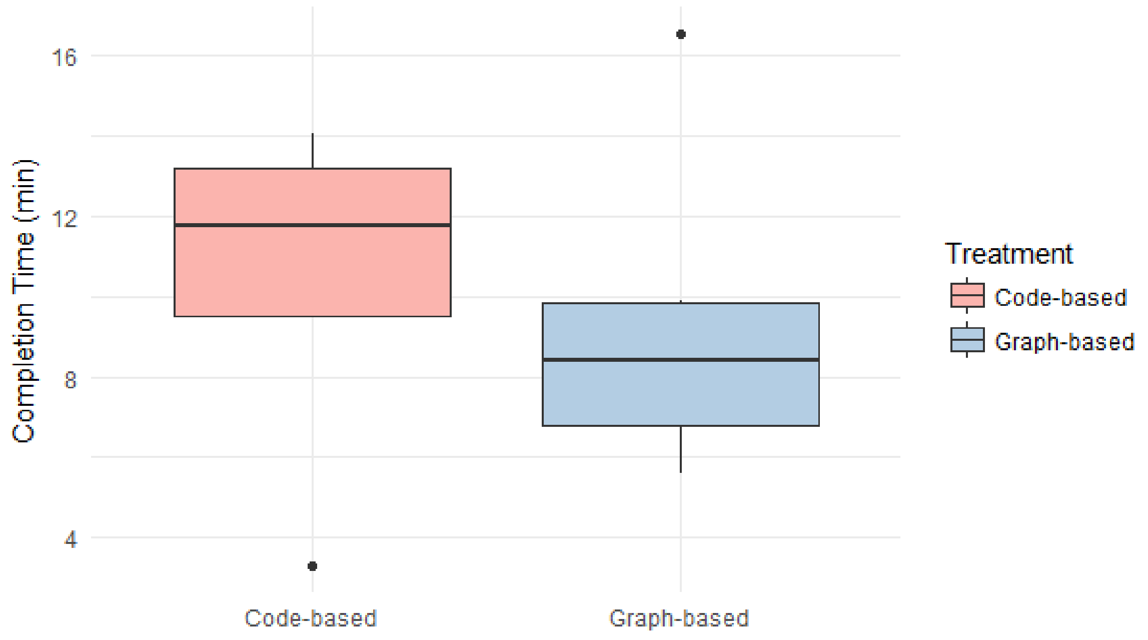


Figure 3-10: Boxplots for Task 5 completion time.



3.3.6 Task 6

No significant differences between both techniques in correctness were found (10.00 vs. 10.00). Evidence suggests that VT leads subjects to use mostly name recognition strategy, while in code-based treatment most subjects inspected code to understand package's functionality, and just one subject used name recognition strategy to solve this task. The code-based group performed better than graph-based in terms of completion time (4.91 vs. 5.66). The effect of the technique on correctness and completion time is shown in Figure 3-11 and 3-12 respectively.

Figure 3-11: Boxplots for Task 6 correctness.

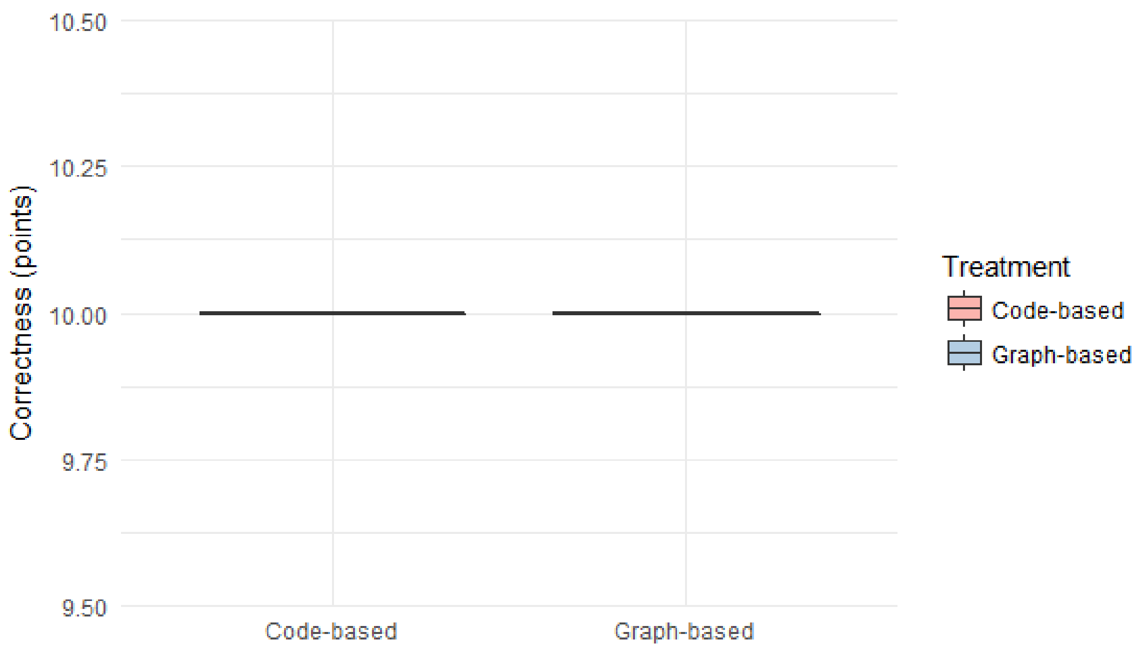
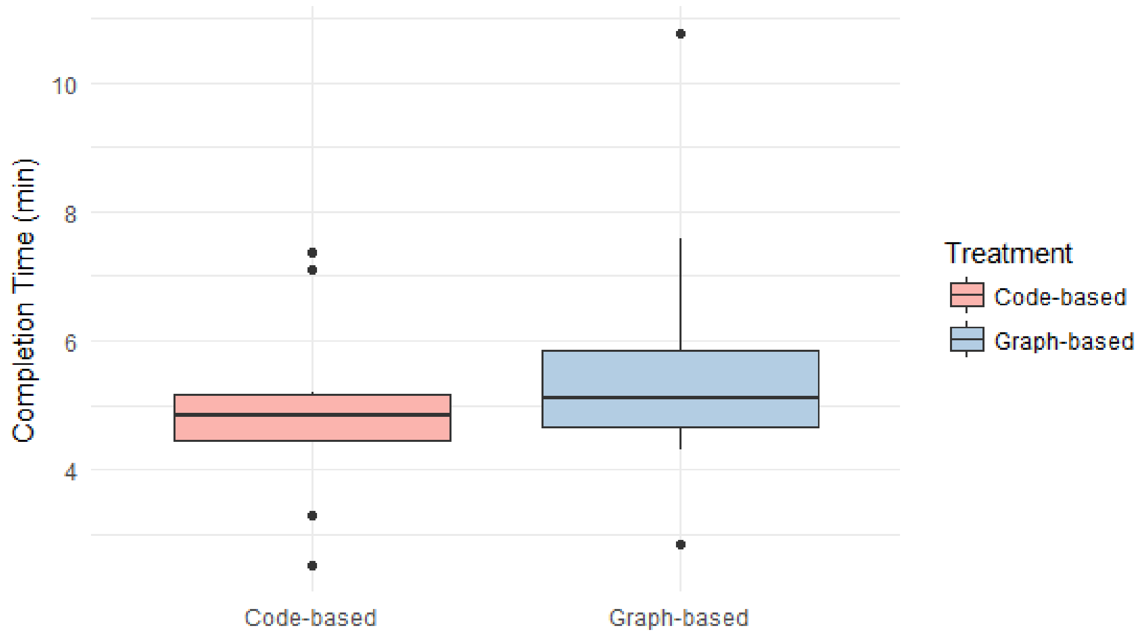


Figure 3-12: Boxplots for Task 6 completion time.



In summary, subjects in the graph-based group performed better in tasks T1, T2, T4, and T5 in terms of correctness, with the largest differences in tasks T1 and T4. For task T6, final results in correctness were exactly the same for both techniques. Task T3 was the only one in which the code-based group overcame the graph-based one. Figures 3-13 and 3-14 show graphical results of correctness.

Figure 3-13: Correctness per task.

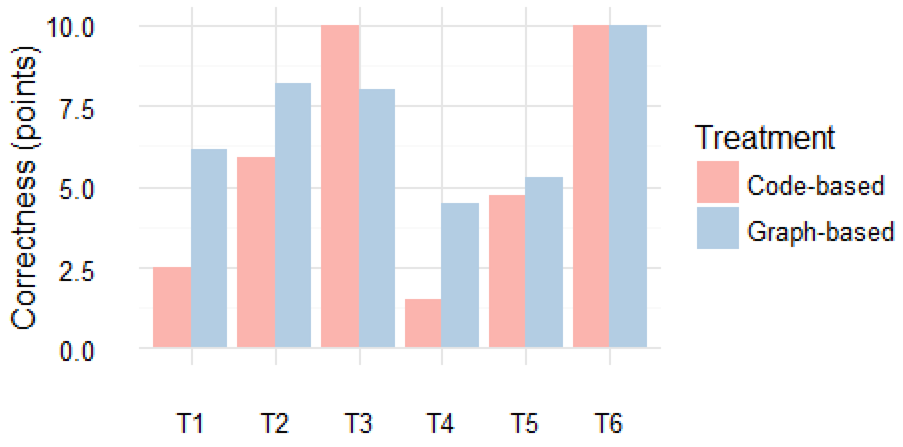
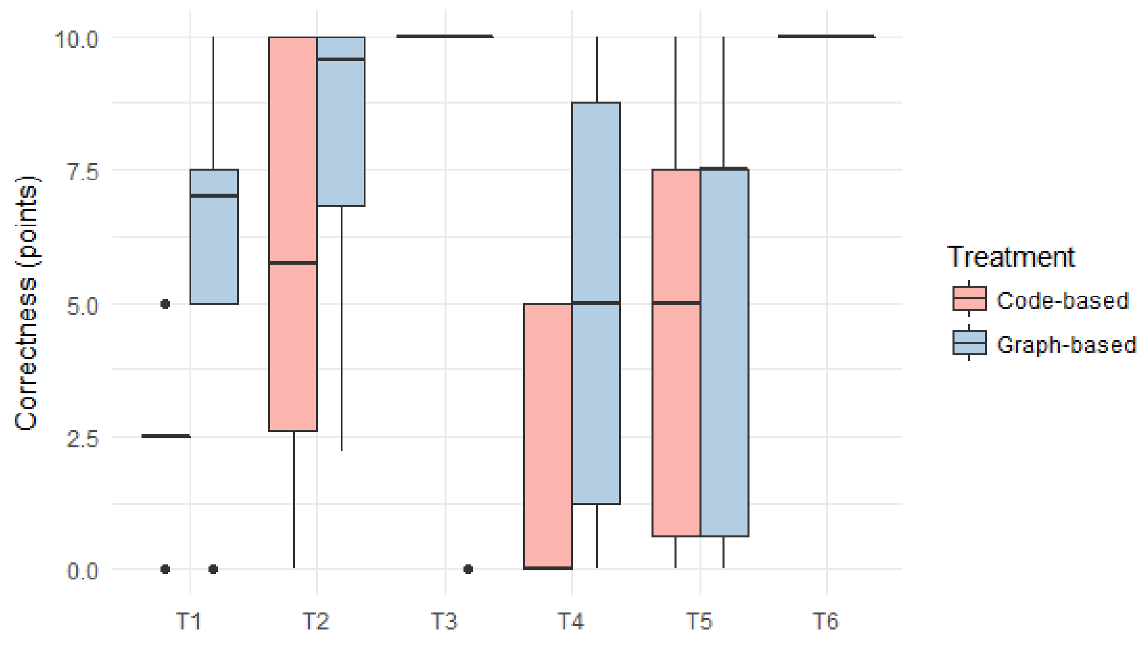


Figure 3-14: Boxplot correctness per task.



Subjects in graph-based treatment performed better in tasks T2, T3, and T5 in terms of completion time. The only one task in which code-based technique overcame graph-based technique was T6; however, the difference was not too large. Figures 3-15 and 3-16 show graphical results of completion time.

Figure 3-15: Completion time per task.

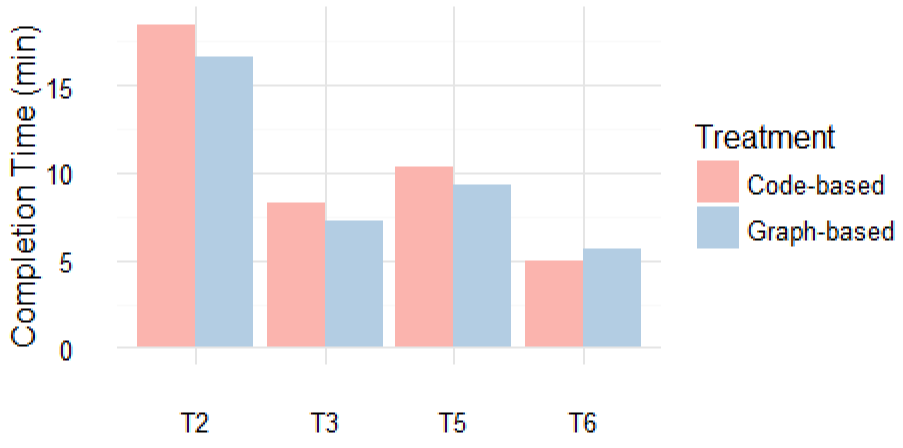
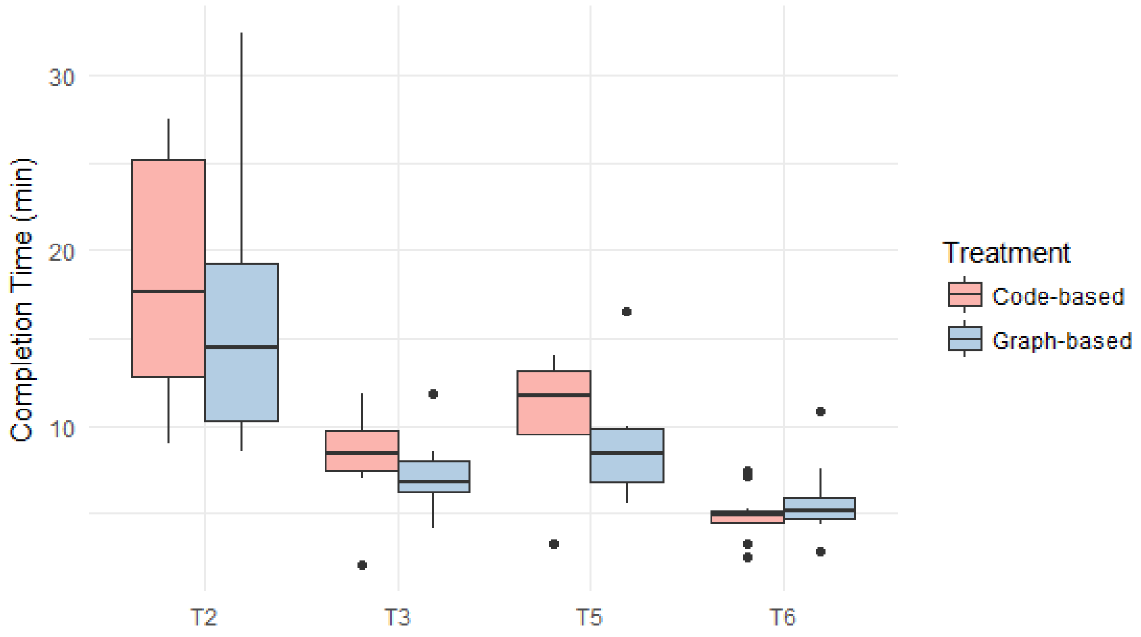


Figure 3-16: Boxplot completion time per task.



3.4 Discussion and comparison of unsuccessful tasks

Both techniques performed low in tasks T4 and T5. Task 4 was used in a study by Cornelissen et al. [33] and task 5 in a study by Wettel et al. [33]. Results in the mentioned studies outperformed considerably ours for both tasks.

The experiment done by Cornelissen et al. [33] evaluates the EXTRAVIS tool, which visualizes dynamic and static analysis against Eclipse. EXTRAVIS provide a two linked views; the massive sequence view to support trace analysis, and the circular bundle view, which is representative of the graph-based technique [2] to perform static analysis. The subjects using EXTRAVIS performed considerably well in this task, in terms of correctness and completion time. Task 5 was taken from a controlled experiment conducted by Wettel et al. [1]. They evaluate the CodeCity tool, which presents a metaphor-based technique [2]. The study was performed over two different software systems against a base-line (Eclipse + Excel), having good results too.

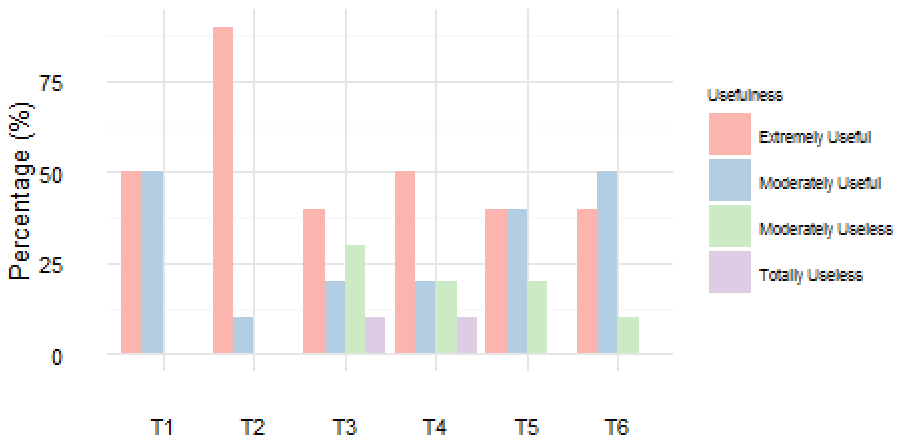
Hence, the low performance in our experiment could be caused by several reasons such as: subjects' background, unsuitability of the graph-based technique for carrying out this particular type of tasks, misunderstanding of the tasks, etc. We asked participants about tasks understandability, and only two claimed not to understand task T4. No subject said not to understand task T5. In order to analyze the others mentioned potential reasons, further research is necessary.

3.5 Usefulness Perception

We asked subjects about how useful was the support provided by the visualization technique to solve each task. The graph-based visualization technique was rated as Extremely Useful and Moderately Useful for most participants in all tasks, especially in tasks T1 (100%), T2 (100%), and T6 (90%). For Task T3 (60%), T4 (70%), and T5 (80%) the visualization technique was less useful.

Subjects perceived the visualization technique more useless for Task 3, where 30% of the subjects perceived it as moderately useless, and 10% as totally useless. This is consistent with their strategy to resolve this specific task, since 5 subjects used code exploration when solved it. The most useful support was perceived in Task 2, where 90% of the subjects rated the visualization technique as extremely useful, and the remaining 10% as moderately useful. The strategy here was to explore visual conventions. Figure 3-17 shows the overall subjects' perceptions of the graph-based group.

Figure 3-17: Usefulness perception of the Visualization Technique.

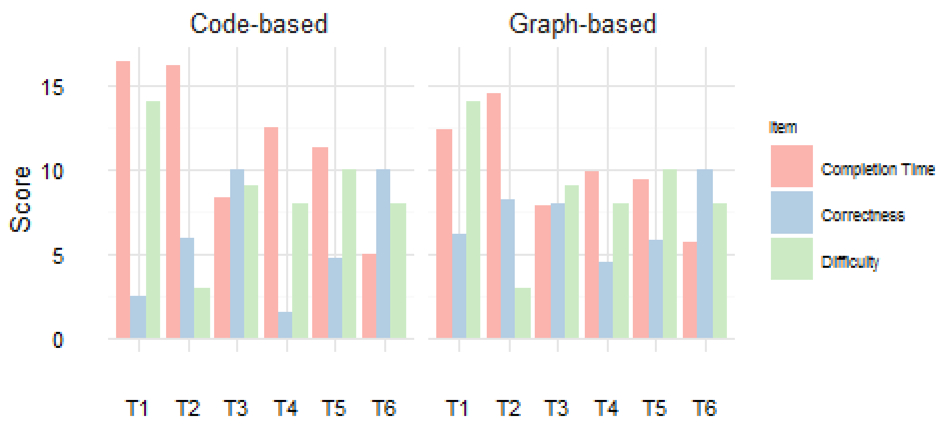


3.6 Post-Experiment Analysis

After the experimental session we surveyed subjects about the tasks difficulty; 19 participants answered the questionnaire. They were asked to select the 3 most difficult tasks. The results show that tasks T1 was considered the most difficult task by participants, which corresponds with their low scores. Task T2 was considered the less difficult which is consistent with the results. Tasks T3, T4, T5 and T6 had an average and similar perception, but different results in correctness and completion time. Tasks T3 and T6 had excellent results, whereas participants' performance was low for tasks T4 and T5. It could be due to participants' confidence in their answers. Unfortunately, we have no evidence to state our hypothesis as we did not ask participants about their level of confidence.

Figure 3-18 compares the difficulty perceptions against both completion time and correctness results.

Figure 3-18: Task difficulty perception.



4. Conclusions and Future Work

4.1 Conclusions

We conducted a controlled experiment in order to assess the impact of using the graph-based visualization technique, when performing typical software understanding tasks. Six of these typical tasks were performed by 20 undergraduate students enrolled in a software engineering course. Half of them used the ISPACE Eclipse plug-in, a graph-based tool for visualizing and analyzing Java dependency graphs; and the other half used only the code exploration features offered by the Eclipse IDE. The subject system was Hippo CMS, a friendly and popular Content Management System written in Java. To assess the influence of the visualization technique, we measured the completion time and the accuracy of each one of the participants' responses.

The study results indicate that the use of the graph-based visualization increases the overall correctness (21.45%) and show no statistical evidence of reduction for overall completion time in program comprehension tasks. Therefore, results show benefits of the graph-based visualization technique on improving the effectiveness to carry out typical software comprehension tasks, and suggest no effect on improving the efficiency.

Furthermore, subjects in the graph-based group performed better in tasks T1, T2, T4, and T5 in terms of correctness. Task 6 had the perfect score for both treatments. Task T3 was the only one in which the code-based group outperformed the graph-based one, the code-based exploration seems to be the best strategy for performing this type of tasks.

Although results in overall completion time show no evidence favoring neither the graph-based visualization technique nor the code-based exploration technique, it is possible to see slight differences per task. On one hand subjects in the graph-based group performed better in tasks T2, T3, and T5, but on the other hand subjects in the code-based group performed better in Task 6. Tasks T1 and T4 were omitted form the analysis on completion time since the poor results on correctness make the analysis unreliable.

The visualization technique had the worst perception in Task 3, where 30% of the subjects perceived it as moderately useless, and 10% as totally useless. This is consistent with their strategy to resolve this specific task, since 5 subjects used code-based exploration technique to solve it. Otherwise the more favorable perception was obtained in Task 2, where 90% of the subjects rated the visualization technique as extremely useful, and the remaining 10% as moderately useful. This perception lays in the strategy taken by the subjects which consisted of exploring visual conventions.

Post-experiment survey shows that *Identifying the most important package(s) in the system* was seen as the most difficult task by the 74% of the surveyed subjects. This fact is consistent with the low scores, especially on the code-based treatment. This post-experiment result plus the results on usefulness perception, where all participants rated the visualization as useful, shows the poorness usability of the code-based exploration technique for supporting high-level understanding. 53% of the subjects found *Describing the class structure of a package* as the easiest kind of task. The excellent results of the graph-based group in that task (T2), and the favorable perception point out the usefulness of the visualization for understanding internal structure of artifacts.

4.2 Future Work

We plan on replicating this study with industrial subjects and organizing subsequent similar studies to assess the influence of other visualization techniques on typical software comprehension tasks. Another direction is to characterize the types of tasks in which each of the four visualization techniques [4] is more appropriate and effective, and similarly, identify those in which the visualization support does not improve neither the efficiency nor the effectiveness.

Since replications play important roles in the construction of knowledge [44], and most of the software engineering experiments have not been replicated [45], we provide a complete experimental package including raw data, R scripts, and other detailed material in order to encourage other researchers to replicate our experiment or take our findings as starting point to carry out other valuables empirical assessments.

Bibliography

[1] Wettel, R., Lanza, M., & Robbes, R. (2011, May). Software systems as cities: a controlled experiment. In Proceedings of the 33rd International Conference on Software Engineering (pp. 551-560). ACM.J.

[2] Shahin, M., Liang, P., & Babar, M. A. (2014). A systematic review of software architecture visualization techniques. Journal of Systems and Software, 94, 161-185.

[3] Seriai, A., Benomar, O., Cerat, B., & Sahraoui, H. (2014, September). Validation of Software Visualization Tools: A Systematic Mapping Study. In Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on (pp. 60-69). IEEE

[4] Dąbrowski, R., Stencel, K., & Timoszuk, G. (2011). Software is a directed multigraph. In Software Architecture (pp. 360-369). Springer Berlin Heidelberg

[5] Wettel, R., Lanza, M., & Robbes, R. (2010). Empirical validation of CodeCity: A controlled experiment. Tech Report 2010/05, University of Lugano

[6] Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. Qualitative research in psychology, 3(2), 77-101.

[7] Salameh, H. B., Ahmad, A., & Aljammal, A. (2016, July). Software evolution visualization techniques and methods-a systematic review. In Computer Science and Information Technology (CSIT), 2016 7th International Conference on (pp. 1-6). IEEE.

[8] Friedenthal, S., Moore, A., & Steiner, R. (2006, July). OMG systems modeling language (OMG SysML™) tutorial. In INCOSE Intl. Symp.

[9] Rufiange, S., & Melançon, G. (2014, September). AniMatrix: A matrix-based visualization of software evolution. In Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on (pp. 137-146). IEEE.

[10] Teyseyre, A. R., & Campo, M. R. (2009). An overview of 3D software visualization. IEEE transactions on visualization and computer graphics, 15(1), 87-105.

[11] Storey, M. A., Wong, K., Fong, P., Hooper, D., Hopkins, K., & Műller, H. A. (1996, November). On designing an experiment to evaluate a reverse engineering tool. In Reverse Engineering, 1996., Proceedings of the Third Working Conference on (pp. 31-40). IEEE.

[12] Lemieux, F., & Salois, M. (2006). Visualization techniques for program comprehension. New Trends in Software Methodologies, Tools and Techniques (eds. H. Fujita and M. Mejri), 22-47.

[13] Storey, M. A., Best, C., & Michand, J. (2001). Shrimp views: An interactive environment for exploring java programs. In Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on (pp. 111-112). IEEE.

[14] Fleischer, R., & Hirsch, C. (2001). Graph drawing and its applications. In Drawing graphs (pp. 1-22). Springer Berlin Heidelberg.

[15] Herman, I., Melançon, G., & Marshall, M. S. (2000). Graph visualization and navigation in information visualization: A survey. IEEE Transactions on visualization and computer graphics, 6(1), 24-43.

[16] Kaufmann, M., & Wagner, D. (Eds.). (2003). Drawing graphs: methods and models (Vol. 2025). Springer.

[17] Mili, R., & Steiner, R. (2002). Software Engineering. In Software Visualization (pp. 129-137). Springer Berlin Heidelberg.

[18] Easterbrook, S., Singer, J., Storey, M. A., & Damian, D. (2008). Selecting empirical methods for software engineering research. In Guide to advanced empirical software engineering (pp. 285-311). Springer London.

[19] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). Experimentation in software engineering. Springer Science & Business Media.

[20] Runeson, P., Host, M., Rainer, A., & Regnell, B. (2012). Case study research in software engineering: Guidelines and examples. John Wiley & Sons.

[21] Robinson, H., Segal, J., & Sharp, H. (2007). Ethnographically-informed empirical studies of software practice. Information and Software Technology, 49(6), 540-551.

[22] Davison, R., Martinsons, M. G., & Kock, N. (2004). Principles of canonical action research. Information systems journal, 14(1), 65-86.

[23] Maalej, W., Tiarks, R., Roehm, T., & Koschke, R. (2014). On the comprehension of program comprehension. ACM Transactions on Software Engineering and Methodology (TOSEM), 23(4), 31.

[24] Murphy, G. C., Kersten, M., & Findlater, L. (2006). How are Java software developers using the Elipse IDE?. IEEE software, 23(4), 76-83.

[25] Ko, A. J., DeLine, R., & Venolia, G. (2007, May). Information needs in collocated software development teams. In Proceedings of the 29th international conference on Software Engineering (pp. 344-353). IEEE Computer Society.

[26] Storey, M. A. (2005, May). Theories, methods and tools in program comprehension: Past, present and future. In 13th International Workshop on Program Comprehension (IWPC'05) (pp. 181-191). IEEE.

[27] Walenstein, A. (2003, May). Observing and measuring cognitive support: Steps toward systematic tool evaluation and engineering. In Program Comprehension, 2003. 11th IEEE International Workshop on (pp. 185-194). IEEE.

[28] Aracic, I., & Mezini, M. (2006). Flexible abstraction techniques for graph-based visualizations. In Eclipse Technology eXchange workshop (eTX) at ECOOP.

[29] Aracic, I., Schaeffer, T., Mezini, M., & Osterman, K. (2007). A survey on interactive grouping and filtering in graph-based software visualizations. Technical Report, Technische Universität Darmstadt.

[30] Lungu, M., Lanza, M., & Gîrba, T. (2006, March). Package patterns for visual architecture recovery. In Conference on Software Maintenance and Reengineering (CSMR'06) (pp. 10-pp). IEEE.

[31] Caserta, P., & Zendra, O. (2011). Visualization of the static aspects of software: A survey. IEEE transactions on visualization and computer graphics, 17(7), 913-933.

[32] Knodel, J., Muthig, D., & Naab, M. (2008). An experiment on the role of graphical elements in architecture visualization. Empirical Software Engineering, 13(6), 693-726.

[33] Cornelissen, B., Zaidman, A., & Van Deursen, A. (2011). A controlled experiment for program comprehension through trace visualization.

[34] Haitzer, T., & Zdun, U. (2013, July). Controlled experiment on the supportive effect of architectural component diagrams for design understanding of novice architects. In European Conference on Software Architecture (pp. 54-71). Springer Berlin Heidelberg.

[35] Quante, J. (2008, June). Do Dynamic Object Process Graphs Support Program Understanding? -A Controlled Experiment. In Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on (pp. 73-82). IEEE.

[36] Fittkau, F., Finke, S., Hasselbring, W., & Waller, J. (2015, May). Comparing trace visualizations for program comprehension through controlled experiments. In Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (pp. 266-276). IEEE Press.

[37] Lungu, M., Lanza, M., & Nierstrasz, O. (2014). Evolutionary and collaborative software architecture recovery with Softwarenaut. Science of Computer Programming, 79, 204-223.

[38] Sillito, J., Murphy, G. C., & De Volder, K. (2006, November). Questions programmers ask during software evolution tasks. In Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (pp. 23-34). ACM

[39] Fittkau, F., Krause, A., & Hasselbring, W. (2015, September). Exploring software cities in virtual reality. In Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on (pp. 130-134). IEEE.

[40] Pacione, M. J., Roper, M., & Wood, M. (2004, November). A novel software visualisation model to support software comprehension. In Reverse Engineering, 2004. Proceedings. 11th Working Conference on (pp. 70-79). IEEE.
[41] Sensalire, M., Ogao, P., & Telea, A. (2009, September). Evaluation of software visualization tools: Lessons learned. In Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on (pp. 19-26). IEEE.
[42] Vegas, S. (2015). What Makes a Good Empirical Software Engineering Thesis?: Some Advice.
[43] Kampenes, V. B., Dybå, T., Hannay, J. E., & Sjøberg, D. I. (2007). A systematic review of effect size in software engineering experiments. Information and Software Technology, 49(11), 1073-1086.
[44] de Magalhães, C. V., da Silva, F. Q., Santos, R. E., & Suassuna, M. (2015). Investigations about replication of empirical studies in software engineering: A systematic mapping study. Information and Software Technology, 64, 76-101.
[45] Da Silva, F. Q., Suassuna, M., França, A. C. C., Grubb, A. M., Gouveia, T. B., Monteiro, C. V., & dos Santos, I. E. (2014). Replication of empirical studies in software engineering research: a systematic mapping study. Empirical Software Engineering, 19(3), 501-557.
[45] Sim, S. E., Clarke, C. L., Holt, R. C., & Cox, A. M. (1999). Browsing and searching software architectures. In Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on (pp. 381-390). IEEE.
[46] Synytskyy, N., Holt, R. C., & Davis, I. (2005, May). Browsing software architectures with LSEdit. In 13th International Workshop on Program Comprehension (IWPC'05) (pp. 176-178). IEEE.
[47] Stratton, W. C., Sibol, D. E., Lindvall, M., & Costa, P. (2007, March). The SAVE tool and process applied to ground software development at JHU/APL: an experience report on technology infusion. In Software Engineering Workshop, 2007. SEW 2007. 31st IEEE (pp. 187-193). IEEE.

[48] Telea, A., Voinea, L., & Sassenburg, H. (2010). Visual tools for software architecture understanding: A stakeholder perspective. IEEE software, 27(6), 46.

[49] Beck, F., & Diehl, S. (2012). Visual comparison of software architectures. Information Visualization, 1473871612455983.

[50] Lungu, M., & Lanza, M. (2007, March). Exploring inter-module relationships in evolving software systems. In 11th European Conference on Software Maintenance and Reengineering (CSMR'07) (pp. 91-102). IEEE.

[51] Balzer, M., Noack, A., Deussen, O., & Lewerentz, C. (2004). Software landscapes: Visualizing the structure of large software systems. In IEEE TCVG.