EVALUATION OF THE USEFULNESS OF DIAGRAMS OF THE RUN-TIME STRUCTURE FOR CODING ACTIVITIES

by

NARIMAN AMMAR

THESIS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

2011

MAJOR: COMPUTER SCIENCE

Approved By:

Advisor	Date

DEDICATION

To my family for their unlimited support.

ACKNOWLEDGEMENTS

This thesis work is a joint effort, and it is a pleasure to thank all those who made it possible. First and foremost, I wish to express my gratitude to my advisor, Marwan Abi-Antoun to whom I attribute the level of my Masters degree. I would like to thank him for his patience, knowledge, and criticism while allowing me to work in my own way. His rigor, enthusiasm, and passion in research motivates me to keep going.

Deepest gratitude is also given to the members of the thesis committee, for their constructive comments on this thesis. I am thankful that they accepted to be members of the reading committee on a short notice. Many thanks go, in particular, to Prof. Marcus, who advised me early during my Masters degree. I am especially thankful for his help in looking for confounds in my experimental design. I want to thank Prof. Reynolds, who gave me new insights on how to improve my research, ideas that I was not aware of. It is an honor for me to have Prof. Scaffidi from Oregon State University on my committee. I am grateful to him for his critical eye and foresight.

Special thanks to Thomas LaToza from Carnegie Mellon University, who helped me learn how to apply HCI techniques to my research and gave me helpful advice and feedback on the study design and various write-ups. I wish to also thank Erkan Isikli and Yong Wang for their useful advice on statistical analysis. I am thankful to Sara Tipton for feedback on several drafts of this thesis.

I would like to thank all my graduate friends. I especially thank the following SEVERE group members: Talia Selitsky, who involved me in a new culture, helped me expand my professional network, provided wonderful company, and bestowed endless support; Sonia Haiduc and Laurentiu Vanciu, for sharing invaluable knowledge and providing critical feedback; Suhib Rawshdeh, for always being supportive; and, Zeyad Hailat, for being a nice lab mate and for criticizing my paper drafts. I would like to thank Surafel Abebe who visited our research lab last year and encouraged me to pursue a Ph.D degree. I would also like to thank Calin Voichita and Cristina Mitrea

from BioInformatics lab for their support and useful advice.

I also offer thanks to several faculty who supported me. I would like to thank Prof. Wolfson for offering me jobs in the summer camps. I was delighted to interact with Prof. Brockmeyer in several exciting projects and workshops. Prof. Goel has always been a constant source of encouragement during my graduate study.

I am extraordinarily fortunate for being a member of the ACM student chapter at Wayne State University. I convey special acknowledgement to Monika Witoslawski, who helped in starting the group early this year, for always being on my side and providing me with valuable opportunities. I want to also thank Paul Janiczek for helping me advertise the study and for providing informative feedback on an earlier draft of this thesis.

I would like to convey thanks to the Department of Computer Science for providing the financial means and laboratory facilities for conducting the study. I was also generously supported by the Graduate School, who awarded me the Graduate Professional Scholarship.

I am indebted to many of my professors, colleagues, and friends in Palestine who supported me before moving to the US. In particular, I thank Prof. Wasfi Kafri, Prof. Abdul-Latif Abu-Issa, and Prof. Jehad Draidi. I am grateful to Mr. Hasan Omar for helping me get my first IT job. I also thank Dr. Yahya Al-Salqan for hiring me and encouraging me to pursue a graduate degree. To the role model, Nour Jbour, I want to thank him for nourishing my maturity and broadening my perspective on the practical aspects in the industry. Not forgetting my best friends, Maram and Nader for always being there for me.

I wish to express my love and gratitude to my family for their understanding through the duration of my study. I would especially like to thank my wonderful parents, sisters, and niece, 'Leen'. I want to thank my brother-in-law, Abdul Jabbar, and my brother, Ammar, for being wonderful mentors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: Introduction

Software maintenance accounts for 50% to 90% of the costs over the life-cycle of a software system. And program comprehension is a major activity during software maintenance, absorbing around half of the maintenance costs [11]. To support comprehension, researchers have produced many tools to visualize the structure of the system. The prevalent thinking is that such diagrams help with program comprehension. For example, a high-level view can help a developer locate where to implement a change [50].

1.1 The Problem

Studies have shown that during code maintenance tasks, developers ask questions about code in order to gather the information they need to successfully make changes [48, 29]. One type of question developers ask is about the relationships between classes or objects in code, including composition and inheritance relationships [48]. We conducted an exploratory study [2], and classified the questions or facts about the object structures that developers ask (Table 1.1). The study identified that 80% of the developers questions during coding activities are about object relations rather than class dependencies (Figure 1.1).

This percentage is unsurprising, since in object-oriented design patterns, much of the functionality is determined by what instances point to what other instances. For instance, in the Observer design pattern [21], understanding "what" gets notified during a change notification is crucial for the operation of the system, but "what" does not usually mean a class, "what" means a particular instance. Furthermore, a class diagram often shows several classes depending on a single container class such as `ArrayList` (Figure 1.2). However, different instantiations of an `ArrayList` often correspond to different elements in the design. Hence we need an instance-based

Table 1.1: Code definitions.

Code	Description
CD	Question/Fact about the Class Diagram
OOG	Question/Fact about the Object Diagram
Is-A	Question/Fact about an Is-A relationship (extends from class, implements interface)
Has-A	Question/Fact about a Has-A relationship
Is-Part-Of	Question/Fact whether an object is logically part of another object (inside a public domain)
Is-Owned	Question/Fact whether an object is strictly owned by another object (inside a private domain)
Is-In-Tier	Question/Fact whether an object is in some runtime tier
Points-To	Question/Fact whether an object points or refers to another object
Has-Label	Question/Fact about the label of an element in the diagram
Cardinality	Question/Fact about the cardinality of an object relation (1-to-1 or 1-to-many)
Navigability	Question/Fact about going from A to B (directed edge)
How-To-Get-X	Question/Fact about how to get a reference to an object X
May-Alias	Question/Fact about whether two references may refer to the same object at runtime
May-Not-Alias	Question/Fact about whether two references may not refer to the same object at runtime

view to *complement* a class diagram. For example, in the Design Patterns book [21], Gamma et al. used both class and object diagrams to explain several structural design patterns such as Proxy, Mediator and Composite.

Class diagrams show the type structure of the program and do not explain the object structure. Another important view is an *object diagram* or *object graph*, where nodes represent objects, i.e., instances of the classes in a class diagram, and edges correspond to relations between objects. An object graph shows distinct instances of the class `ArrayList` (Figure 1.3). Therefore, an object diagram makes explicit the structure of the objects instantiated by the program and their relations, facts that are only implicit in a class diagram. While in the class diagram a single node represents a class and summarizes the properties of all of its instances, an object diagram represents different instances as distinct nodes, with their own properties [50].

The software research community has focused heavily on class diagrams and partial object diagrams, such as sequence diagrams, and studied their usefulness empirically [22]. Many tools can automatically generate class diagrams from pro-

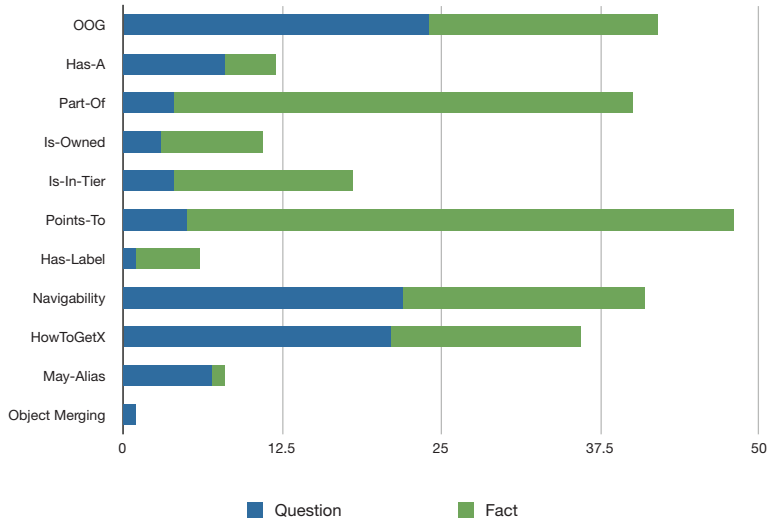


Figure 1.1: During coding activities 80% of developers questions are about the object structure. The horizontal axis shows how frequently developers ask a question or state a fact about the object structure. The vertical axis classifies the specific types of questions according to the coding scheme in Table 1.1.

grams [30]. Object diagrams, on the other hand, have less mature tool support and less widespread use. Few tools extract object diagrams, and the ones that do extract flat object graphs that do not scale to an entire program (Figure 1.4(a)).

1.2 A Solution

Abi-Antoun and Aldrich have recently proposed the Scholia approach to extract hierarchical ownership object graphs (OOGs) from object-oriented code, which serve as global object diagrams of the entire system [3]. The goal of this thesis work is to evaluate the usefulness of OOGs to developers doing code modifications. Therefore, we set out to investigate the following research question:

> *"Do developers who have access to diagrams of the run-time structure, i.e. OOGs, in addition to diagrams of the code structure, i.e. class diagrams, perform code modification tasks more effectively than developers who have access to only diagrams of the code structure?"*

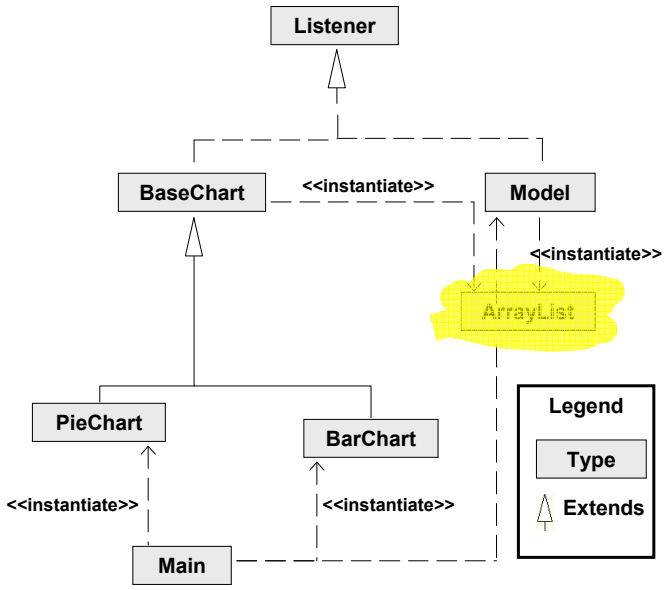


Figure 1.2: ArrayList on a UML class diagram.

1.3 Contributions

Our contribution in this thesis is the first controlled experiment to evaluate global object diagrams. We designed and conducted a controlled experiment on a pedagogical object-oriented framework from which we extracted both global class diagrams and global object diagrams. We observed 10 developers, organized into an experimental and a control group, use the diagrams while they perform realistic code modification tasks.

1.4 Thesis Statement

The thesis is:

> *"Developers who have access to diagrams of the run-time structure are more effective in performing code modification tasks than developers who*

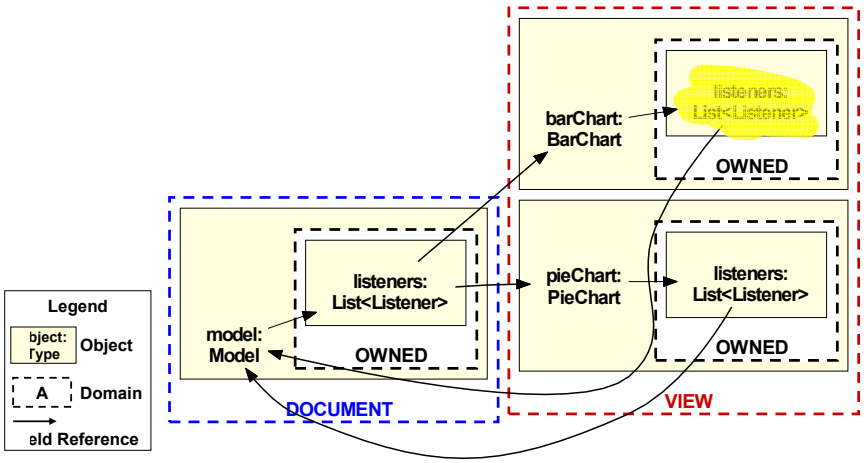


Figure 1.3: ArrayList on an object graph.

have access to only diagrams of the code structure."

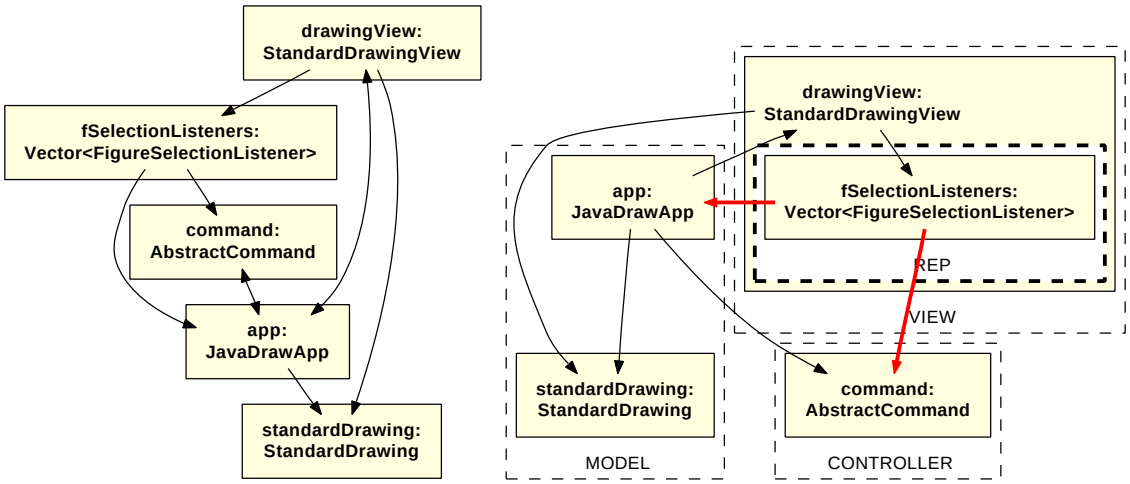
1.5 Hypotheses and Measures

We created several corresponding hypotheses, subordinate to the main thesis. Since each hypothesis is smaller than the main thesis, each can be directly supported by evidence.

H1: Developers who have access to OOGs answer questions about the object structure that cannot be answered using class diagrams.

Success Criteria. The success criteria to objectively measure or falsify this hypothesis include:

- Developers perform their tasks in a sequence of activities related to questions about the object structure.
- Developers use relations between objects on an OOG to answer their questions about the object structure.



(a) Flat object graph. (b) Hierarchical Ownership Object Graph (OOG).

Figure 1.4: MicroDraw object structures.

H2: Developers who have access to OOGs explore fewer code elements to complete their tasks.

Success Criteria. The success criteria to objectively measure or falsify this hypothesis include:

- Developers use OOGs to understand object relations, and navigate fewer paths through the code.
- Developers use traceability links on the OOG to navigate to only relevant code.

H3: Developers who have access to OOGs take less time to complete their tasks.

Success Criteria. The success criteria to objectively measure or falsify this hypothesis include:

- Developers who have access to OOGs take less time to gain a high level understanding of the code.
- Developers who have access to OOGs follow shorter paths through the code to complete their tasks.

H4: Developers who have access to OOGs are more successful on their tasks.

Success Criteria. The success criteria to objectively measure or falsify this hypothesis include:

- Developers who have access to OOGs are more focused and follow fewer unsuccessful paths through the code.
- Developers who have access to OOGs make correct assumptions and have enough time to complete their tasks.

Evidence. We support each of the success criteria for each of the above hypotheses with direct evidence by analyzing the results of a controlled experiment conducted on multiple developers modifying a well-designed object-oriented framework.

1.6 Outline

The remainder of this thesis is organized as follows: Chapter 2 gives some background information on the diagrams we are evaluating. Chapter 3 discusses how we extracted and refined the OOGs we gave to the study participants. Chapter 4 discusses the design of the experiment, and in Chapter 5, we provide qualitative and quantitative analysis of our findings. Chapter 6 discusses related work. We discuss limitations of this work, lessons learned, and future work in Chapter 7 and conclude.

Chapter 2: Background

In SCHOLIA, architectural extractors use ownership domain annotations to annotate an object-oriented system, then they use static analysis to extract a hierarchical Ownership Object Graph (OOG) that soundly approximates a run-time object graph that any program run may generate. An extracted OOG is sound in two respects. First, each run-time object has exactly one representative in the OOG. Second, the OOG has edges that correspond to all possible run-time points-to relations between those objects [1].

To achieve hierarchy in an object graph, SCHOLIA requires that a developer pick a top-level object as a starting point, then use modular ownership annotations in the code [9] to impose a conceptual hierarchy on objects. The annotations specify object encapsulation, logical containment and architectural tiers within the code, constructs which are not explicit in most programming languages. The SCHOLIA tools use existing language support for annotations. In addition, the annotations implement a type system, so a typechecking tool can validate the annotations and identify inconsistencies between the annotations and the code.

2.1 Ownership Domains

An *ownership domain* is a conceptual group of objects with an explicit name and explicit policies that govern how a domain can reference objects in other domains [9]. Each object is assigned to a single domain that does not change at runtime. A developer indicates the domain of an object by annotating each reference to that object in the program. The annotations define two kinds of object hierarchy, logical containment and strict encapsulation, defined below (Fig. 2.1).

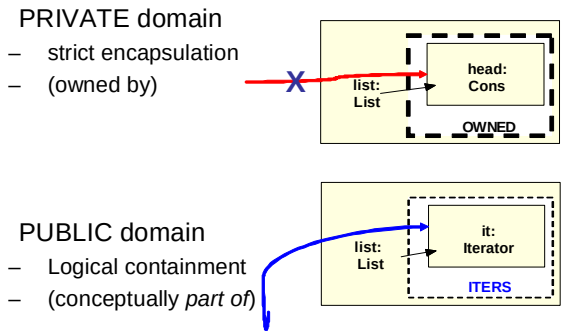


Figure 2.1: Public domains vs. private domains.

2.1.1 Logical Containment (part-of)

A public domain provides *logical containment*, thus making an object conceptually *part of* another object. Having access to an object gives the ability to access objects inside all its public domains.

2.1.2 Strict Encapsulation (owned-by)

A private domain provides *strict encapsulation*, thus making an object *owned by* its parent object. Then, a `public` method cannot return an alias to an object in a private domain, even though the Java type system allows returning an alias to a field marked `private`.

2.2 Ownership Object Graph (OOG)

Once the annotations are in place, we use a static analysis to extract a hierarchical Ownership Object Graph (OOG). An OOG provides architectural abstraction by ownership hierarchy, by showing architecturally significant objects near the top of the hierarchy and data structures further down. An OOG can also provide abstraction by types, by collapsing objects further based on their declared types.

2.2.1 Graphical Notation

The visualization uses box nesting to indicate containment of objects inside domains and domains inside objects (Fig. 1.3). Dashed-border, white-filled boxes represent domains. Solid-filled boxes represent objects. Solid edges represent field references. An object labeled `obj:T` indicates an object reference `obj` of type `T`, which we then refer to either as "object `obj`" or as "`T` object," meaning "an instance of the `T` class." A private domain has a thick, dashed border; a public domain, a thin one. Finally, having a hierarchical representation enables displaying or eliding information at any level to show overviews of the runtime architecture at the desired level of abstraction. A (+) symbol on an object or a domain indicates that it has a collapsed substructure. Dotted edges summarize any solid edges by lifting them from elided objects to visible ones.

2.2.2 Facts Developers Can Learn from an OOG

By looking at the OOG (Figure 1.3), developers may learn several facts or answer several of their questions about the system, which we classify using the codes from our coding model (Table 1.1):

- The OOG conveys which object is in which tier (Code *Is-In-Tier*). For instance, the `pieChart`, `barChart`, and `model` objects appear in different architectural tiers. The `pieChart` and `barChart` are in the `UI` tier, and the `model` is in the `DOCUMENT` tier. Such information is missing from the class diagram.

- The OOG also shows that some objects are owned by others (Code *Is-Owned*). Each of the `pieChart`, `barChart`, and `model` objects owns a distinct listeners: `ArrayList` object. In contrast, the class diagram shows a single `ArrayList` class. Indeed, an OOG can distinguish between different instances of the same class that are in different domains.

- The OOG also conveys some precision about aliasing (Code *May-Alias*). Objects in different domains may not alias. So, the `listeners` object inside the `model` object cannot alias the `listeners` object inside the `pieChart` object.

- The OOG also shows object references in the system (Code *Points-To*). Both `pieChart` and `barChart` register themselves as listeners to the `model` and listen to changes on the `model` to update their corresponding views. The OOG does not show a points-to edge between `pieChart` and `barChart`. In contrast, in the class diagram, several classes have an association with the `Listener` interface, but it is unclear, if at run-time, instances of all these classes share the same `Listener` object.

In summary, an OOG shows more precise information than a class diagram (Figure 1.2). An OOG may convey the design intent in the system, such as the Observer design pattern in this instance. An OOG may also help developers understand the object structure of the system. In particular, they can learn facts about which run-time tier an object belongs to (*Is-In-Tier*), which object points to which other object (*Points-To*), or which object is part of or owned by another object (*Is-Par-Of*, *Is-Owned*), among others.

Chapter 3: OOG Extraction and Refinement

An OOG is a global object diagram of the entire system. Because an OOG is a global rather than a task-specific view, it runs the risk of not being useful to developers. We make an OOG relevant for code modification tasks by making it convey design intent and reflect the developers' mental model of the system. In the Scholia approach, the abstraction is not hard-coded in the tools, but is customizable in various ways.

Producing an OOG is an iterative process. We first use annotations in the code to specify some design intent, such as architectural tiers and architectural hierarchy, which cannot be expressed in general purpose programming languages. We then pick a top-level object as a starting point, then use ownership annotations in the code to impose a conceptual hierarchy on all the objects in the system. Thus, in an OOG, architecturally significant objects appear near the top of the hierarchy and data structures appear further down. Once the annotations are in the code and type check correctly, we run a static analysis to extract some initial OOGs. Typically, the initial OOGs need further refinement. We refine an OOG by changing the annotations in the code or by fine-tuning some of the static analysis settings.

In this chapter, we discuss the setup to produce OOGs for the subject system (Section 3.1). We then discuss adding annotations (Section 3.2), extracting initial OOGs (Section 3.3) and refining them (Section 3.4). Finally, we discuss limitations (Section 3.5) and various issues (Section 3.6) and conclude.

3.1 Study Setup

We discuss the setup to extract initial OOGs for the subject system and refine them.

3.1.1 Subject System

As the subject system, we chose MiniDraw [36], a pedagogical object-oriented framework. MiniDraw consists of around 1,500 lines of Java code, 31 classes and 17 interfaces (Figure 3.1). We chose MiniDraw because it is a well-designed, object-oriented framework. Also, MiniDraw comes with insights into its design, discussed in a book by Christensen [13].

MiniDraw is an example of a framework which is a highly flexible software system customizable in a number of different ways. It is especially suited to support the graphical aspects of board games and the like that let users manipulate two dimensional graphical objects. Some of the applications supported by MiniDraw include board games such as BreakThrough, LogoPuzzle, HotGammon, and HotCiv. Among the other applications are RectangleDraw which is a drawing application similar to JHotDraw [27]. We selected the Breakthrough board game application.

Metrics - MiniDraw_Participant - Number of Static Attributes (avg/max per type)

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Overridden Methods (avg/max per type)	20	0.645	1.233	4	/MiniDraw_Participant/src/minidraw/standard/SelectionTool.java	
Number of Attributes (avg/max per type)	65	2.097	1.802	6	/MiniDraw_Participant/src/minidraw/boardgame/BoardActionTool.java	
Number of Children (avg/max per type)	11	0.355	0.863	4	/MiniDraw_Participant/src/minidraw/standard/AbstractTool.java	
Number of Classes (avg/max per packageFragment)	31	6.2	3.311	12	/MiniDraw_Participant/src/minidraw/standard	
src	31	6.2	3.311	12	/MiniDraw_Participant/src/minidraw/standard	
minidraw.standard	12					
minidraw.breakthrough	7					
minidraw.boardgame	5					
minidraw.standard.handlers	5					
minidraw.framework	2					
Method Lines of Code (avg/max per method)	641	3.356	4.491	38	/MiniDraw_Participant/src/minidraw/standard/ImageManager.java	registerAllImages
src	641	3.356	4.491	38	/MiniDraw_Participant/src/minidraw/standard/ImageManager.java	registerAllImages
minidraw.standard	333	2.973	4.505	38	/MiniDraw_Participant/src/minidraw/standard/ImageManager.java	registerAllImages
minidraw.boardgame	139	5.148	5.835	23	/MiniDraw_Participant/src/minidraw/boardgame/BoardDrawing.java	pieceMovedEvent
minidraw.breakthrough	74	3.7	4.173	18	/MiniDraw_Participant/src/minidraw/breakthrough/BreakthroughPieceFact...	generatePieceMultiMap
minidraw.standard.handlers	83	3.32	2.724	12	/MiniDraw_Participant/src/minidraw/standard/handlers/StandardRubberBa...	selectGroup
minidraw.framework	12	1.714	0.452	2	/MiniDraw_Participant/src/minidraw/framework/FigureChangeEvent.java	FigureChangeEvent
Number of Methods (avg/max per type)	188	6.065	4.866	26	/MiniDraw_Participant/src/minidraw/standard/StandardDrawingView.java	
Nested Block Depth (avg/max per method)		1.241	0.574	4	/MiniDraw_Participant/src/minidraw/boardgame/BoardActionTool.java	mouseUp
Depth of Inheritance Tree (avg/max per type)		2	1.437	6	/MiniDraw_Participant/src/minidraw/standard/MiniDrawApplication.java	
Number of Packages	5					
Afferent Coupling (avg/max per packageFragment)		5.8	6.4	18	/MiniDraw_Participant/src/minidraw/framework	
Number of Interfaces (avg/max per packageFragment)	17	3.4	4.224	11	/MiniDraw_Participant/src/minidraw/framework	
McCabe Cyclomatic Complexity (avg/max per method)		1.346	0.848	6	/MiniDraw_Participant/src/minidraw/boardgame/BoardActionTool.java	mouseUp
Total Lines of Code	1489					
src	1489					
minidraw.standard	707					
minidraw.boardgame	269					
minidraw.standard.handlers	194					
minidraw.breakthrough	177					
minidraw.framework	142					
Instability (avg/max per packageFragment)		0.568	0.328	1	/MiniDraw_Participant/src/minidraw/breakthrough	
Number of Parameters (avg/max per method)		1.199	1.05	4	/MiniDraw_Participant/src/minidraw/boardgame/BoardFigure.java	BoardFigure
Lack of Cohesion of Methods (avg/max per type)		0.332	0.36	1	/MiniDraw_Participant/src/minidraw/framework/FigureChangeEvent.java	
Efferent Coupling (avg/max per packageFragment)		4.2	3.311	10	/MiniDraw_Participant/src/minidraw/standard	
Number of Static Methods (avg/max per type)	3	0.097	0.296	1	/MiniDraw_Participant/src/minidraw/breakthrough/BreakThrough.java	
Normalized Distance (avg/max per packageFragment)		0.138	0.091	0.286	/MiniDraw_Participant/src/minidraw/standard/handlers	
Abstractness (avg/max per packageFragment)		0.344	0.301	0.846	/MiniDraw_Participant/src/minidraw/framework	
Specialization Index (avg/max per type)		0.277	0.709	3.6	/MiniDraw_Participant/src/minidraw/standard/StdViewWithBackground.java	
Weighted methods per Class (avg/max per type)	257	8.29	6.228	34	/MiniDraw_Participant/src/minidraw/standard/StandardDrawingView.java	
Number of Static Attributes (avg/max per type)	10	0.323	1.118	5	/MiniDraw_Participant/src/minidraw/breakthrough/Constants.java	

Figure 3.1: MiniDraw code statistics, obtained using the Eclipse Metrics Plugin [20].

3.1.2 People Involved

We use the term *developer* (the author of this thesis) to refer to the person who was refining the OOG to convey design intent. The developer built her mental model of the system, by performing several code modification tasks on the system.

We refer to the person who was adding annotations to the code and extracting OOGs as the *architectural extractor*. Three architectural extractors were involved in the annotation process of MiniDraw, 2 Ph.D. students[1] and one of the developers of Scholia and the tools. The architectural extractors added annotations to the code and ran the static analysis to extract OOGs. Based on feedback from the developer, they also fine-tuned the extracted OOGs to reflect the developer's mental model.

3.1.3 Tools and Instrumentation

Architectural abstraction. An OOG provides architectural abstraction by ownership hierarchy and by types. The first form of abstraction, by ownership hierarchy, is based on annotations in the code. Architectural extractors push less architecturally relevant objects, such as data structures, underneath more architecturally relevant ones, such as objects that are application-specific. The second form of abstraction is by types, where the architectural extractors specify architecturally relevant types, such as core framework interfaces. The analysis then merges objects of those types that are in the same domain to produce a less cluttered OOG.

The type checker tool, ArchCheckJ. The architectural extractors add ownership domain annotations in the code. Then, they use the ArchCheckJ type checker to check that the annotations are consistent with each other and with the code.

[1]The annotation effort is currently estimated at around 1-hour per 1,000 Lines of Code [1]. We do not require 2 Ph.D. students for adding annotations to MiniDraw (1,500 LOC). MiniDraw was the first system on which they practiced adding annotations and extracting OOGs. The third extractor was an expert in adding annotations, and he mentored the first two extractors, to ensure that they added good quality annotations.

ArchChechJ displays annotation warnings in the Eclipse Problems window. The extractors double-click on the annotation warning to go to the problematic line of code, and manually fix the annotation warning.

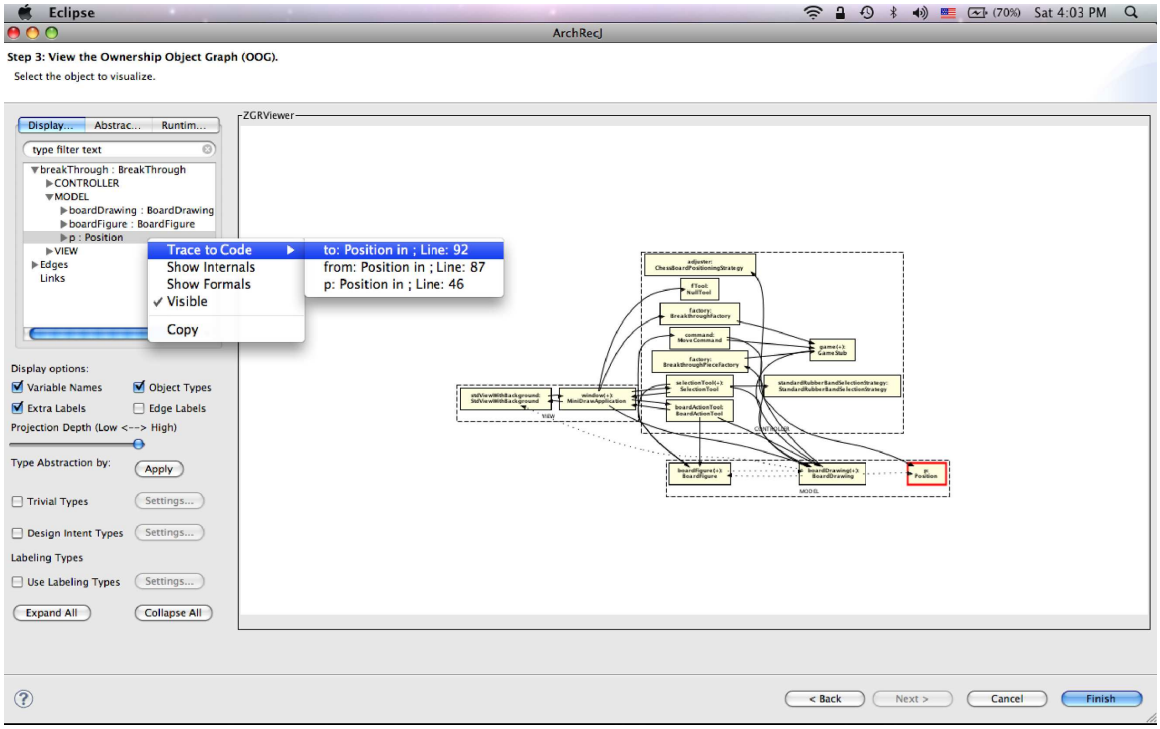


Figure 3.2: The ArchRecJ tool used by the architectural extractors to view and manipulate the extracted OOG.

The OOG extraction tool, ArchRecJ. The architectural extractors used the OOG extraction tool (Figure 3.2), ArchRecJ, to extract OOGs. ArchRecJ has features to allow the architectural extractors to manipulate the extracted graph (Table 3.1).

3.1.4 General Procedure to Extract and Refine the OOG

The general procedure for producing an OOG consists of the following steps:

1. **Add annotations to the code:** in this step, the architectural extractors add annotations to the code, run the typechecker to check the annotations, and manually fix any annotation warnings. The goal in this step is to minimize the

Table 3.1: ArchRecJ features.

Feature	Description
Select top-level object	the user can interactively select an object as the root of the graph to view its substructure
Set trivial types	a developer can specify an optional list of trivial types to use the abstraction by types feature
Set design intent types	a developer can specify an optional list of design intent types to use the abstraction by types feature
Display inheritance hierarchy	the tool can display the inheritance hierarchy of the types of the field declarations that a display object merges, to help the developer fine-tune the list of trivial types or design intent types for the abstraction by types
Collapse/expand objects	a developer can collapse or expand the sub-structure of a selected object or domain
Control unfolding depth	a developer can control the visible depth of the ownership tree using the slider control
Set object labels	Each object in an extracted object graph represents at least one field or variable declaration in the program. An object might have multiple types, and the analysis picks one of those types as the label. ArchRecJ can label objects with an optional field name or variable name and an optional type name. The type used in the label consists of a least-upper-bound type or a design intent type or a labeling type (discussed below)
Set additional labeling types	the object graph extraction non-deterministically selects a label for a given object o based on the name or the type of one of the references in the program that points to o. A developer can specify an optional list of labeling types for labeling objects.
Trace to code	the tool can show the list of field declarations and their types that a given display object merges. In addition, the developer can trace from the field declarations to the right lines of code. This feature is useful to guide the developer to the field declarations in the program that require different annotations.
Persist extracted OOG	the tool can persist an extracted OOG into an XML file. This file can then be viewed using a standalone viewer.

number of annotation warnings. The warnings produced by the type checker are alerts that the OOG may be unsound, so the architectural extractors must attempt to resolve most of these warnings before taking the next steps.

2. **Extract initial OOGs:** in this step, the architectural extractors run the static analysis to extract initial OOGs, and tweak the annotations to try to obtain a less cluttered OOG. The goal here is to reduce the number of objects in the top-level domains in the OOGs.

3. **Refine the extracted OOGs:** in this step, the architectural extractors work with developers to make the OOG more relevant,and make it convey their design

intent. Indeed, one way to assess the usefulness of a reverse-engineered diagram is to ask the original designers of the system if the extracted diagram reflects their design intent [50]. Another way is to try to use the diagram while doing code modification tasks. Because we did not have access to the original designers of MiniDraw, the developer performed a few representative code modification tasks and gave the architectural extractors feedback on the OOG (Figure 3.3).

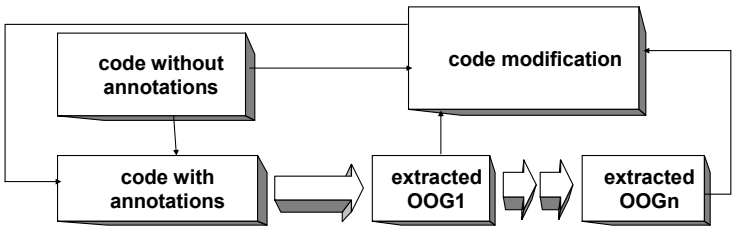


Figure 3.3: Refining the OOG while performing code modifications and referring to the OOG.

In Sections 3.2, 3.3, 3.4, we discuss each step in turn, and how we followed the procedure for the MiniDraw subject system.

3.2 Adding Annotations

The architectural extractors first added ownership domain annotations to the code to specify some design intent, such as architectural tiers and architectural hierarchy. To guide the annotation process, the architectural extractors relied on some of the design documents provided by the MiniDraw designers [13]. In their book, the MiniDraw designers indicated that MiniDraw follows the Model-View-Controller design pattern [13]. So, the architectural extractors organized instances of the core types into three top-level domains, `MODEL`, `VIEW` and `CONTROLLER`, to represent a three-tiered architecture, as follows:

- `MODEL`: has instances of `Drawing` and `Figure` objects. A `Drawing` is composed of `Figures` that know their containing `Drawing`. The class `BoardDrawing` implements the `Drawing` interface.

- `VIEW`: has instances of `MiniDrawApp` and `DrawingView` objects. The `StdViewBckgnd` class implements `DrawingView` interface.

- `CONTROLLER`: has instances of `Command` objects. `MoveCommand` implements the `Command` interface. They also placed instances of the types that has to deal with the game logic, e.g. `GameStub` in this tier.

```
@Domains({"VIEW","CONTROLLER","MODEL"})
public class BreakThrough {
  @Domain("CONTROLLER<VIEW,CONTROLLER,MODEL>")
  Game game = new GameStub();
  @Domain("CONTROLLER<VIEW,CONTROLLER,MODEL>")
  BreakthroughFactory factory = new BreakthroughFactory(game);
  @Domain("VIEW<VIEW,CONTROLLER,MODEL>")
  DrawingEditor window = new MiniDrawApplication( "Breakthrough Demo",factory);

  public void init() {
    window.open();
    @Domain("MODEL<VIEW,CONTROLLER,MODEL>")
    BoardDrawing drawing = (BoardDrawing) window.drawing();
    @Domain("CONTROLLER<VIEW,CONTROLLER,MODEL>")
    GameStub gameStub = (GameStub) game;
    gameStub.addObserver( drawing );
    @Domain("CONTROLLER<VIEW,CONTROLLER,MODEL>")
    BoardActionTool boardActionTool = new BoardActionTool(window);
    @Domain("CONTROLLER<VIEW,CONTROLLER,MODEL>")
    SelectionTool selectionTool = new SelectionTool(window);
    window.setTool( boardActionTool );
  }

  public static void main(@Domain("lent[shared]")String[] args) {
    @Domain("lent")BreakThrough breakThrough = new BreakThrough();
    breakThrough.init();
 }}
```

Figure 3.4: Annotations added to the root class `BreakThrough`.

Figure 3.4 illustrates how the architectural extractors declared the top-level domains (`@Domains("VIEW","CONTROLLER","MODEL")`), and how they used the `@Domain("CONTROLLER<VIEW,CONTROLLER,MODEL>")` to declare the reference `game` of type `GameStub` in the `CONTROLLER` domain. Additional details on the annotation process are available in a separate technical report by the architectural extractors [23].

3.3 Extracting Initial OOGs

Once the architectural extractors fixed most of the annotation warnings, they ran the extraction tool to extract OOGs, and in a sense, visualize the added annotations. Based on visualizing the OOG, they refined the annotations to make the OOG less cluttered. They repeated this process until they got an OOG that was at an adequate level of abstraction.

One of the initial MiniDraw OOGs is in Figure. 3.5. The figure does not show the `breakThrough:BreakThrough` object since we often choose to hide the root object to reduce visual clutter. In that case, the domains declared on the root class become top-level domains. All the objects that appear on the OOG are instances within the root object and appear in different domains. For example, in the `MODEL` domain, a `BoardDrawing` points to `Position` and has a `selectionHandler` instance which has a list of figures (`selectionList:ArrayList<Figures>`. Instances in the `VIEW` domain point to instances in the `CONTROLLER`, e.g. `window` points to `factory`.

Until this point, the architectural extractors were mostly guided by the type-checker, and aimed to reduce the annotation warnings. They also worked to minimize the number of objects in the top-level domains in the extracted OOGs. At this point, it made sense to evaluate the quality of the extracted OOGs by giving them to the developer, and let her use them to determine if they conveyed important design intent, as she proceeded to do some code modifications on the system. She also had access to the OOG extraction tool to refine the OOGs further, as she deemed fit.

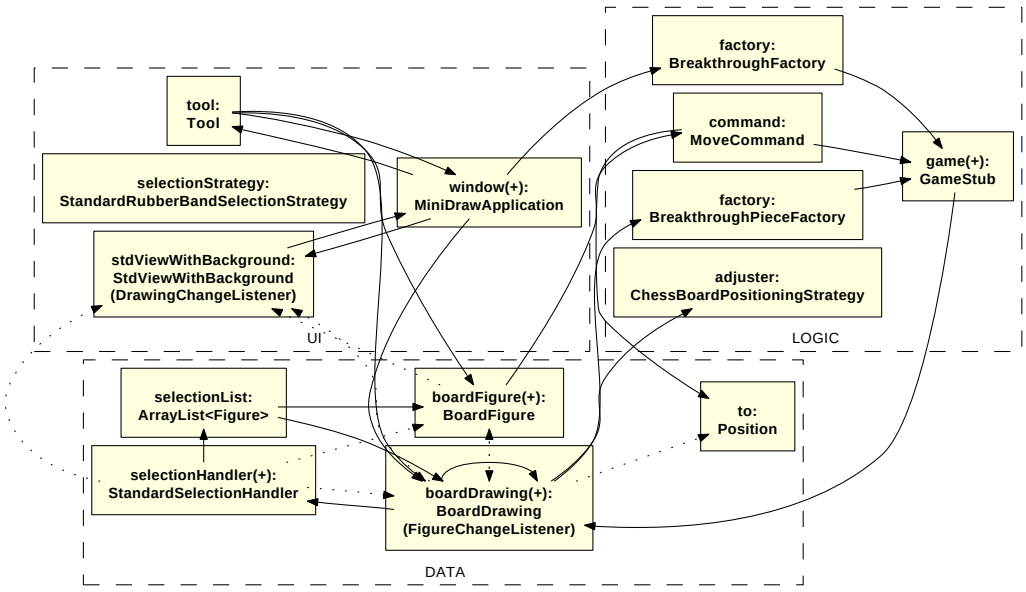


Figure 3.5: An OOG for BreakThrough (OOG1).

In the following section, we illustrate how the developer used the incremental knowledge that she built while doing the code modifications to refine the extracted OOGs. Eventually, she was able to get an OOG that she thought could be useful to developers attempting code modifications on MiniDraw in the future.

3.4 Refining the Extracted OOGs

The developer referred to Christensen's book [13] to better understand MiniDraw's design before she attempted to do any code modification task. According to the MiniDraw designers, MiniDraw uses the Model-View-Controller (MVC) architectural pattern. Furthermore, MiniDraw illustrates that MVC does not usually simply define three classes; here each are actually small subsystems with their own internal structure. Each subsystem by itself is programmed with focus on some of the principles of flexible design: program to an interface instead of an implementation class, favor object composition over inheritance, and use several design patterns.

3.4.1 Code Modification Tasks for Refining the OOG

MiniDraw has variability points that allow it to be customized with regards to the figures to show, the set of images to load, the tools that affect the figures, the type of drawing and view to use, and the ability to make objects observe figure state changes. Taking MiniDraw variability points into consideration, the developer designed several tasks to modify the different applications of MiniDraw, but she performed only the following tasks related to BreakThrough:

- **Implement an undo feature.** The goal of this task was to use the Command design pattern and to learn about the communication between view (`VIEW`), tool (`CONTROLLER`), drawing, and figure (`MODEL`).
- **Validate the movement of a piece.** The goal of this task was to use the strategy pattern to implement the validation logic which could be different from one board game to another.
- **Implement the capture of a piece.**
- **Implement auto alignment/adjustment of pieces.** The pieces can be put anywhere on the board, so the developer modified the code such that when a piece is dropped it is moved to the center of the square. This task was a seeded defect.
- **Implement owner for each piece.** To simulate the two roles/players in the game.

Attempting these tasks helped the developer incrementally build her mental model of the system, and let her suggest possible refinements of the OOG. The developer first gave the architectural extractors general feedback on the top-level OOG. Then, she used the knowledge she was building incrementally due to implementing the tasks

to provide more feedback. Table 3.2 summarizes the OOG refinement steps. She then used the refined OOGs during code modification tasks and provided more detailed refinement requests. She attempted to do some of these refinements herself and made changes to the annotations directly. But some of her refinements were not supported by the code, and generated annotation warnings. In those cases, she reverted the changes to the annotations, or sought the assistance of the architectural extractors.

In this section, we discuss how the developer requested several refinements to the OOG, and how the architectural extractors used the two types of architectural abstraction to make the OOG reflect the developer's mental model of the system. Note: some of the OOGs shown below may be unsound, i.e., may be missing edges because they were intermediate snapshots and extracted from the code with annotations that did not fully type check, before they were fixed.

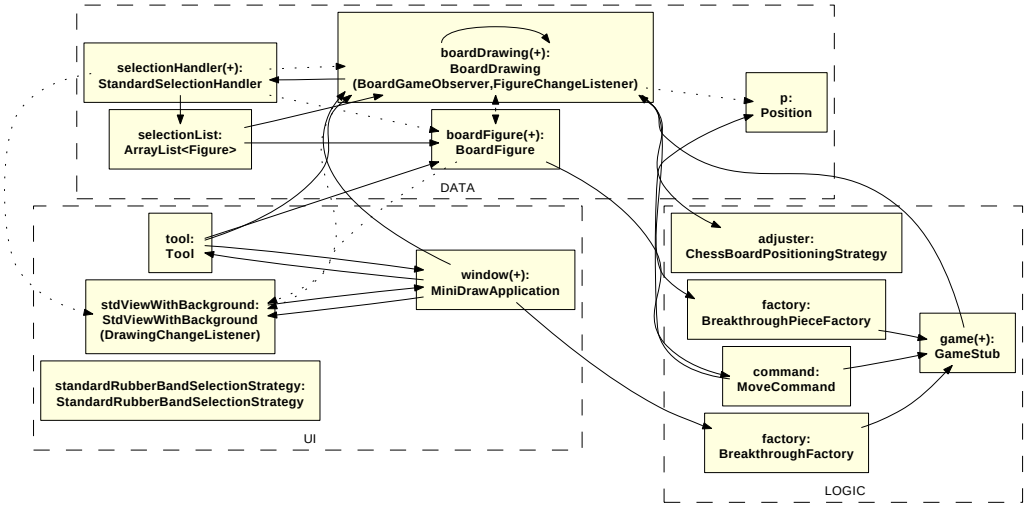


Figure 3.6: An OOG for BreakThrough (OOG2).

3.4.2 Moving objects across domains

At first glance, the developer thought that all instances of `Tool`, such as: `boardActionTool:BoardActionTool`, `cachedNullTool:NullTool`, `dragTracker:DragTracker`, and `selectAreaTracker:SelectAreaTracker`

Table 3.2: OOG refinement steps

Refinement step	Type of refinement	OOG
Added `BoardGameObserver` as a labeling type. Fix annotation warning related to having `lent` as return value of method. Made `List.iterator()` method return `unique`. Had to change AliasXML file	Displaying labeling types	OOG2
Renamed top level domains as `MODEL`, `VIEW`, `CONTROLLER`. Moved `rubberBandSelectiontStrategy` to `CONTROLLER`.	Moving objects across domains	OOG3
Added virtual field also to AliasXML for `java.util.List`. Added qualified domain names		OOG4
Removed `StandardDrawingChangeListenerHandler` from top-level domain. Pushed `SelectionHandler` into a private or public domain to remove from top-level domain	Manipulating ownership hierarchy	OOG7
Placed `StandardDrawingChangeListenerHandler` and `SelectionHandler` into a public domain on `StandardDrawing` called `HANDLERS`	Manipulating ownership hierarchy	OOG8
Added public domain `SUBS` to `DrawingChangeListnerHandler` and put the `listenerList` inside it. The `listenerList` now points to `stdView` to show that `view` listens to changes on `Drawing`	Adding missing edges	OOG9
Added virtual object allocation to make `Thread` appear. Thread is not instantiated using "new", but using a factory method inside the JDK, so we need to tell the analysis about this virtual allocation. Otherwise, Thread does not appear inside `LOCKS` in an Instantiation-Based View which looks for "new" object allocations `private @Domain("LOCKS") Thread NEWWWthread`	Adding missing objects	OOG9
No longer add `@Domains("owned")` to an interface. An interface cannot reference the `owned` PRIVATE domain. A class that implements that interface would have to make all those methods public. A public method cannot have `owned` anywhere in its signature. It makes sense to declare PUBLIC domains on an interface, however.		OOG10

should be moved to the `CONTROLLER` domain. Also, handlers such as `selectionHandler:StandardSelectionHandler` and all objects representing the strategy design pattern such as `rubberBandSelectionStrategy:RubberBandSelectionStrategy` should be moved to the `CONTROLLER` domain.

3.4.3 Manipulating the ownership hierarchy

The developer considered that the initial OOG showed non-architecturally significant objects in the top-level domains. For example, `map:Map<Position, List<Figure>>` and `l:ArrayList<BoardFigure>` should be pushed underneath

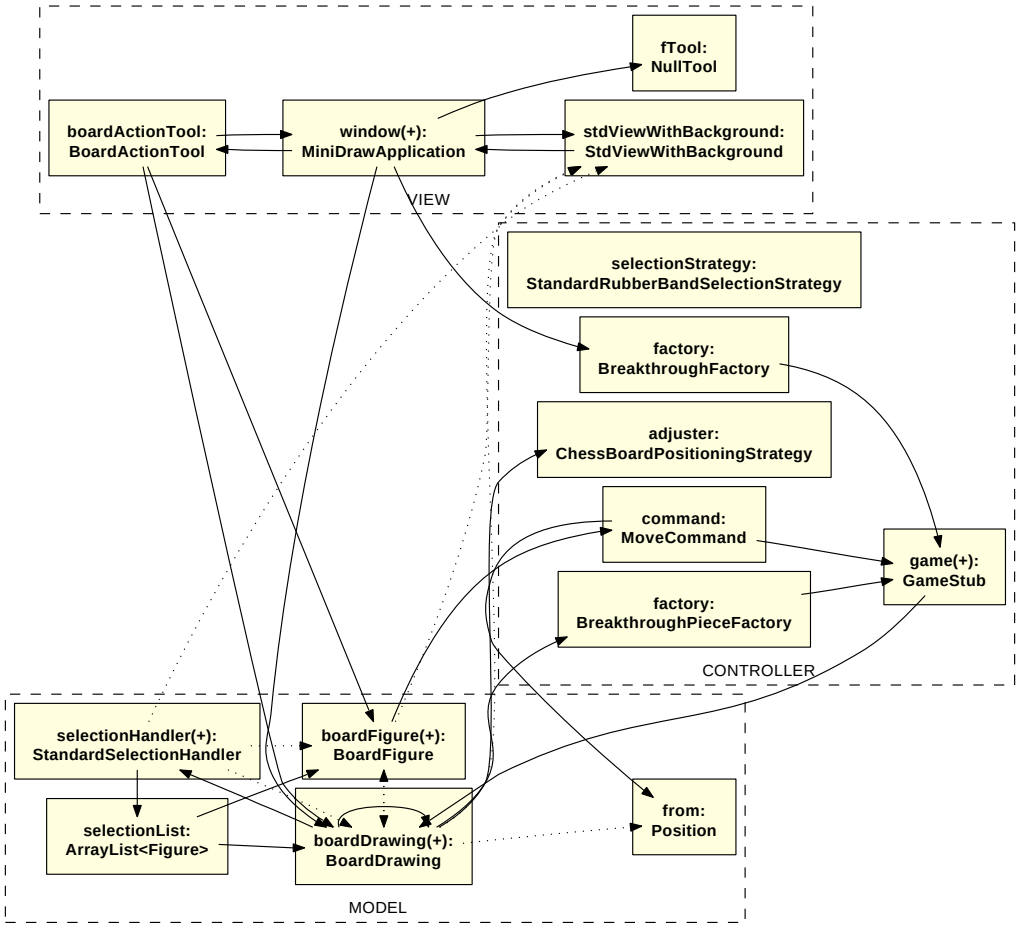


Figure 3.7: An OOG for BreakThrough (OOG3).

`drawing:BoardDrawing`. Also, trackers should not be treated as tools by themselves as they are part of the `SelectionTool` according to the JavaDoc. As a result she required to show the `selectionTool:SelectionTool` object and to push `dragTracker:DragTracker` and `selectAreaTracker:SelectAreaTracker` objects underneath this object. To solve this problem, the architectural extractors annotated these objects with more precise annotations which caused them to be moved to the proper domains. They moved `map:Map<Position, List<Figure>>` to a public domain `MAP`s inside `drawing:BoardDrawing`. Also, they moved `Tracker` objects to a public domain `TRACKERS` inside `tool:SelectionTool`.

Modifying the annotations caused more objects to appear on the OOG, such as

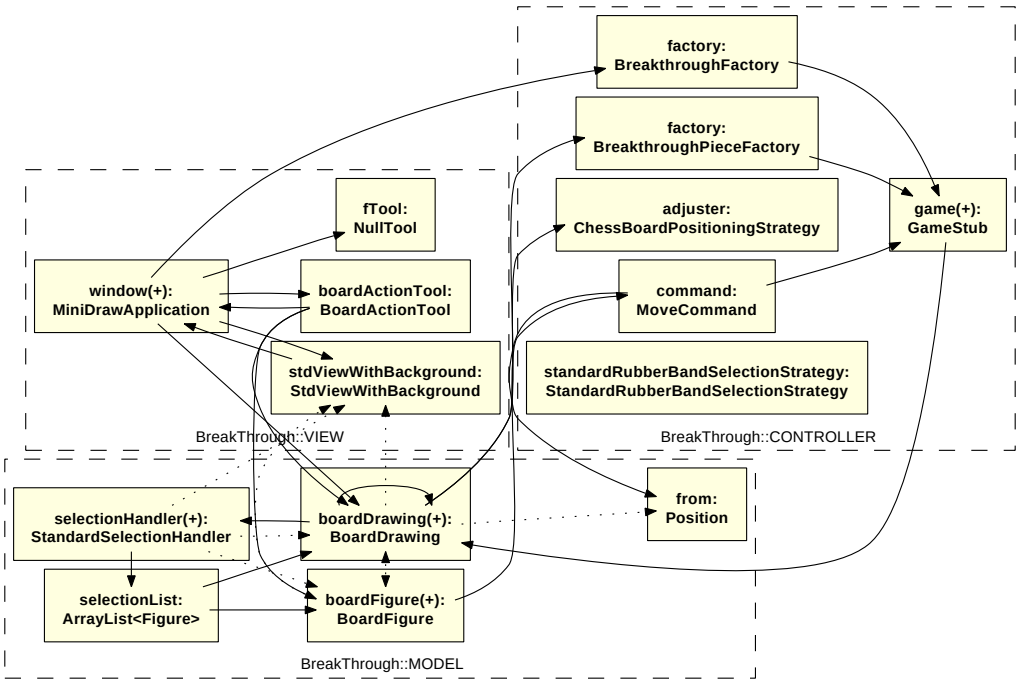


Figure 3.8: An OOG for BreakThrough (OOG4).

`selectionList:ArrayList<Figure>`. The developer wanted this object to be pushed underneath `drawing:BoardDrawing`, since she already knew that each drawing maintains a temporary possibly empty subset of all figures called a selection. She also knew that in MiniDraw, an effort has been made to repartition the `Drawing` interface. According to the new design, some of the drawing responsibilities are expressed in smaller and more fine-grained interfaces. Namely, `DrawingChangeListenerHandler` which defines the management of listeners or observers, and the `selectionHandler` which defines the selection handling responsibility. As a result, she asked the architectural extractors to push `selectionList:ArrayList<Figure>` underneath `selectionHandler:StandardSelectionHandler`, then to move both `selectionHandler:StandardSelectionHandler` and `listnerhandler:DrawingChangeListenerHandler` from the top level domain `CONTROLLER` to a public domain `HANDLERS` underneath `drawing:BoardDrawing`. Also, the above information helped the developer understand that the observer de-

sign pattern is not expressed at the level of the `Drawing` object itself, but at the level of `listenerHandler` which should have the `Listener` list as part of its substructure and not inside the `Drawing` object itself. The OOG was showing that there is a list of listeners inside drawing, due to extra annotations on the implemented interfaces which caused duplication in the extracted OOG. Also, to express more clearly the observer pattern, the listener list should point to `stdViewWithBackground`.

3.4.4 Displaying missing objects

The extracted OOG was missing several useful objects. For example, to avoid `ConcurrentModificationException`s when adding or removing figures to a drawing, one can acquire a lock on the list of figures inside drawing when other threads are potentially iterating over the very same list. The developer found it useful to see the `lockholder:Thread` object inside a private domain `_locks` inside `drawing:BoardDrawing`.

To convince future developers that the extracted OOG is sound, the developer wanted to see every single domain name on the extracted OOG qualified by the name of the declaring type. For example, `window:MiniDrawApp` is of `MiniDrawApp` which is a `DrawingEditor` which is ultimately an `Object`. However, the extracted OOG only shows the `window:MiniDrawApp` instance. To solve this problem, we show objects inside the `owned` domain of `window` qualified by their declaring type. For example, the `imageManager` instance is actually inside `owned::DrawingEditor` not `owned::MiniDrawApp`.

3.4.5 Displaying missing edges

According to the developer, the OOG was missing some important edges. For example, she wanted to see that the `MoveCommand` points to the `Figure`, especially since the `Figure` has an instance of `Command`. Also, after reading

the JavaDoc, she found that `selectAreaTracker:SelectAreaTracker` is a tool to select a set of figures using a rubber-band, and that it has an instance of `StandardRubberbandSelectionStrategy`, which entails that this object should point to `strategy:RubberBandSelectionStrategy` object. Similarly, she found that `Command` instances should point to `Position` instances. Adding more precise annotations to instances of `Command` and trackers caused these objects to have points to edges to instances of their declaring types. Finally, she wanted to see an edge between `stdViewWithBackground` and `imagemanager:ImageManager` inside `window:MiniDrawApp` object.

3.4.6 Merging objects that share a common supertype

Object merging occurs when two or more objects of the same type, i.e., the same declared type or are subtypes of a common supertype, reside in the same domain, which causes these objects to be collapsed as one object on the extracted OOG. Using the abstraction by types, the architectural extractors collapsed all the different instances of `Tool` in one instance, `tool:Tool` to reduce the clutter in the top level architecture. However, the developer reported that for a code modification task, clarity is more important than reducing clutter. For the OOG to be useful, it should explain what exactly happened when the merging of objects occurred. The developer requested one of the following: either show all the different instances of `Tool`, or show these objects in different domains inside `tool:Tool` with domain names qualified by the name of the declaring type, i.e., `BoardActionTool::owned`, `SelectionTool::owned`, etc. This also should be reflected on the traceability to code by showing multiple traceability options whenever the developer attempts to navigate to code from the `tool:Tool` object.

3.4.7 Displaying labeling types

While doing code modifications, the developer noticed that the Template design pattern is being used where you instantiate the application by creating the editor, then you get the drawing, then you create the game instance, then you register the drawing with the specific game instance (`game:gameStub`). The `Drawing` implements the `BoardGameObserver` interface, and the only way to display this on the OOG is by displaying `BoardGameObserver` as a labeling type on the `drawing:BoardDrawing` object. Similarly, it should be clear that `view:StdViewWithBackground` is a `DrawingChangeListener` since it is an observer of changes on a drawing. Also, `Drawing` is a `FigureChangeListener`, since it listens to changes on figures. `Drawing` is also divided into two interfaces:`SelectionHandler` and `DrawingChangeListenerHandler`. We used labeling types to display this information on the OOG.

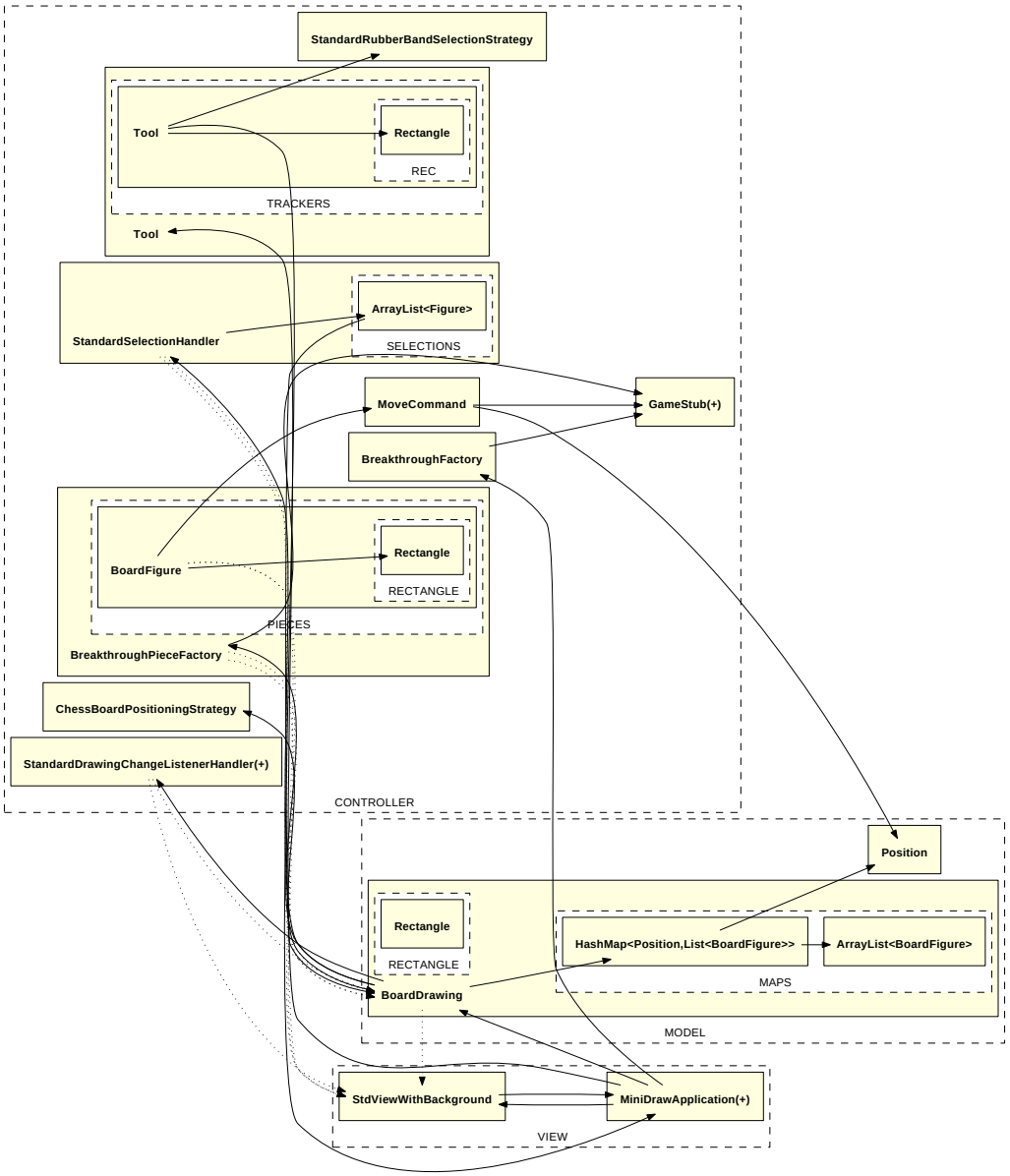


Figure 3.9: An OOG for BreakThrough (OOG5).

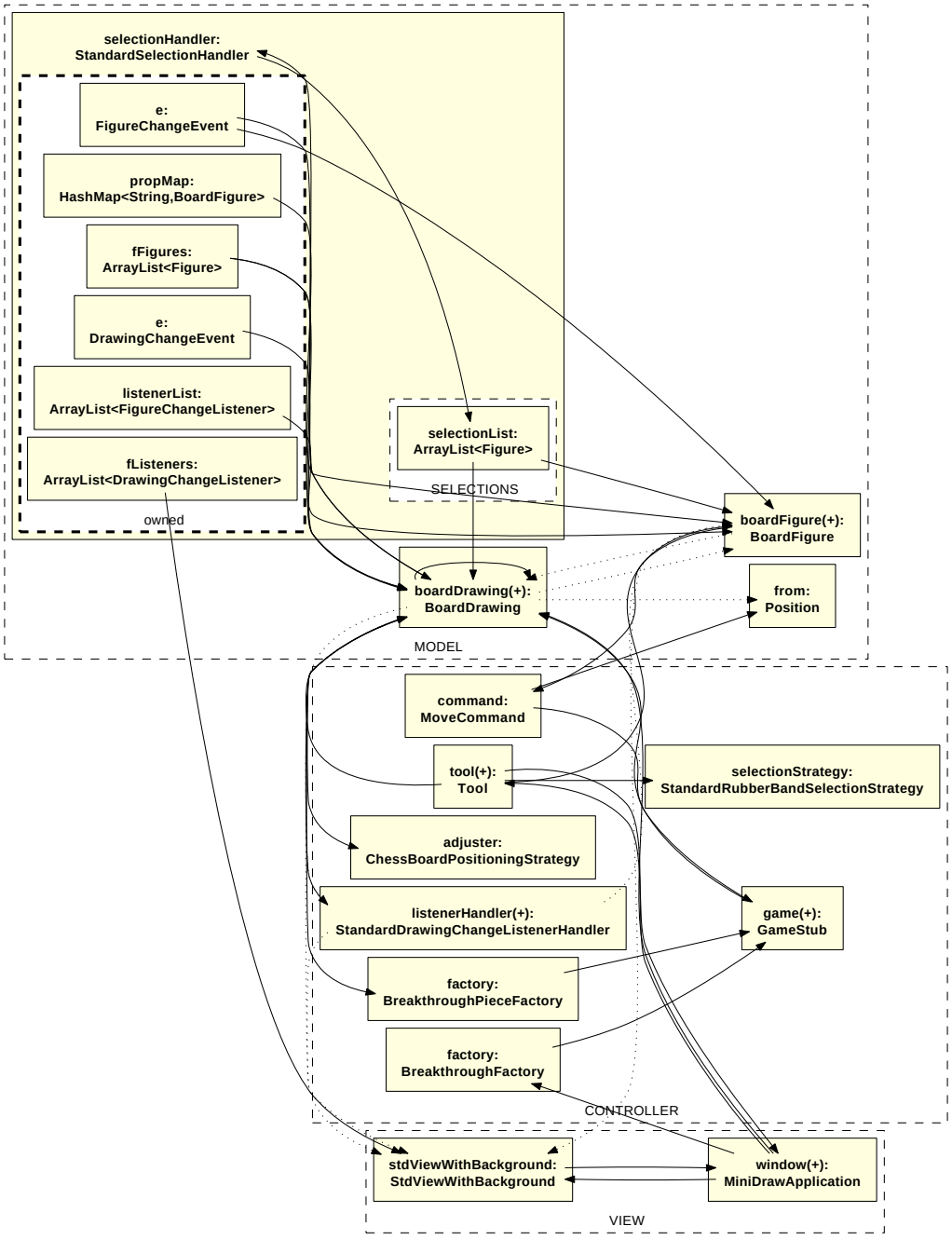


Figure 3.10: An OOG for BreakThrough (OOG6).

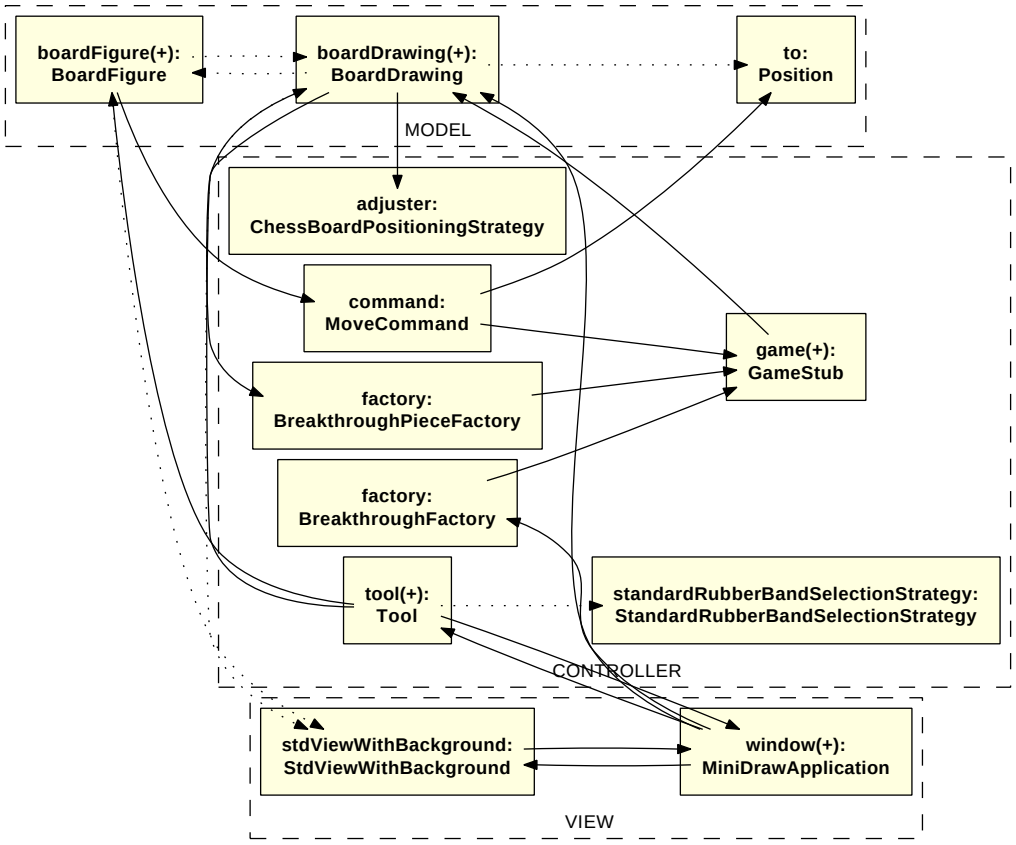


Figure 3.11: An OOG for BreakThrough (OOG7).

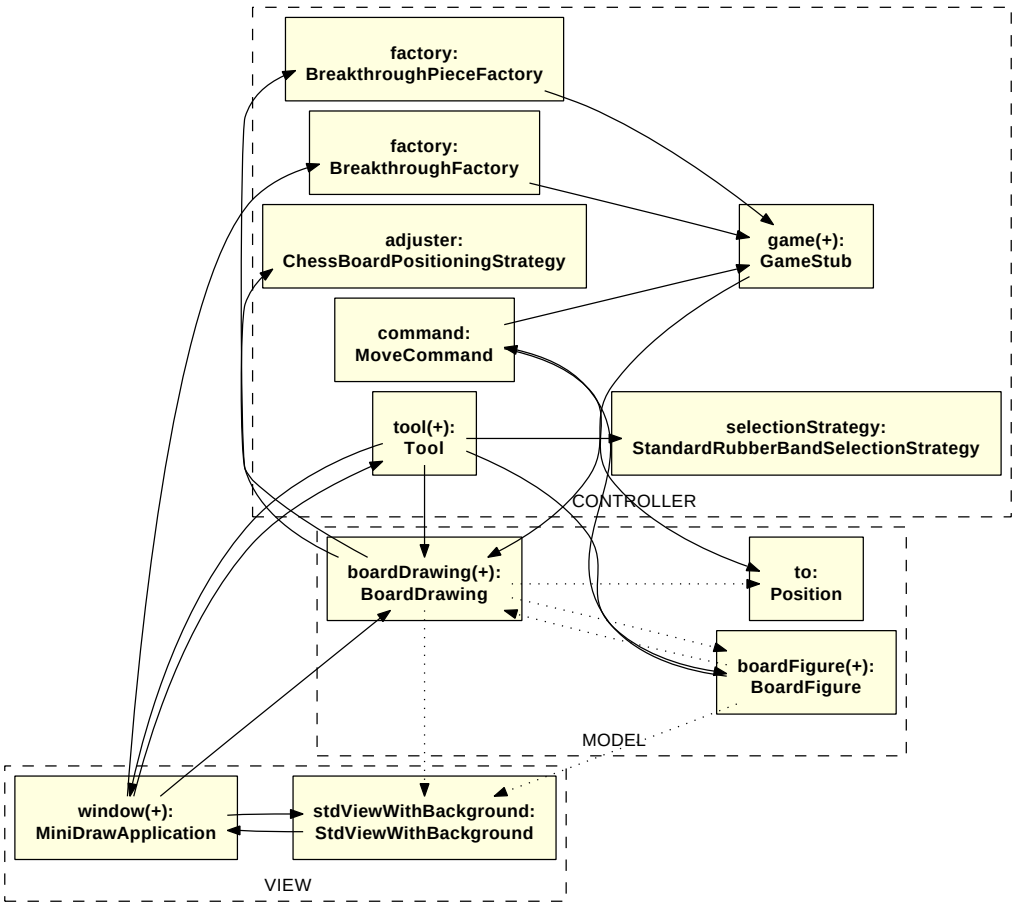


Figure 3.12: An OOG for BreakThrough (OOG8).

3.5 Limitations

In this section, we discuss some of the limitations we encountered while adding annotations to MiniDraw, extracting OOGs, and refining them.

3.5.1 Adding Annotations

We were able to fully annotate MiniDraw, without any issues. There are only 2 remaining warnings, which are due to a minor bug in the type checker.

3.5.2 Extracting OOGs

Limits of static analysis. An OOG is a static sound approximation of the object structures. As a result, it is sometimes less precise than what the developer might want to see. More specifically, if the program instantiates multiple objects of the same type, then each object serves a specific purpose in the program. Therefore, developers want to see all those different objects on the diagram.

In order to show a finite graph for a possibly infinite number of objects, the OOG merges all the objects of a given type inside a domain into one canonical object. Therefore, a developer cannot rely on the OOG to distinguish between several related instances of the same type that are in the same domain. For example, the `p:Position` instance in the top level domain `MODEL` represents multiple instances of the type `Position`. However, the analysis does keep track of all of these allocations. So a developer can use the OOG Viewer to trace to multiple allocations of type `Position` in the code (Figure 3.2). On the other hand, an OOG can distinguish between different instances of the same type that are in different domains. So, if distinguishing between some objects is important, the architectural extractors can place these objects in different domains. For example, the `ArrayList` of `Figure`s declared inside the `owned` domain of `selectionHandler` (Figure 3.10) is distinct from the `ArrayList` of `Figure`s

inside the `SELECTIONS` domain.

Displaying `shared` objects. Library code, such as the Java standard library (JDK), contributes many interesting elements to the MiniDraw OOG, namely, many objects, e.g., `ArrayList`, and many edges, e.g., the edges between `ArrayList` and its contained objects. However, there are a few classes that are not particularly interesting to developers such as `String`, `Font` and `Color`. The architectural extractors typically place these objects in a global `shared` domain that we do not show by default on the extracted OOG. However, the developers can display the `shared` domain on the OOG, even though displaying the domain tends to increase the clutter without providing much useful information.

In MiniDraw, the architectural extractor did not want to reason about some objects, such as `Rectangle` from the AWT Library, so we placed them all in `shared`. There were instances, however, when the developer wanted to see `displayBox:Rectangle` inside `BoardFigure` and `selectionArea` inside `Tool`.

Instantiation-Based View vs. Declaration-Based View The OOG uses a whole-program static analysis in order to extract a global object diagram. As a result, it is not possible to extract the OOG of an incomplete application such as a framework. So we extract OOGs for specific framework instantiations. In the case of the MiniDraw framework, we selected the BreakThrough sample application. However, this makes it challenging to deal with objects that are instantiated in one framework application, but not in the other. For example, in RectangleDraw, a user can use the selection tool to select a certain shape and drag it using trackers. In the case of `BreakThrough`, this is not applicable. Therefore, there is no instance of `SelectionTool` inside the `BreakThrough` class.

At some point, the developer suggested pushing the `dragTracker:DragTracker`, and `selectAreaTracker:SelectAreaTracker` objects underneath

`tool:SelectionTool`. The architectural extractor pushed the `dragTracker` objects underneath the `tool` object. Since no instance of type `SelectionTool` was created inside `BreakThrough`, neither the `tool` nor the `dragTracker` objects appeared on the OOG (the OOG displays child objects or child domains only if their corresponding parent object is displayed). Indeed, the OOG static analysis uses an instantiation-based view which considers only object creation expressions [1]. To get these objects to appear on the OOG, the architectural extractor explicitly declared the `tool:SelectionTool` instance inside the `BreakThrough` class.

As a result, it may be a good idea to also support a declaration-based view in addition to an instantiation-based view [1]. A declaration-based view, however, is likely to be less precise than an instantiation-based view. We also have a similar issue with objects that are allocated inside library code instead of framework or application code. Those have to be summarized with a special annotation that indicates an object allocation for the instantiation-based view.

No points-to edges to instances within method bodies. The developer wanted to see that `tool:BoardActionTool` points to `command` since `BoardActionTool` calls the method `command.exectue()` when a user drags a figure on the board. However, the OOG currently shows only points-to edges, which correspond to field references. So, these instances within method bodies will not have any edges. One solution, however, is to add a virtual field inside the `Command` class, to cause the edge between `command` and `position` to appear.

A better solution is to show more than points-to edges on the OOG. Points-to edges focus on persistent reference relations between objects on the heap, rather than transient relations on the stack. Developers typically need to see richer edge information between objects. One solution is to also show usage edges, which show dataflow communication [42, 41].

Traceability to Library Code. Edges from the children of an object get lifted to the parent object whenever the substructure of the parent object is collapsed. When the children objects contain data structures such as lists, this hides the data structures from the top level but summarizes the interesting relations. For example, the OOG shows that `boardDrawing` has a points-to dotted edge to `stdViewWithBackground`. But this edge is not due to a field declaration of type DrawingView inside BoardDrawing. This edge can be explained by expanding the substructure of `boardDrawing` and then expanding the `listenerHandler`. Doing so eliminates the lifted edge and shows a real points-to edge from a list, `fListeners` inside a domain of `listenerHandler`, which is an `ArrayList` of `DrawingChangeListener` objects, which in turn , points to a `stdViewWithBackground` object in the `VIEW` domain.

The class `ArrayList` is in the Java Standard Library. The current annotation system uses external files to annotate portions of the library code in use [1]. In particular, the edge from an `ArrayList` object to its containing elements is indicated in a virtual field in one of these external files. Having these external files interferes with the traceability to code because the ArchRecJ tool would have to trace to the underlying library code. As a result, developers using ArchRecJ cannot trace from some edges associated with containers like `ArrayList` objects to meaningful lines of code. We believe that making ArchRecJ trace to some virtual field in an external file would be less useful to developers. One way to solve the problem with the traceability feature could involve attaching the library sources to ArchRecJ, as must be done today in the Eclipse IDE in order to step into library code while using the debugger, to avoid stepping into a binary class file instead of a Java source file.

3.5.3 Refining the Extracted OOGs

There were cases where the developer attempted to refine the OOG on her own, so she changed a few annotations in the code. But the type checker produced several

warnings, because the changes she wanted were not supported by the as-built the code or were limited by the expressiveness of the type system. In those cases, she reverted her changes to eliminate the annotation warnings.

Several of the key expressiveness limitations in SCHOLIA are due to the single ownership model and the lack of ownership transfer [1]. An object is in exactly one domain, which does not change at run-time. Furthermore, the code must respect the assignment rule, i.e., a variable in domain A cannot be assigned to another in domain B.

An object cannot be placed in two different domains simultaneously. The developer wanted to create a new private domain `SUBTOOLS` inside `tool:SelectionTool` to hold the `nullTool` and `aChild` instances. She wanted to do that since a JavaDoc comment states that there is an invariant that these objects should never be set to null. However, these objects are assigned to tracker objects which already reside in the `TRACKERS` domain. The type checker prohibits them from being in two domains at the same time and thus produces an annotation warning. Similarly, the developer complained about not having a direct relation between `gameStub` and `drawing`, even though a `drawing` is an observer of the `game`. She wanted to add a public domain `GAME_SUBS` inside `gameStub` and put a drawing instance inside it or at least show that a drawing has a points to edge to `gameStub`. However, the type checker complained about having the drawing instance in two different domains.

Some labeling types could not be selected interactively. The developer wanted to add some labeling type to several objects on the OOG. For example, she wanted to see the listener interfaces `MouseListener`, `MouseMotionListener`, and `KeyListener` on the `stdViewWithBackground` object. However, in some cases, the labeling types feature in ArchRecJ did not show all the possible types among

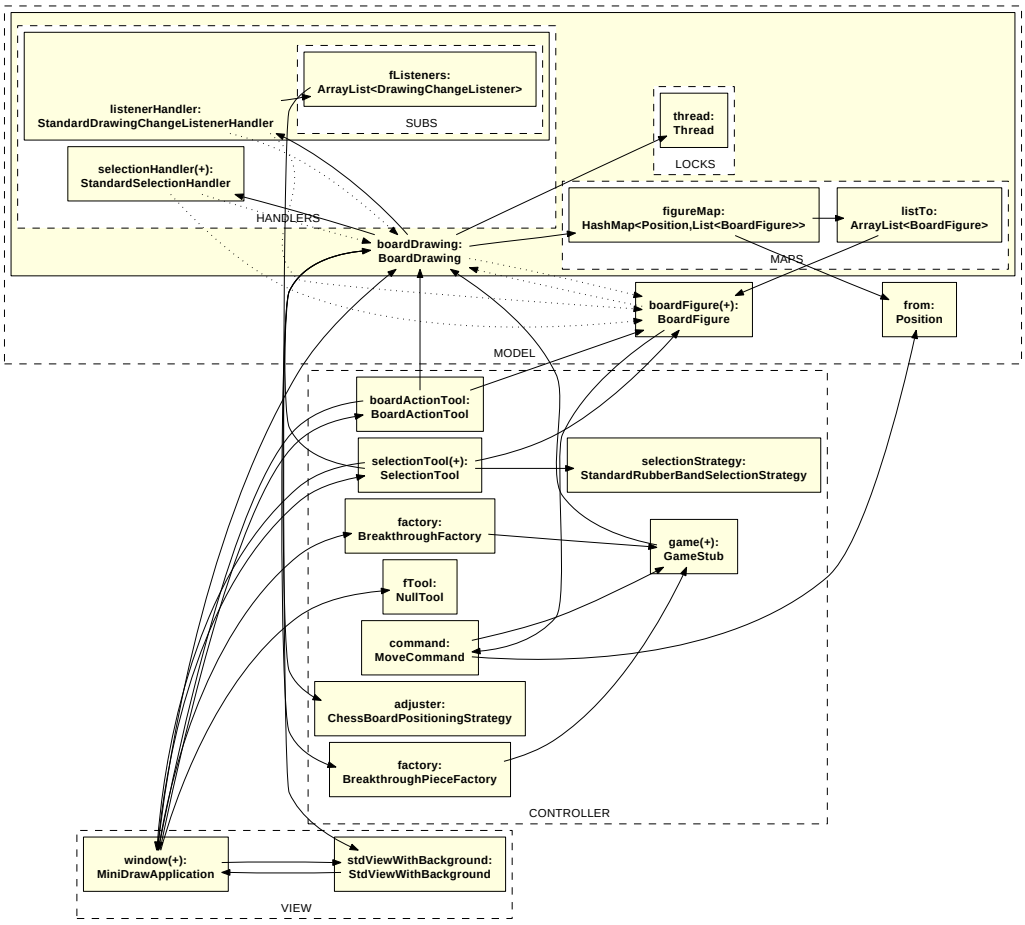


Figure 3.13: An OOG for BreakThrough (OOG9).

the suggested labeling types, due to a minor bug in the tool. In this case, the class `StdViewWithBackground` implemented these interfaces. There was an easy workaround in that the developer could edit an external file with the list of labeling types, which the static analysis then applied to the OOG.

Choosing the right root class. MiniDraw is a framework, i.e. an incomplete application. The developer had access to several applications implemented using MiniDraw including Breakthrough. Therefore, she was confused about what should be considered the main class or the root object in the case of MiniDraw. Her justification was that Framework is an incomplete application, but it has a default implementation. She thought that `MiniDrawApp` should be considered the main class

instead of `BreakThrough` since it is a `DrawingEditor` and the editor must instantiate all parts of the application.

Using public domains is hard. The developer wanted to use public domains heavily on the extracted OOG. However, this was not possible in a few cases due to how the code was written, and the assignment rule. For example, the developer wanted to place `drawingView` in a public domain `SUBS` inside `Drawing`. However, `drawingView` was also in the `UI` domain, and this would not type check.

Also, a figure is created using a factory class which suggests putting figures in a public domain `PIECES` inside `BreakThroughPieceFactory`. The architectural extractor decided against this because a factory typically does not own any of the objects it creates. Alternatively, the developer thought about placing figures in a `DRW_MODELS` public domain inside `drawing`, since `drawing` has the map from positions to figures. But placing objects inside a public of an object indicates logical containment, a loose form of encapsulation. In this case, `figure` objects had to be freely accessible by many other objects in the system.

As another example, she considered that the object `figure:BoardFigure` should not appear in the top-level domain `MODEL`. After looking at the code, the developer found that there are no figures at the application level, since you instantiate the application by creating the `editor`, using a specific `factory` where you create the `drawing`. The `drawing` then creates the `figure`s using another factory object (`factory:BreakThroughPieceFactory`) which is in turn referenced by the root object (`root:BreakThrough`). So, the developer wanted the object hierarchy to reflect the order of creation which means that `figure:BoardFigure` should be pushed into a public domain `PIECES` inside `factory:BreakThroughPieceFactory`. However, the architectural extractors insisted that the factory creates unique `figure` objects that then flow into other domains inside other objects.

Following this line of reasoning, she thought that the OOG may be duplicating some information. For example, the OOG shows an instance `figure` inside a `drawing` which is true since a `drawing` is composed of `figure`s.

3.6 Discussion

3.6.1 Refinement Bias

The architectural extractors use hierarchy to convey architectural importance. They keep architecturally relevant objects at a higher level than less important objects to reduce clutter in the OOG. Therefore, the OOG may hide low-level objects that are implementation details, such as data structures. Also, the OOG often does not display the `shared` domain which contains uninteresting objects such as `java.lang.String`. Reverse-engineered class diagrams tend to show low-level types (Figure E.9).

We justify the refinement step because the architectural extractors, in this case, were not experts in the subject system, so the extracted OOG did not fully convey the intent of MiniDraw's original designers. On the other hand, the developer understood the code better since she read a book about MiniDraw and performed a few modifications on the system. As a result, the developer was better placed to provide feedback to the architectural extractors and help them refine the OOG to reflect the design intent in the system.

In any case, neither the architectural extractors nor the developers have the ability to draw whatever they like on the OOG, because the desired annotations must type check at all times, so the type checker keeps them honest. Indeed, Section 3.5 identified that there were some desired changes that the code did not support, and led to annotation warnings. Further, they do not even control the layout, the sizes, or the colors of the nodes on the OOG. They can only refine the extracted diagram using two types of abstraction: (1) abstraction by ownership, by changing the own-

ership annotations in the code, or (2) abstraction by types, by controlling the static analysis. They can also set various display options like labeling types. And they can control the level of visual detail by collapse or expand individual objects to highlight them.

3.6.2 The OOG as a Global, Rather than a Task-Specific View

Most approaches generate only task-specific views. It is even generally understood that the suitability of a visualization is always task-dependent. After we extract OOGs, we refine them by adding ownership domain annotations to encode the design intent. Therefore, an OOG is intended as global view of the system, akin to a global architectural diagram of the system. We believe this is consistent with the need for global class diagrams that many developers typically generate semi-automatically. These diagrams tend to differ from automatically extracted ones in that they leave out low-level classes. A fundamental design choice in the Scholia approach is to extract an object graph which shows all the objects in the system, and their points-to relations. Hierarchy is crucial to make the visualization scale. Developers who want highly precise information within a method in which they are working can use a shape analysis to show the input and the output shape graphs [12]. Of course, scaling the graphs to the entire program will likely generate large shape graphs.

3.6.3 OOG vs. Object Diagram

In this section we discuss some differences between OOGs and object diagrams which may have caused some confusion to developers. Many object diagrams do show multiple instances of the same object (Figure 3.15). We extract a global object diagram using static analysis, and the static analysis runs into decidability issues as

to how many instances of some entity are created at runtime, or the specific values of the object's fields (such as x = 6.1 in Figure 3.15). When using object diagrams to model a few objects manually, such issues do not occur. However, when a developer is dealing with global diagrams showing all the objects in the system, the object diagram visualization does not scale. As a result, when we extract an OOG, we care that it is at an adequate level of abstraction for developers, i.e., an OOG is not too abstract to the point of being useless. Thus, an OOG is more abstract than other typical, hand-drawn object diagram, yet, developers still find it useful.

3.6.4 Other diagrams

For the sake of comparison, we also used other automated tools that extract object graphs without requiring annotations. Such tools extract flat object graphs. For instance, we used Womble [26] to extract a flat graph for MiniDraw (Figure 3.16). Even for a small system like MiniDraw, such a diagram does not seem very readable, nor does it convey much design intent. Since the flat graph does not seem very usable, it may be worthwhile to do all the work to extract a hierarchical object graph, including adding annotations, extracting OOGs, and refining the extracted OOGs. It would be even more worthwhile if other developers find the OOG useful while they are doing code modification tasks on the system.

3.7 Summary

In this chapter, we discussed how we produced OOGs for a small system, MiniDraw. We also discussed how we made the diagram potentially useful by having a developer (the author of the thesis) rely on the diagram while she performed code modification tasks on the system. We believe that such an OOG could be also useful to other developers evolving the MiniDraw system. To answer this research question,

we conducted a controlled experiment in which we recruited outside participants who were not involved with the production of the OOGs. We report on this experiment in the next chapters.

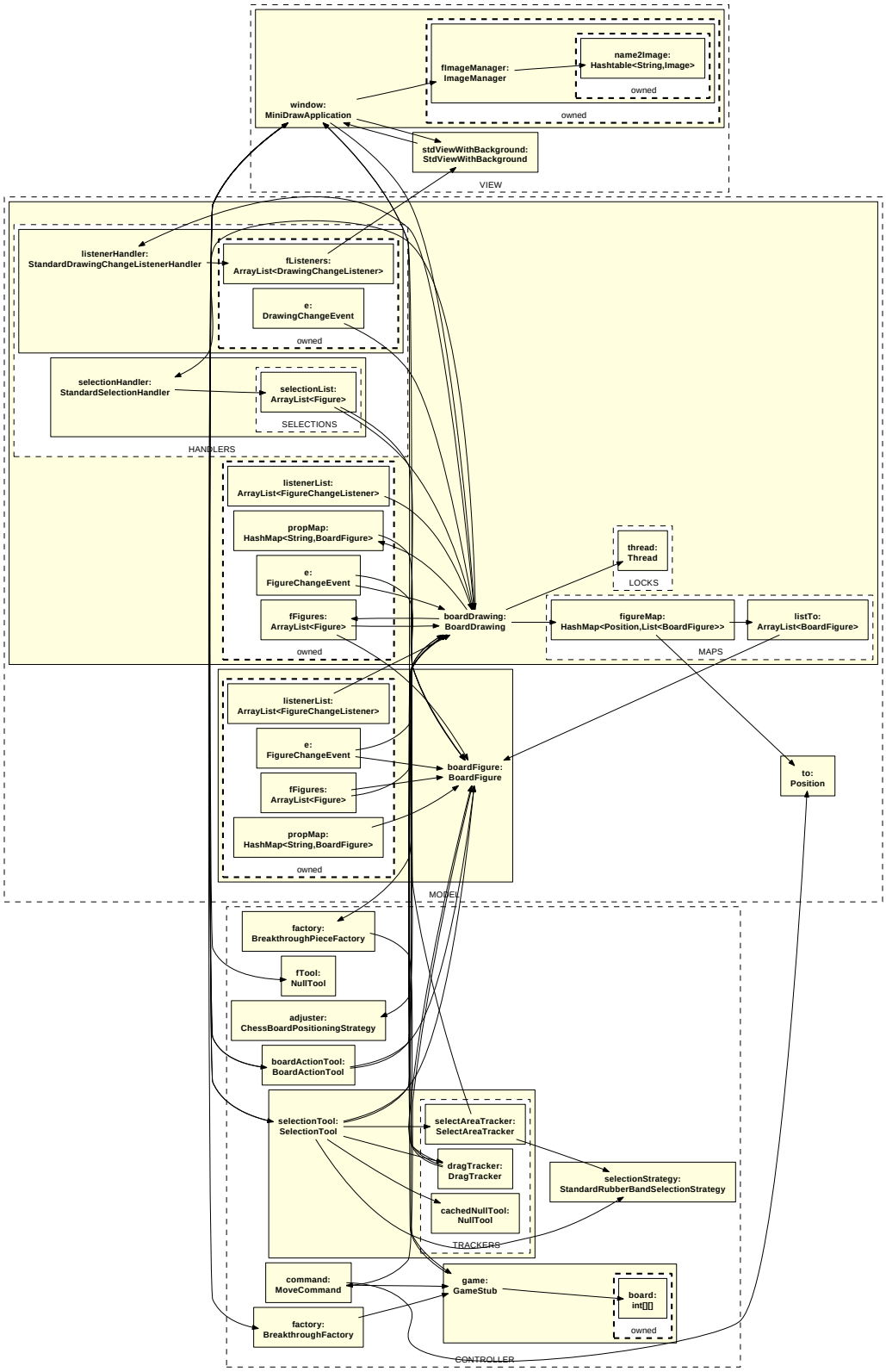


Figure 3.14: An OOG for BreakThrough (OOG10).

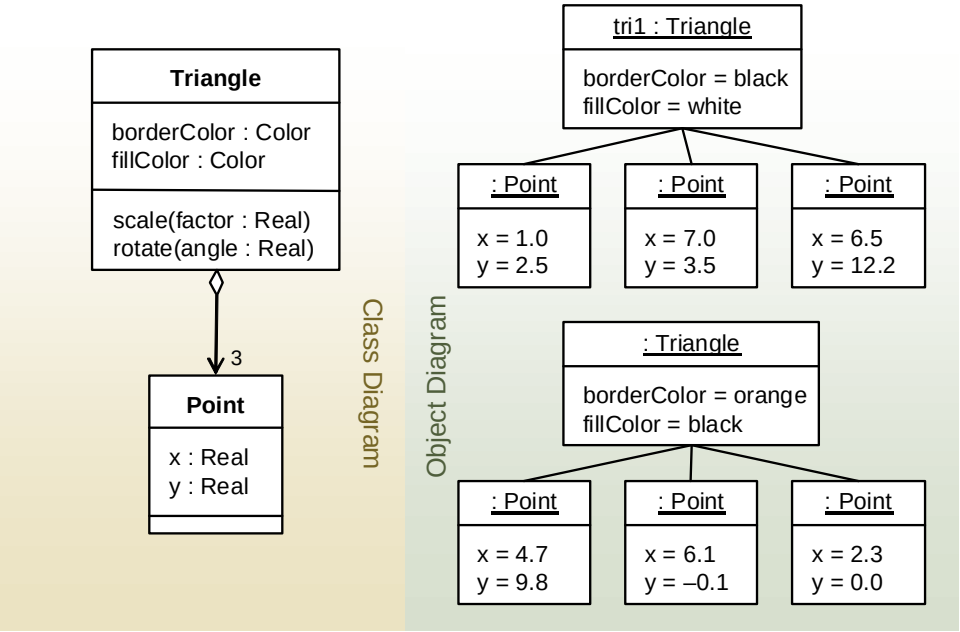


Figure 3.15: Example of an object diagram.

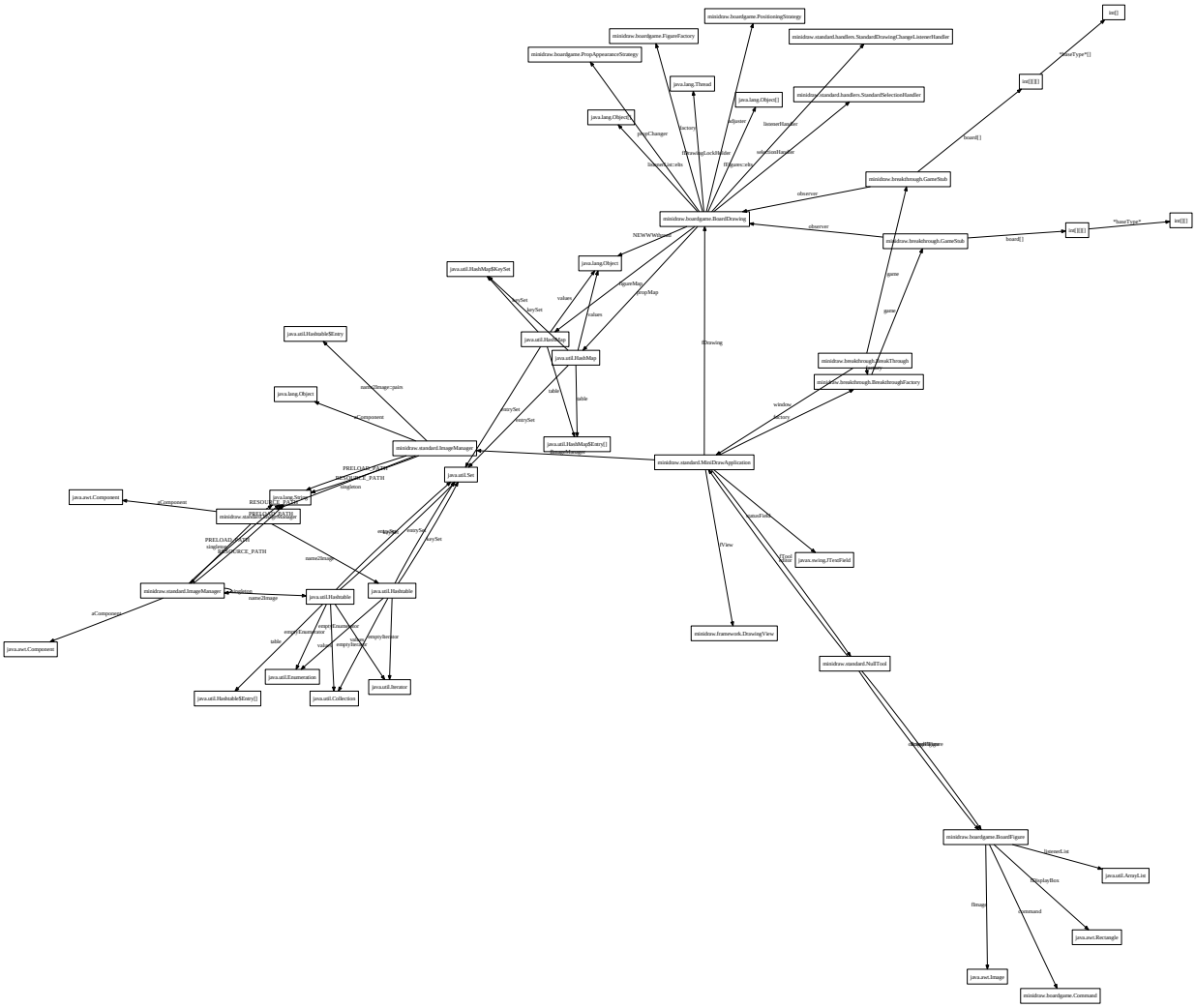


Figure 3.16: A flat object graph for MiniDraw, extracted using Womble [26].

Chapter 4: Controlled Experiment

After extracting an OOG for BreakThrough, conducted a controlled a controlled experiment to measure the usefulness of the extracted diagram to external developers for code modifications. In this chapter, we explain the experimental design. We analyze and discuss our results in Chapter 5.

4.1 Subject System

For the experiment, we used the BreakThrough game implemented using the MiniDraw framework, and for which we extracted an OOG (Chapter 3). BreakThrough is a two-person game played on an 8x8 chessboard. Each player has 16 pieces, initially positioned on the two rows nearest to the player (Fig 4.1). The winner is the first player to move one of his pieces to the home row, that is, the row farthest from the player. Players alternate to take turn. The white player begins.

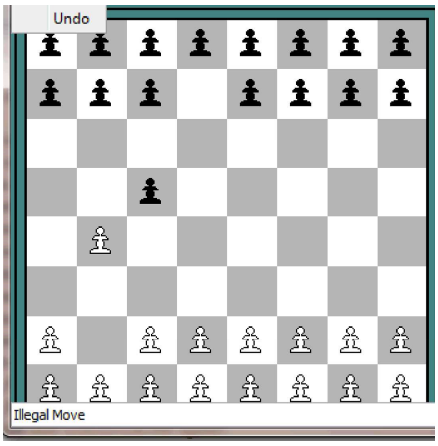


Figure 4.1: The BreakThrough board game application implemented using MiniDraw.

The BreakThrough implementation we gave to our participants had a drawing of the board with the pieces on it, but was missing the game logic and we asked the participants to reuse the framework instead of introducing new concepts, e.g. player. The BreakThrough code was mostly core framework classes

(e.g. `MiniDrawApplication`) and interfaces organized into three main packages: `minidraw.framework`, `minidraw.standard`, `minidraw.standard.handlers`. The `minidraw.boardgame` package contained code that is related to board games in general, (e.g. `BoardDrawing`) and `minidraw.breakthrough` package had classes to handle the BreakThrough game in particular (e.g. `BreakThrough`).

4.2 Task Design

We designed three tasks which we thought were difficult to do just by looking at the code or at the UML class diagrams. The tasks involved knowledge related to frameworks and design patterns. MiniDraw has variability points that allow it to be customized with regards to the figures to show, the set of images to load, the tools that affect the figures, the type of view and drawing and view to use, and the ability to make objects observe figure state changes. Therefore, we selected tasks that focused on MiniDraw variability points.

To verify that the tasks required knowledge about the object structures, we tested the tasks on ourselves, and conducted a pilot study with one developer. The purpose of the pilot was to see if the instructions were clear and if the tasks involved using the OOG and the OOG Viewer. We analyzed the data gathered during the pilot according to our coding model (Table 1.1) and ensured that at least some of the questions that were raised involved the object structures. We dropped a fourth task which did not involve any of these questions. We selected two bug fixes and one feature request:

- **Task 1: Bug Fix: Implement validation on the piece movement.** There is no validation of the move. For example, a piece can move to a non-empty square. The rules of movement are simple. A piece may move one square straight or diagonally forward towards the home row if that square is empty. A

piece, however, may only capture an opponent piece diagonally. When capturing, the opponent piece is removed from the board and the player's piece takes its position, as in the chess game.

- **Task 2: Bug Fix: Implement the capture of a piece.** A piece may only capture an opponent piece diagonally. When capturing, the opponent piece is removed from the board and the player's piece takes its position, as in chess.

- **Task 3: Feature Request: Implement an undo feature to allow a player to undo a movement.** Show a menu bar with a menu item "Undo move". In the cases where the move cannot be undone (e.g., a piece is captured), display a message on the status bar.

Our tasks could be solved in multiple ways and were not contrived to the point of requiring very specific tool support. Because we fine-tuned the task design after the pilot, we excluded the pilot data from our results. Task 1 and Task 2 are related. In order to validate the movement of a piece for Task 1, the constraints on direction were related to Task 2 where you can only move a piece diagonally in the case of capturing. This may have been an issue which we discuss in Chapter 7.

4.3 Experimental Design

In order to validate or reject our hypotheses, we followed the *between-subjects design* [45] by having two groups of participants solve the same tasks. One of these groups (*control group*) worked with standard Eclipse navigation features and had access to diagrams of only the code structure, i.e. class diagrams, while the other group (*experimental group*) additionally used OOGs and a viewer to visualize and interactively navigate the OOG. In the rest of this thesis, we refer to the control group participants as the *C participants* and the experimental group participants as the *E Participants*. We use the term *experimenter* to refer to the person who was

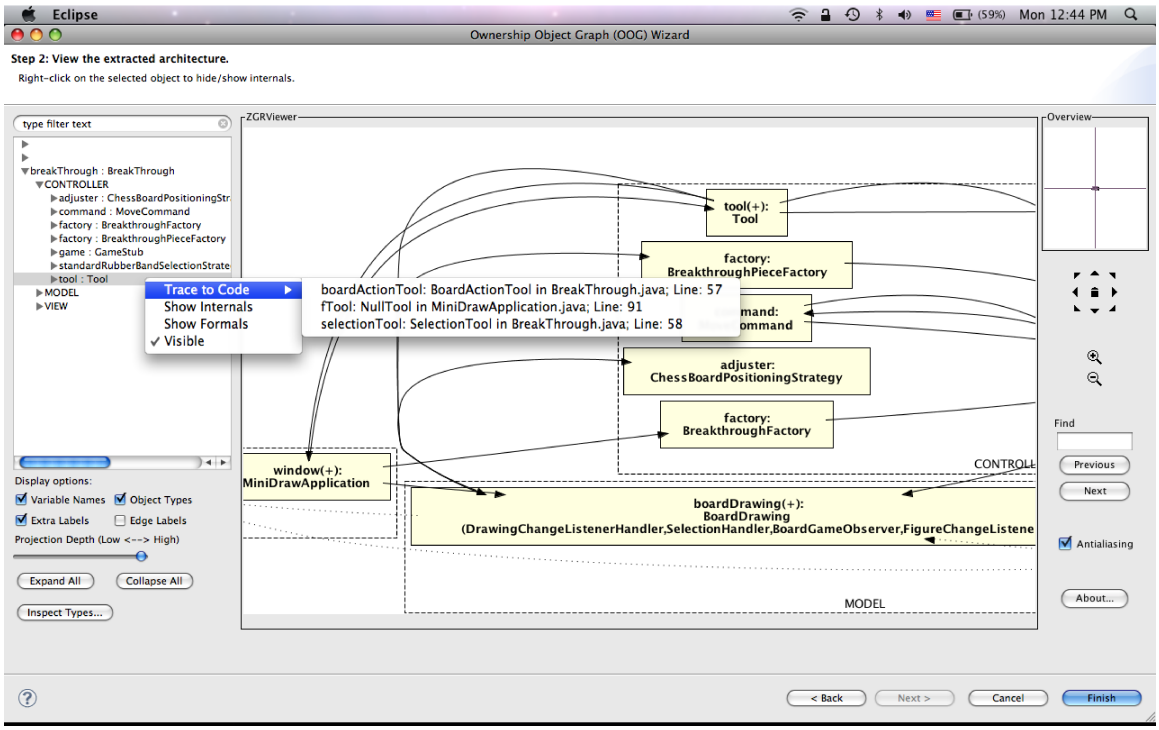


Figure 4.2: The OOG Viewer used to interactively navigate the OOG.

running the study and observing the participants while performing the tasks (the author of this thesis).

The **independent variable** in our experiment was having access to diagrams of the run-time structure (OOGs) in addition to other tools. In order to answer our research question, we used the following **dependent variables** to measure the effect of controlled variation of the independent variable:

1. The number of code elements explored to complete a task.

2. The time to task completion.

3. The success on a task.

Material provided to the C participants. We provided C participants with UML class diagrams adopted from the original designers [13]. We also provided them

Table 4.1: Different versions of class diagrams provided to participants. These diagrams appear in Appendix E: CD1 (Figure E.4), CD2 (Figure E.5), CD3 (Figure E.6), CD4 (Figure E.7), CD5 (Figure E.8), CD6 (Figure E.9), CD7 (Figure B.1), CD8 (Figure B.2), CD9 (Figure B.3), and CD10 (Figure B.4).

Diagram	Source	Purpose
CD1	Reverse engineered using AgileJ	classes in the `minidraw.boardgame` package
CD2	Reverse engineered using AgileJ	classes in `minidraw.breakthrough` package
CD3	Reverse engineered using AgileJ	classes in `minidraw.framework` package
CD4	Reverse engineered using AgileJ	classes in `minidraw.standard` package
CD5	Reverse engineered using AgileJ	classes in `minidraw.standard.handlers` package
CD6	Reverse engineered using AgileJ	class dependencies on the main class `BreakThrough`
CD7	UML class diagram	classes in the `MODEL` part
CD8	UML class diagram	classes in the `VIEW` part
CD9	UML class diagram	classes in the `CONTROLLER` part
CD10	UML class diagram	classes related to the `DrawingEditor`

with different class diagrams reverse-engineered using AgileJ [8] (Table 4.1). In the rest of this thesis, we refer to these diagrams by their numbers.

Material provided to the E participants. We provided the E group with the same material provided to the C group. Also, we provided the participants in this group with a tutorial that explained the basic OOG notation and covered the important features of the OOG Viewer (Appendix F). We also provided them with examples of tasks or activities done with the OOG Viewer in the form of text and hands-on exercises. We summarize the exercises in Table 4.6. We provided the E group with two OOGs, an abstract graph that shows the different types of tools collapsed into one object(Figures E.1) and another, less abstract one (Figure E.2). We also provided them with XML versions of the extracted OOGs to be able to load them in the OOG Viewer.

Both groups were provided with an instruction sheet. The instruction sheet had an appendix to clarify the notation for OOGs, UML and AgileJ. The appendix also included code snippets on how to use Swing since we did not assume familiarity with Swing (Appendix D).

4.4 Participants

We advertised the study around the Computer Science Department at Wayne State University using flyers, mailing lists, and through word of mouth. We had 14 respondents, which we pre-screened for 1 hour each. The pre-screening session included three parts: 1) completing a registration form; 2) answering a quiz on basic Java concepts; and 3) reasoning about a problem then implementing an object-oriented solution using Eclipse. We used the registration form to classify respondents into professional programmers or students. The form gathered background and demographic information such as experience and time in a team. The quiz helped us ensure that the respondents were qualified for the study. The quiz consisted of multiple choice and free response questions. The multiple choice provided faster, standardized response, whereas the free response required more time and details, and was open-ended. We used this test to filter out non-professional Java developers. The final test verified a basic knowledge of object-oriented concepts (Appendix A). We also asked the respondents to use Eclipse to implement their solution to test their familiarity with Eclipse.

Based on the pre-screening, we selected 10 participants (Table 4.2): 4 professional programmers, 3 Ph.D. students in their 4th year, 2 M.S. students, and 1 senior undergraduate. All participants were familiar with Eclipse with a median of 8.5 years of programming experience. Most of the participants reported that they spend from 10 to 20 hours of programming per week. The participants had a median of 4 years of experience coding in Java. All participants were familiar with UML. All except one claimed that they were familiar with frameworks and design patterns.

Table 4.2: Participants' self-reported experience using 1-5 scale (1 is beginner; 5 is expert). *P* refers to the participant. *Yrs Exp.* is the number of years of programming experience. *Ind. Exp.* is the number of years of industrial experience. *Hrs/wk* is the number of hours per week they typically spend programming. *LOC* is the longest program (Lines of Code) they wrote. *Yrs Java, C#, C++* refer the number of years of Java, C#, C++ programming experience, respectively. *Eclipse*, *UML*, *Fwks/Ptrns*, *Swing* indicate their familiarity with Eclipse, UML, Frameworks/Design Patterns, and Swing, respectively.

P	**Yrs Exp.**	**Ind. Exp.**	**Hrs/ wk**	**LOC**	**Yrs Java**	**Yrs C#**	**Yrs C++**	**Eclipse**	**UML**	**Fwks/ Ptrns**	**Swing**
C1	4	0 (Ph.D.)	15	10KLOC	5	2	4	3	Yes	Yes	1
C2	20	6 (Ph.D.)	2–5	$\geq$ 2KLOC	8	2	11	5	Yes	Yes	3
C3	9	4 (M.Sc.)	20	5KLOC	2	3	7	3	Yes	No	1
C4	4	0 (Ph.D.)	20	2KLOC	4	$\leq$ 1	4	3	Yes	Yes	1
C5	6	0 (B.S.)	5–20	2KLOC	3	0	4	3	Yes	Yes	1
E1	3	2 (M.Sc.)	10–20	1-1.5 KLOC	1	0	6	3	Yes	Yes	1
E2	8	0.5 (Ph.D.)	20	1.5 KlOC	4	1	2	5	Yes	Yes	3
E3	25	20 (M.Sc.)	40–50	500 KLOC	5	4	15	5	Yes	Yes	5
E4	24	20 (Ph.D.)	10	10KLOC	10	2	0	5	Yes	Yes	3
E5	10	2 (B.S.)	4–12	7KLOC	3	3	5	3	Yes	Yes	1

4.5 Tools and Instrumentation

For each participant, we recorded audio and the screen using Camtasia. We provided the E participants with the OOG Viewer tool (Fig. 4.2). The OOG Viewer has several features that enables developers interactively navigate the OOG (Table 4.3). In the rest of this chapter, we refer to these features by their numbers.

Table 4.3: The OOG viewer tool features used by the participants during the experiment.

No.	**Feature**	**Description**
Ov.F1	Search Ownership Hierarchy	search for an object in the ownership tree by type or field name.
Ov.F2	Trace To Code	trace from an object or edge on the graph to the corresponding lines of code.
Ov.F3	Examine incoming/outgoing edges	By double clicking an object on the graph, developers can view all objects interacting with it.
Ov.F4	Collapse/expand internals	collapse or expand the sub-structure of a selected object either on the graph or in the tree.
Ov.F5	Navigate	zoom in or out, pan, scroll, etc.

We provided our participants with the Eclipse IDE (version 5.3), and allowed them to use several of the standard features in Eclipse (Table 4.4). We tutored the participants on all these features before they changed the code using hands-on

exercises. In the rest of this chapter, we refer to these features by their numbers.

Table 4.4: Eclipse features used by the participants during the experiment.

No.	Feature	Description
Ec.F1	Type hierarchy	view the classes that inherit from a class or implement an interface.
Ec.F2	Call hierarchy	view all the methods that call another method in a hierarchy
Ec.F3	References in project	search for all the classes that reference a certain type.
Ec.F4	Declarations in project	search for all the classes that declare instances of a certain type.
Ec.F5	File search	search for a keyword or string
Ec.F6	Code hinting	access methods and fields of a class through its object.
Ec.F7	JavaDoc	hovering on a certain element to view documentation.
Ec.F8	Package explorer	the organization of classes (files) into packages.
Ec.F9	Open Type	open a class by typing its name in a dialog box

4.6 Procedure

Table 4.5 provides an overview of our experimental procedure. The experimenter started with a brief introduction on the subject system (Appendix B). Then she allowed the participants to interactively tutor the main navigation features in Eclipse using hands-on exercises on MiniDraw (Appendix C). This part took 5 minutes.

The concepts of OOGs and ownership are not general knowledge. Therefore, the experimenter gave the E group a 20-minute tutorial, explaining the new terminology to enable them to understand an OOG (Appendix F). During the tutorial, she presented the OOG Viewer. She loaded an interactive version of the MiniDraw OOG in the viewer to explain the graphical notation of an OOG in general without providing any hints about the object structure of MiniDraw. During the 20 minutes, she tutored the participants on the OOG Viewer by having them do simple hands-on exercises to measure their understanding of the OOG (Table 4.6). The experimenter answered the participants questions as they arose.

In the remaining 2.5 hours, the experimenter asked the participants in both groups to read the instructions sheet and perform the tasks (Appendix D). The participants were given ample time to plan their changes before making them, and document their

Table 4.5: Overview of the experimental procedure (3-hour session). The C participants received the OOG and the OOG Viewer tutorials during the last 20 minutes.

Part I (25 min)	Introduction (Appendix B)	2 min
	Eclipse Tutorial (Appendix C)	3 min
	OOG Tutorial (Appendix F)	10 min
	OOG Viewer Tutorial (Table 4.6)	10 min
Part II (2.5 hours) (Appendix D)	Plans	Implementations
	Task 1	Task 1
	Questionnaire	Questionnaire
	Task 2	Task 2
	Questionnaire	Questionnaire
	Task 3	Task 3
	Questionnaire	Questionnaire
Part III (5 min)	Exit Interview (Table 4.8)	5 min

plans in the form of pseudocode:

```
// Task 1: TODO:
get obj1 from obj2, then call m() on obj1
```

After that, the participants started the actual implementation based on their high-level understanding. The participants were not forced to work in two phases, so they approached the tasks the way that worked best for them. Some of them preferred to work in two phases while the others preferred to implement the changes while they planned for them. If they got stuck, the participants preferred to comment the change and move on to the next task; which was closer to their natural way of working. The implementation phase involved mainly testing, debugging, and refactoring.

During the planning phase, the experimenter asked the participants general questions to measure the comprehension at an architectural level, i.e., system structure (Table 4.7). To measure the comprehension at the level of implementation details and behavior, she asked them intermediate questions while doing the implementations. The questionnaires between the tasks helped ensure that the participants understood the code, and that the things that they were learning helped them do the modifications better. The participants were allowed to test their modifications on the GUI by running the program whenever they wanted. If the participants struggled while

doing the tasks, the experimenter kept them on track by referring them to diagrams and asking them whether they found anything useful.

Table 4.6: Hands-on exercises used to tutor the OOG.

	Navigating the Graph View
	Notice that the root object `breakthrough:Breakthrough` is not shown, but keep in mind that it exists in a global `shared` domain and it points to all these objects.
	Notice that the top level architecture of the system is MVC
	Notice that some objects have several decorating labels, and it is hard to tell how these types are related to the object type. Notice that this information is about the type hierarchy, so you may need the class diagram or Eclipse to explain this. Based on this please answer the following questions:
Q1.1	How do the high-level objects of the system interact?
Q1.2	Try to navigate from any object on the graph using the "trace to code" feature. To which class does the traceability take you? What does this tell you?
Q1.3	Pick one object on the diagram that has a decorating label and explain how this information is useful to you.
Q1.4	Pick one object on the diagram and explain how you can get to this object. (Hint: double click the object name and observe what gets highlighted).
	Exploring the Object Hierarchy
Q2.1	Expand the substructure of any object. Can you explain object relations within the substructure? How is this useful?
Q2.2	Notice that some of the nested objects are inside thick borders or private domains called `owned`. Why do you think this is the case? (Hint. Use the trace to code feature.)
Q2.3	As you can see, some objects point to other objects by a dotted edge. Can you expand these objects and explain why that is the case?
	Exploring the Tree view (Ownership tree)
Q3.1	Pick one object that could be related to the first task and use the tree view to search for all the objects that are communicating with it. For example to see all possible edges between `factory` and `window` in the tree you can type the expression (*factory*–>*window*).
	Exploring Objects
Q4.1	How many objects does the position object represent? (Hint: use the trace to code feature)
Q4.2	Pick a certain scenario related to breakthrough and try to explore one of the objects related to this scenario using the trace to code feature. What does this tell you?
	Exploring Edges
Q5.1	Explore the edge between any two objects using the trace to code feature. What does this tell you?

Table 4.7: Recurring questionnaire asked to participants between the tasks.

No.	Question
QX.1	Can you formulate a hypothesis?
QX.2	Do you think the diagrams are useful?
QX.3	Do you think the package structure is useful?
QX.4	Can you map GUI components to code elements?
QX.5	What classes will you modify to perform this task?
QX.6	What objects will be communicating in this case?

The C participants received the OOG tutorial during the last 20 minutes. During

Table 4.8: Exit interview questions

No.	Question
I1	Do you think the tasks were hard or easy?
I2	Do you think the experimenter was directing you towards using the OOG?
I3	Do you think the experimenter was a pair programmer or a mentor?
I4	Did the experimenter refer you to use the OOG more than the CD or she referred you to both equally?
I5	Do you think the OOG is useful?
I6	Is it more useful than a class diagram or complementary?
I7	Where there any usability issues with the OOG Viewer?
I8	Do you think your change respected the design?
I9	Is there anything else you would like to add?

that time, the experimenter explained the OOG briefly and asked them if they could have done better using the additional diagram (OOG). Since they were struggling with certain parts of the problem, she asked them if the OOG made visually obvious the object relations for which they were searching. She also asked them whether having access to OOGs could have helped do a modification that respected the design.

Near the end of the study, the experimenter asked participants in both groups exit interview questions (Table 4.8). The purpose of the interview was to capture general subjective feedback about whether diagrams in general and OOGs in particular can help developers understand the code to do the modification that respects the design. The interviews also included questions about the experiment and other open comments the participants wanted to add.

Chapter 5: Analysis of Results

In this chapter, we explain how we analyzed the collected data in the transcripts (Section 5.1). Then we compare the overall performance of participants in both groups both quantitatively (Section 5.2) and qualitatively (Section 5.3).

5.1 Data Transcription

We transcribed the recordings offline[1]. The transcripts were our primary source of data as well as the participants answers to the questionnaires and interviews. Based on the transcripts, we tracked the code elements that the participants explored, the time they spent, and their think-aloud (Figure 5.1). In the column *Question Code*, we classify the questions they were struggling with. In *Tool/Diagram*, we capture whether they relied more on Eclipse, on the OOG viewer, or on diagrams and what types of diagrams were most helpful. In *Feature*, we capture which features were used both in eclipse and the OOG viewer to answer which questions. The *Navigation event* could be: a navigation with no target (e.g., scrolling, stepping in debugger) where the developer was rapidly skimming through text without a clear target, navigation commands that changed which element was currently visible (e.g., *ReferenceTo*), and navigation commands that generated some list of elements that could be navigated to (e.g., *References*). So in the first case, the targets are things that are now visible whereas in the second case the targets are something on which a command was executed. The *Navigation target* could be a class name, a method name qualified by the class name, an OOG, a class diagram, or a feature in the OOG Viewer.

[1]We reused a scheme that was developed by Thomas LaToza.

Time in video	Think aloud	Question Code	Tool used	Feature	Navigation Event	Navigation Target
			Eclipse	type hierarchy	References	Command
						MoveCommand
						NullCommand
0:20:00	move command there we go thats what I needed				Reference To	BreakthroughPiece
0:21:00	**Task1: TODO**				scrolling	
	ok the game		CD7			
0:22:00	Im just trying to think ... but it does not appear					
	Im trying to figure out who is storing the location here	**Get-X-From-Y**				
	because we have to be able to access ...					
	I mean somebody is keeping track of all these pieces					

Figure 5.1: Excerpts of the developer's thought process recorded in the transcripts.

5.2 Quantitative Analysis

Before we provide analysis of our results, we give a brief overview of how our participants used the time available to them. Our experiment was in the form of a 3-hour session (Table 4.5). Unsurprisingly, the first task was the most challenging to all participants since it was their first attempt to explore the code. As a result, we broke down the first hour into different parts: formulating hypotheses, answering the general questionnaire, and providing the plan. We used the participants answers to the general questionnaire (Table 4.7) to measure their level of comprehension. The participants did not answer all of these questions immediately, but made initial assumptions which were either correct or incorrect. Therefore, we measure the time they spent to validate their initial assumptions.

We measured the participants response times to the questionnaires and the time spent on each task separately based on the time recorded in the transcripts (Figure 5.1). For the questionnaires, we did not impose any limits on the response time. As a result, some participants answered these questions immediately while others still required more time. Also, we could not ask the participants some of the questions until they reached a stage in the experiment when these questions could be asked. Since some of the participants spent most of the available time on the first task,

they did not answer some of the questions. Some participants found the answer to a question while searching for answers to an other question. In our analysis, we count the time they spent from the moment the experimenter raised that question.

The participants performed the tasks in two phases, i.e., planning and implementation, so we measured the time they spent in both. On average, the participants spent most of the available time on Task 1. In the cases of participants who did not complete all the requirements for Task 1, the experimenter had to ask them to move to the next tasks. Therefore, our results do not include all the data for all the activities in all tasks.

5.2.1 Summary of Results

We found that the availability of OOGs does improve developers' performance. Two E participants succeeded in their tasks compared to only one C participant (Table 5.2). On average, the E participants were able to finish their tasks by browsing less irrelevant code with relative percentage difference of (10%–60%), and spent less time (by 22%–60%) (Table 5.1). We use bar charts to show the average difference of each of our dependent variables between the two groups. The bar charts also show the *standard error* within each group, a measure of the reliability of the calculated mean, The standard error was calculated by dividing the standard deviation in each group by the square root of the sample size. The error whiskers on the bar charts indicate that there was high standard deviation in the C group data for these two variables (Figure 5.2) which means that the individual observations were not close to the mean. This could be due to the small sample size.

To illustrates the distribution of our variables, we also summarize our results using box-and-whisker plots. These plots helped us detect things such as outliers, median, minimum and maximum values. A box plot can be read through a five-number summary: the whiskers represent the *minimum* and *maximum* observations,

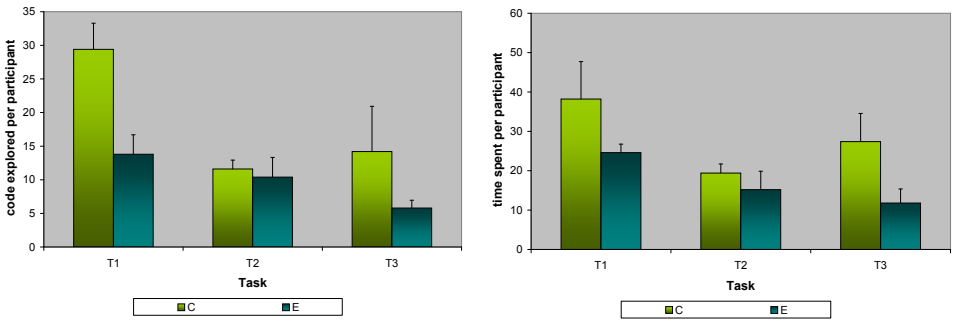
Table 5.1: Summary of quantitative measures

Variable	Task	p-value	Percent. Diff.
Code explored	T1	0.00794	53%
Code explored	T2	0.264	10%
Code explored	T3	0.0687	60%
Time	T1	0.147	36%
Time	T2	0.232	22%
Time	T3	0.0476	60%

Table 5.2: Summary of participants' performance and relation to programming experience. The column *Ind. Yrs* refers to their years of professional programming experience. Time is measured in minutes. Code explored is measured in units of code (class or method).

P	Ind.	Task1			Task2			Task3		
	Yrs	Success	Time	Code	Success	Time	Code	Success	Time	Code
C1	0	No	41	30	No	28	15	No	45	10
C2	6	No	29	31	No	18	13	No	43	41
C3	4	No	22	26	No	14	8	Yes	15	7
C4	0	No	25	18	No	19	13	No	10	7
C5	0	No	74	42	No	18	9	No	24	6
E1	2	No	27	24	No	30	14	No	2	17
E2	0.5	Yes	25	13	Yes	22	20	Yes	9	7
E3	20	Yes	24	15	Yes	10	6	Yes	6	6
E4	20	No	17	10	No	9	8	No	4	5
E5	2	No	30	7	No	5	4	No	23	9

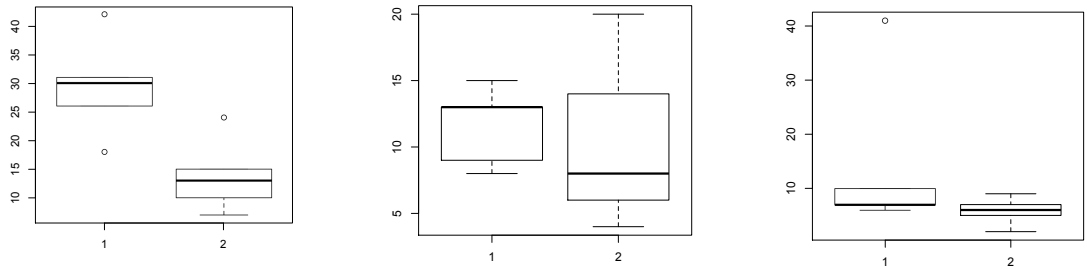
while the box reflects the lower quartile (Q1=25%), the *median* (Q2) represented by the thick line, and the upper quartile (Q3=75%). A quartile is any of the three values which divide the sorted data set into four equal parts, so that each part represents one fourth of the sampled population. *Outliers* are points which are more than 1.5 the interquartile range (Q3-Q1) away from the interquartile boundaries, and are



(a) Average code explored in the three tasks.

(b) Average time spent in the three tasks.

Figure 5.2: The mean difference for two variables in the three tasks.

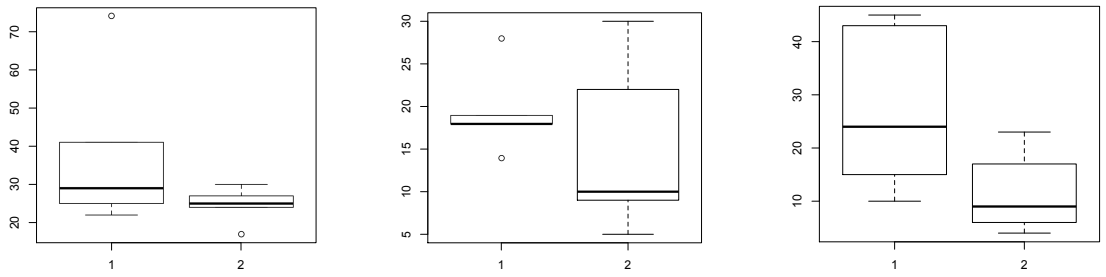


(a) Task1(p=0.008) (b) Task2(p=0.26) (c) Task3(p=0.07)

Figure 5.3: Boxplots for the number of code elements explored in the three tasks. The 1 on the horizontal axis refers to the C group and the 2 refers to the E group. The median code explored in the three tasks was less in the case of the E group. Only the code explored in Task1 was significantly less for the E group (p=0.007). The difference was not significant in the other two tasks due to ties between the observations in both groups.

marked individually as dots. The boxplots provided us with more information on the distribution of the data. The dots on each plot indicate that there were outliers in both groups for both variables (Figures 5.3(a), 5.3(c), 5.4(a), and5.4(b)). The skewness was positive most of the time, but the kurtosis was different. In the cases were the kurtosis was high, it was an indicator that the observations are more outlier-prone.

The box plots helped us describe the statistical distribution of the data, i.e., the spread of each variable (measure) within each group independently. But it did not tell much about the differences between the two groups. Therefore, we also used tests for statistical significance. We performed non-Parametric tests since they do not require any prior assumptions about the population such as equality of variances or normality of the samples. The non-parametric tests were valid in our case due to small sample size. We used the *Mann Whitney U test* since it is the non-parametric equivalent to the Student's t-Test. This test considers the ranking of data even if the shapes of the distributions are different. We performed lower one-sided tests. We display the results of applying the u-test below each plot as p-values. The p-values indicate that the mean difference in the code explored was significant in only Task 1



(a) Task1(p=0.15) (b) Task2 (p=0.23) (c) Task3 (p=0.05)

Figure 5.4: Boxplots for the time to task completion of the three tasks. The 1 on the horizontal axis refers to the C group and the 2 refers to the E group. Both groups are positively skewed, but the kurtosis in the C group is also obvious. The E participants completed Task3 in significantly less time (p=0.04). The test did not provide significant difference in the case of the other two tasks since there were ties between the two groups.

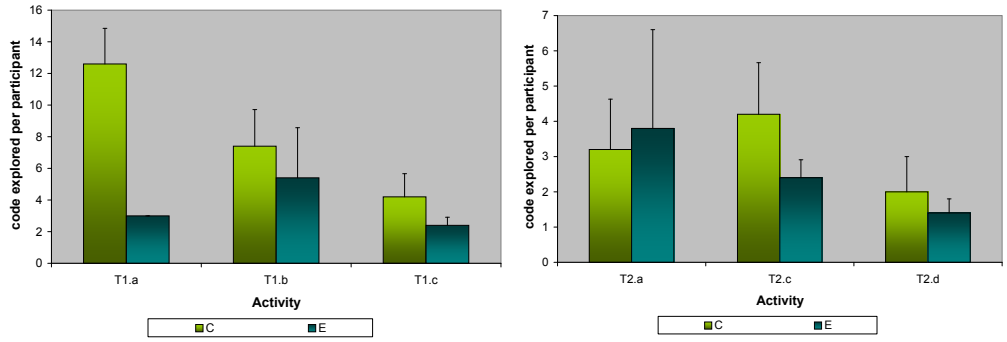
(Table 5.1). For the time spent, we found that the mean difference was significant in only Task 3. The tests also indicated that there were ties between the observations in some tasks. We provide analysis for each of our variables for the three tasks, and the activities within each task in the following sections.

5.2.2 Code Elements Explored

For each task, we captured the average number of code elements (classes or methods) explored by the participants in each group while doing a task (Figure 5.2(a)). The figure reflects how the E participants ended up exploring fewer code elements. Figure 5.5(a), 5.5(b), and 5.5(c) show details at the level of the activity.

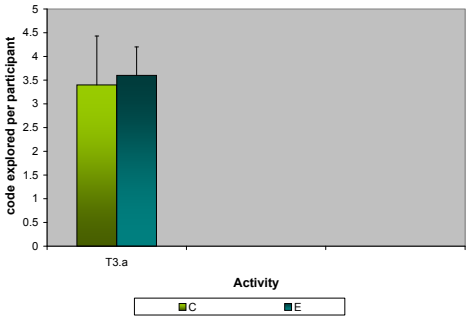
The median code explored in the three tasks was less in the case of the E group (Figure 5.3). Only the code explored in Task1 was significantly less for participants who had access to OOGs (p=0.007). The difference was not significant in the other two tasks due to ties between the observations in both groups.

We also plotted graphs for the different activities within each task (Figures 5.6, 5.7, and 5.8). In the activities that required answering questions related to object relations, the difference was even more significant in the case of Task1. However, a



(a) Average code explored in the activities of Task1.

(b) Average code explored in the activities of Task2.



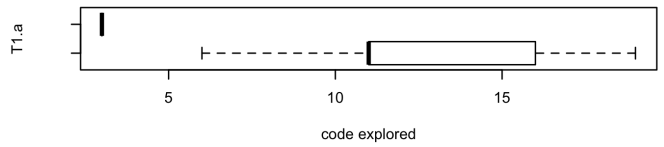
(c) Average code explored in the activities of Task3.

Figure 5.5: Average number of code explorations. Error whiskers on the bar charts indicate that there was high standard deviation in the C group.

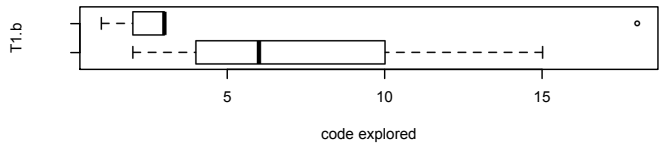
closer analysis of the activities in the other two tasks did not show any significant difference due to ties between the observations in both groups.

5.2.3 Time to Task Completion

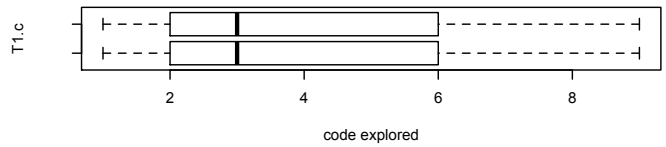
On average, the E participants completed Task 1 twice as fast as the C participants (Fig. 5.2(b)). More detailed bar charts appear in Figures 5.9(a), 5.9(b), and 5.9(c). We also show box plots to compare the three tasks (Figure 5.4) and the activities in each task (Figures 5.10, 5.11, and 5.12). Figure 5.4(a), shows that both groups are positively skewed, but the kurtosis in the C group is also obvious. The U-test suggest that the E participants completed Task3 in significantly less time (p=0.04). The test did not provide significant difference in the case of the other two tasks since there were ties between the two groups.



(a) Activity T1.a



(b) Activity T1.b



(c) Activity T1.c

Figure 5.6: Boxplots for the number of code elements explored in the activities of Task1. The upper plots represents the C group and the lower plot represent the E group. In the activities that required answering questions related to object relations, the difference was even more significant in the case of Task1.

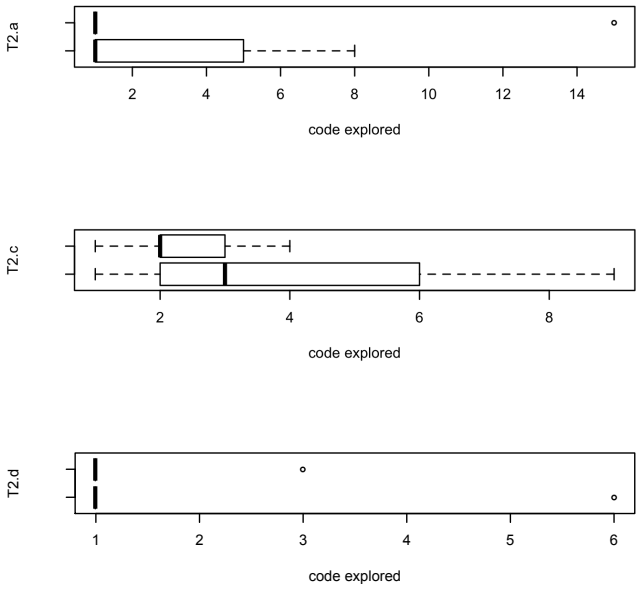


Figure 5.7: Boxplots for the number of code elements explored in activities T2.a, T2.c, and T2.d of Task2. The upper plots represents the C group and the lower plot represent the E group. The tests did not show any significant difference due to ties between the observations in both groups.

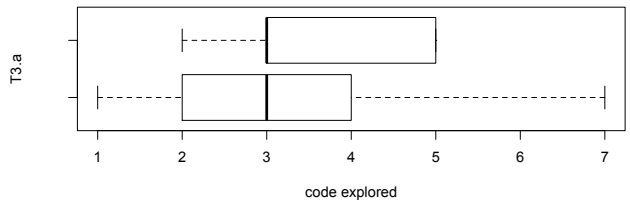
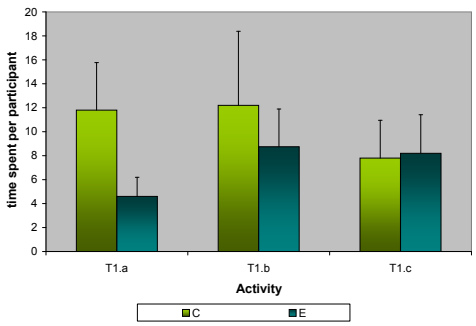
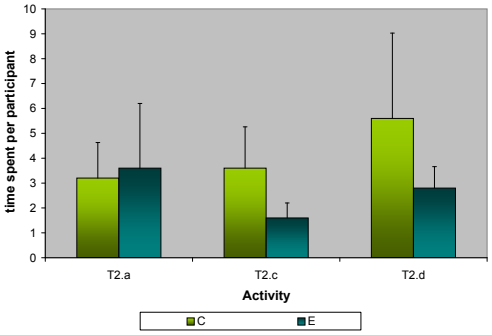


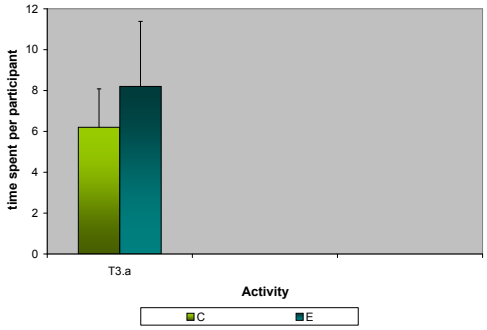
Figure 5.8: Boxplots for the number of code elements explored in activity T3.a of Task3. The upper plot represents the C group and the lower plot represent the E group. The tests did not show any significant difference due to ties between the observations in both groups.



(a) Average time spent in activities of Task1.



(b) Average time spent in activities of Task2.



(c) Average time spent in activities of Task3.

Figure 5.9: Average time to task completion. Error bars represent the standard error. On average, the E participants completed Task 1 twice as fast as the C participants

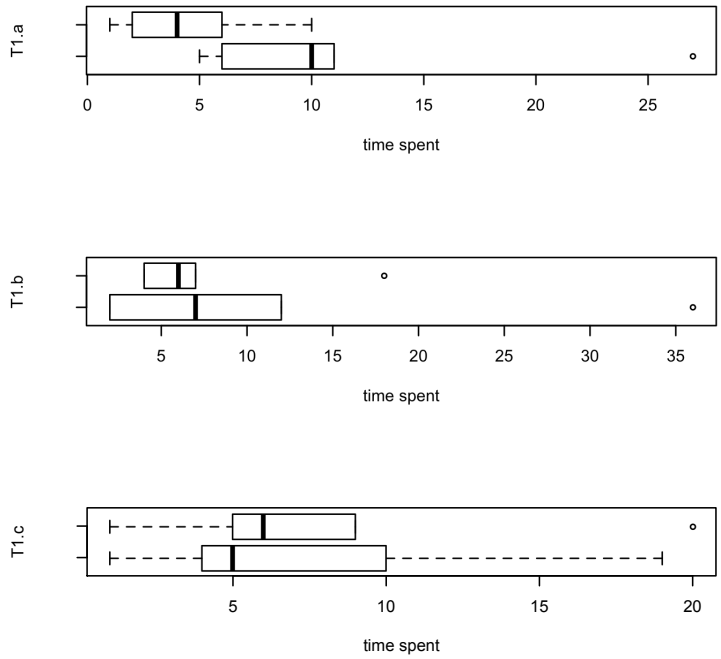


Figure 5.10: Boxplots for the time spent in the activities of Task1. The upper plot represents the C group and the lower plot represent the E group. The dots on each plot indicate that there were outliers due to small sample size. Kurtosis was obviously higher in the second activity in the case of the E group which makes the data more outlier-prone.

5.2.4 Success on a Task

Two of the E participants completed the three tasks compared to only one C participant who completed only one task (Figure 5.13(a)). We define the success on a task as the ability of a participant to provide a complete implementation of all the required functionality for the task, and test that their implementation was correct by running the application and demonstrating the change on the GUI. The participants who did not succeed, were able to provide plans for all the required activities in each task, but either did not have enough time to provide implementations or tested the implementation but found bugs that prohibited them from proceeding. On the other hand, we define completion of a certain activity as either the ability to provide a precise plan of which objects will be used to perform the actual implementation and whether these objects were accessible inside the class where the implementation

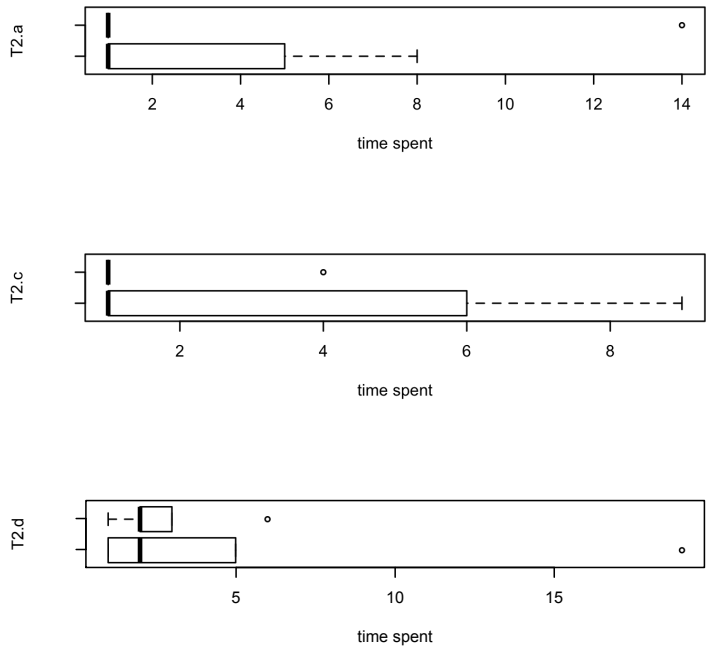


Figure 5.11: Boxplots for the time spent in activities T2.a, T2.c, and T2.d of Task2. The upper plot represents the C group and the lower plot represent the E group. The median was the same in all cases, but the data from the E group was always positively skewed. Also, there were more outliers in the case of the C group.

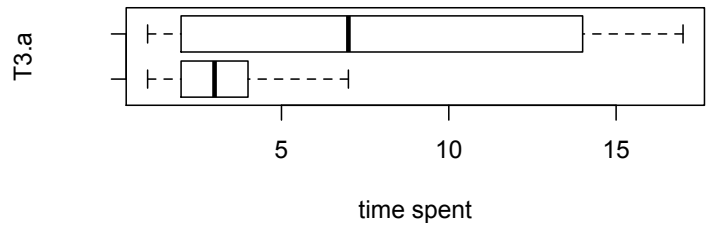
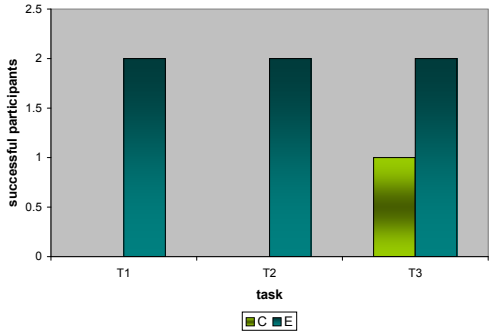
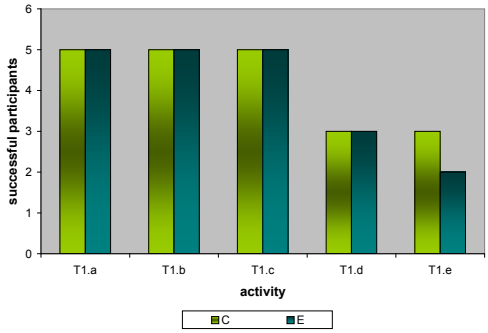


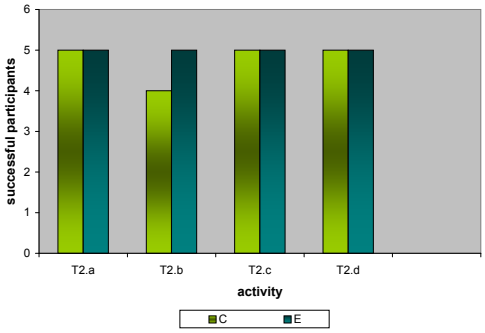
Figure 5.12: Boxplots for the time spent in activity T3.a of Task3. The upper plot represents the C group and the lower plot represent the E group. No outliers were indicated, but the kurtosis is much higher in the C group.



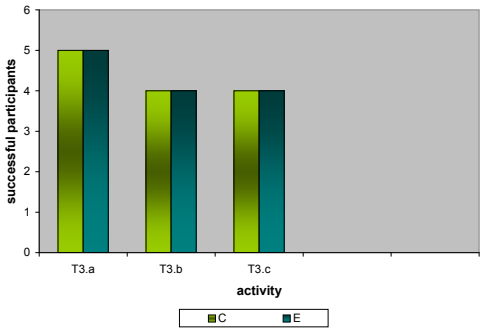
(a) Successful participants in the three tasks.



(b) Participants who completed required activities in Task 1.



(c) Participants who completed required activities in Task 2.



(d) Participants who completed required activities in Task 3.

Figure 5.13: Two of the E participants completed the three tasks compared to only one C participant. Most of our participants were able to complete most of the required activities in the three tasks.

should take place. Most of our participants were able to complete most of the required activities in the three tasks (Figures 5.13(b), 5.13(c), and 5.13(d)).

5.3 Qualitative Analysis

According to the transcripts, all participants divided the tasks into smaller activities to be able to solve them, which reflected on how they provided their plans and implementations (Tables G.1 and G.2). Therefore, we decided to analyze our data at the level of the task as well as the activities within each task. For each task, we studied the sequence of activities in which all the participants engaged (Table 5.3). All the participants mentioned that Task 2 is related to Task 1, so most of them implemented these tasks together. As a result, the participants did not follow these activities in the same sequence, but for simplicity and to be able to compare results, we list them in the order specified. In the rest of this chapter, we will refer to these activities by their numbers.

Table 5.3: Activities performed in each task.

Activity	Description	Question involved
T1.a	Locate in which class to implement the validation logic	Is-In-Tier
T1.b	Look for the data structure representing the game board	Has-A
T1.c	Get hold of that object inside the class responsible for validating movements	Is-Owned/Is-Part-Of
T1.d	Locate which object is responsible for showing the status message	Is-In-Tier
T1.e	Get hold of that object inside the class responsible for validating a movement	How-To-Get-X
T2.a	Locate in which class to implement the capture	Is-In-Tier
T2.b	Figure out which object represents a piece to compare it to an opponent piece	Has-A
T2.c	Get hold of that object inside the class responsible for handling captures	How-To-Get-X
T2.d	Figure out how to remove a piece from the game board	May-Alias
T3.a	Locate in which class to add the menu bar	Is-In-Tier
T3.b	Get hold of the objects that handle the movements and the captures inside that class	How-To-Get-X

For each activity, we studied the different navigation paths that the participants followed. A *navigation path* is a sequence of navigation targets that lead a participant to an outcome. For each path, we studied the length of a path (number of navigation targets including repetition), the time spent in following it, and the *outcome*, which

includes finding the answer to the question, locating the right place in the code to make the change, or validating an assumption. A *successful* path is one which leads to a successful outcome. An *unsuccessful* path is one that a participant follows for while, then abandons or finds distracting.

5.3.1 H1: Developers who have access to OOGs answer questions about the object structure that cannot be answered using class diagrams

We had two observations that provide direct evidence to each of the success criteria for this hypothesis.

H1.1. Developers perform their tasks in a sequence of activities related to questions about the object structure. The activities in which the participants engaged (Table 5.3) were often in the form of questions about the object structure (Table 5.4), so we classified the questions involved in each activity using our coding model (Table 1.1). The participants performed some of these activities during the planning phase, and these often involved questions in which architectural tier an object existed. For example, in Task 1 the participants asked about the class inside which the validation logic should be implemented (activity T1.a), so we coded this question as (Is-In-Tier). Some other activities were performed during the implementation phase, and that often involved asking questions about how to get an object inside the class where the modification should be implemented (activity T1.e), so we used the code (How-To-Get-X).

H1.2. Developers use relations between objects on an OOG to answer their questions about the object structure. We identified the challenging questions with which the participants struggled and how they answered them using either an

Table 5.4: Questions about the object structure common to all participants.

P	Think-aloud	Code
C1	"I think the list here has to be something accessible from I guess we need to find some kind of coordination between different functions they are talking to the same list right? So i wanna make sure not create a new list here"	May-Alias
	"so this is basically where we are going to get the position for the list so basically that function [performAction()] needs to basically access this [generatePiece-Multimap()] I'm looking at this function here [MoveMommand.execute()] then from that function I think they are getting the list of all figures"	How-To-Get-X
C2	"I mean somebody is keeping track of all these pieces so i want to figure out who is doing that"	How-To-Get-X
	"and then figure out how in this execute function access that . . . "	How-To-Get-X
	"I'm more concerned about how to get to [BoardDrawing]"	How-To-Get-X
C3	"now I want to find who is creating this object"	How-To-Get-X
C5	"I mean any of these are really a possibility of where it might have all the positions of all the pieces. I guess I should be looking for some sort of a data structure"	Has-A

OOG, a class diagram, or a feature in Eclipse. All the C participants seemed to have difficulty figuring out how to get to certain objects. Since they did not have access to OOGs, they used alternative methods such as: exploring call hierarchies, or looking up certain keywords (Figure 5.14(a)), or referring to the class diagrams. Those who referred to the class diagrams were often looking for a starting point, which was always a certain type, so they wondered wether there existed a feature in Eclipse that can give them a list of all the classes that instantiate that type. They often used the feature of searching for references of a certain type in Eclipse (Figure 5.14(b)). One participant from the control group used print statements to check whether a method was triggered when a piece is dragged on the board.

We compare between participants from each group while trying to answer these questions (Tables 5.5 and 5.6). The tables show which diagram or tool the participants used to answer their questions about the object structure and how long it took them to answer the question, with or without the OOG. The OOG helped the E participants answer their questions about the object structure most of the time. Since the C participants did not have access to the OOG, they were either unable to find answers

or struggled using other means such as class diagrams, Eclipse, or reading through the code. For example, C4 was unable to find an answer to his question since the class diagram he looked at did not show lists of figures or listeners or specific image instances. He implemented a hack to work around the problem. This also applies to E1 who was provided with an OOG but refused to use it in this instance since he claimed that he could answer his questions just by looking at the code.

Table 5.5: Some of the methods followed by E participants to answer questions about the object structure while doing Task 1.

P	Think-aloud	Code	Diagram	Tool	Outcome
E4	"what I'm looking for what represents the board so there is got to be a starting point, so the question is there has got to be a position so if I'd go to somewhere so what references the position ah okay alright so that [OOG] actually helps looks like there are just two objects"	Points-To	OOG	Ov.F3	Succeed
	"so I'm gonna go inside this one [`boardDrawing`] even though I did not think it had anything to do with it because its name doesn't make any sense to me so abstractFigure::owned I was there and it wasn't there aha [figureMap] position"	Has-A/Is-Part-Of/Is-Owned	OOG	Ov.F4	Succeed
E5	"does board figure mean that it's the piece? or I can go here [`boardDrawing`:BoardDrawig] and check if it has objects called `boardFigure`"	Is-Part-Of/Is-Owned	OOG	Ov.F4	
	"`boardDrawing` has MAPS okay these are the logical groupings I want to make sure if it actually contains these guys [`boardFigure` objects]"	Is-Part-Of	OOG	Ov.F4	
	"okay so it has a dotted edge to `boardFigure` a dotted edge means that it has something that is not exposed"	Points-To			
	"okay so now its being expanded"		OOG	Ov.F4	
	"okay so its says that these guys are part of `boardDrawing`"	Is-Part-Of			
	"okay so based on this [`fFigures:ArrayList<Figure>`] its an array list of figures then you can say okay this is probably the big board"				Succeed

Table 5.6: Some of the methods followed by C participants to answer questions about the object structure while doing Task 1.

P	Think-aloud	Code	Diagram	Tool	Outcome
C1	"I'm just gonna poke around different classes seeing which one might be I know thats probably no the most efficient method so yeah right now just looking for the class has a function that keeps track of all these positions of all the pieces"			Ec.F1, Ec.F8	Fail
	"it does not really help you a lot in a sense I mean its nice to see how it is connected, but I need to see a specific I mean I guess the position of the piece is going to play a role, but it doesn't really seem"		CD6,8		Fail
	"so this is the one I'll probably be focusing on so there are limited amount of things that are interfacing with position here"		CD1		Fail
	"so this is basically where we are going to get the position for the list so basically that function [performaction()] needs to basically access this [generatepiecemultimap()] I'm looking at this function here [movecommand.execute()] then from that function I think they are getting the list of all figures"	How-To-Get-X		Ec.F1, Ec.F7	Succeed
C3	"I'll go to the position class first. I think this class... an object will be created when you make a move."		CD6		
	" Now I want to find who is creating this object?"	Points-To		Ec.F3, Ec.F9	
	"since this function [adjustFigurePosition()] instructs to move [moveby()] let me execute it to see... can i use this one to debug ... I want to make sure that this function getting called when you make a move"				Succeed

5.3.2 H2: Developers who have access to OOGs explore fewer code elements to complete their tasks

We had two observations that provide direct evidence to each of the success criteria for this hypothesis.

H2.1. Developers use OOGs to understand object relations, and navigate fewer paths through the code. We compared between two participants from each group while doing the same activity (Tables 5.7 and 5.8). The participants followed different navigation paths, with one path focusing on objects while the other path focusing on navigating classes. Using the OOG helped E4 focus more, and navigate

only relevant elements in the code to answer his question, while C1 was distracted twice.

In general, participants in both groups spent a fair amount of time understanding the code and performing the actual implementation, but the E participants explored relatively less code, especially on the first task. All the code they explored was relevant and led them to successful paths. The E participants explored almost always one path which was successful. The extra time the E participants spent was to investigate further relations of class type hierarchies. This means that all their time was spent effectively unlike the C participants who spent much time just navigating randomly. For example, E5 and E3 spent relatively long time compared to their colleagues only to investigate further that `BoardDrawing` is a `BoardGameObserver` and can be safely casted to that type[2].

Table 5.7: Two navigation paths followed by C1 for activity T1.b.

Path	Time	Navigation Target	Outcome

Path1	6	`BoardFigure`
		`ImageFigure`
		`AbstractFigure`
		`PositioningStrategy`
		`PositioningStrategy.CalculateFigureCoordinatesForLocation()`
		`PositioningStrategy`
		`ChessBoardPositioningStrategy`
		`Position`
		`PositioningStrategy`
		`FigureFactory`
		`Position`
			Fail

Path2	4	CD7 CD1 CD6 CD1	Fail

Path3	2	`FigureFactory`
		`BreakThroughPieceFactory`
		`BreakThroughPieceFactory.generatePieceMultiMap()`
			Succeed

H2.2. Developers use traceability links on the OOG to navigate to only relevant code. All participants in the study were able to narrow the scope of navi-

[2]All the examples in this section are illustrated using Figure 3.13 in Chapter 3.

Table 5.8: Two navigation paths followed by E4 for activity T1.b.

Path	Time	Navigation Target	Outcome

Path1	6	`CompositeFigure.add()`
		`Figure`
		`CompositeFigure`
			Fail

Path2	1	`moveCommand:MoveCommand`
		`boardDrawing:BoardDrawing`
		`AbstractFigure::owned`
		`BoardDrawing::MAPS:figureMap:HashMap<Position,List<BoardFigure>>`
			Succeed

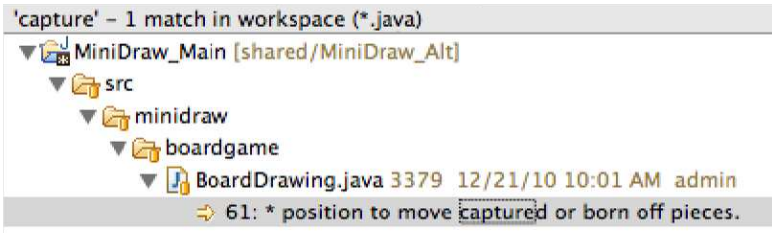
gations down to 7 classes: `BreakThrough`, `BreakThroughPieceFactoryMoveCommand`, `BoardFigure`, `GameStub`, `BoardDrawing`, and `MiniDrawApp`. However, the C participants spent along time exploring much irrelevant code before they were able to narrow the scope compared to the E participants. For example, after 1 hour and 22 minutes, C1 realized that he should not have explored a large number of classes, and should have focused on only 3 classes:

> "All the changes are gonna come in these 3 files [`BreakThrough`, `BreakThroughPieceFactory`, and `boardDrawing`]"

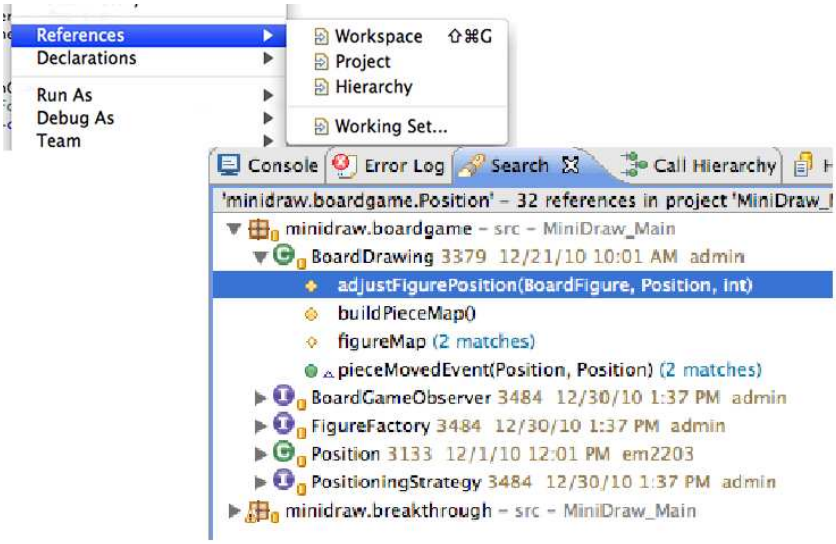
Most of the time, the C participants, were navigating code randomly especially during the first 60 minutes. When we asked the C participants to formulate their hypotheses for Task1 (QX.1, Table 4.7), they provided the answers in Table 5.11. The answers indicate that they relied on the following in order: Package Explorer (Eclipse), file search, documentation (JavaDoc), or reading through code. Most of the time, the file search disappointed them (Figure 5.14(a)). For example, C1 used the file search randomly to locate where to implement the capture:

> "I did not assume any kind of a method to be implemented, but I just assumed that maybe searching for capture might narrow my search down a little bit if it yield something then its gonna guide me to the right direction if not then I probably prefer the documentation"

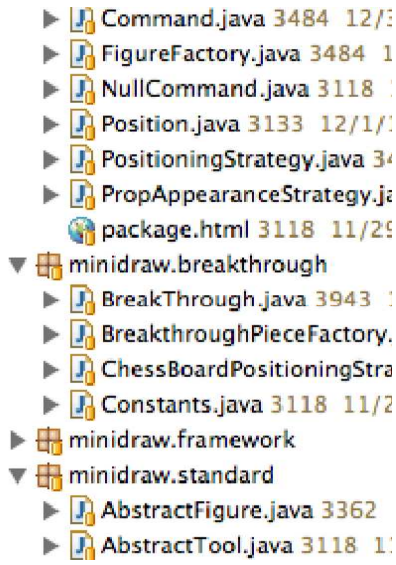
All the E participants, on the other hand, worked within the scope of only the above mentioned classes from the beginning (Table 5.12). They also followed a certain



(a) Eclipse file search.



(b) Eclipse feature to find references or declarations in project.



(c) Eclipse package explorer

Figure 5.14: Developers who do not have access to OOGs use alternative techniques in Eclipse to answer questions about the object structure.

navigation path. They first navigated from the `command:MoveCommand` object on the OOG to `command` instance inside `BreakThroughPieceFactory`. Then they navigated to `MoveCommand` class where they were curious to understand the relation between this class and `BoardFigure`. Then they navigated back to the factory class where an instance of `GameStub` was created. Then to see who calls `GameStub` they navigated to `Breakthrough`. Then they referred back to the OOG to see how on a high level these main objects are interacting. From the OOG, they navigated back to either `GameStub` or `MoveCommand` to start the implementation. Then they referred back to the OOG to see if there was a way to get a certain data structure that holds all pieces in certain positions on the board. They often expanded the `BoardDrawing` or `BoardFigure` objects to answer this question where they found a Part-Of relation between the two objects represented in `figureMap:HashMap<Position, List<BoardFigure>>`.

5.3.3 H3: Developers who have access to OOGs take less time to complete their tasks

We had two observations that provide direct evidence to each of the success criteria for this hypothesis.

H3.1. Developers who have access to OOGs take less time to gain a high level understanding of the code. We found that there is a difference in the comprehension level among the participants in both groups. Overall, the E participants took less time to understand the code, presumably by looking at object relations on the OOG and the OOG Viewer (Tables 5.9 and 5.10).

On average, the E participants spent less time than those in the C group to perform the activities in Task 1 (Table H.1). Activity T1.b, for instance, required the ability to map GUI components to their corresponding code elements,i.e., classes. Some of the C participants such as C2 and C3 knew immediately that `BoardDrawing` represented the

chess board. C2 had over 20 years of experience and was familiar with design patterns and frameworks. Things get more interesting in the case of C3, who made a correct guess about `BoardDrawing` but spent a long time in validating his assumption of where to perform the validation logic, i.e., activity T1.a. He used print statements as a debugging technique to make sure that `BoardDrawing.adjustFigurePosition()` is the method being called in the case of a movement.

We also present results for individual participants (Tables 5.7 and 5.8). Most of the E participants did not immediately understand that `BoardDrawing` is the chess board. Still, they spent their time effectively, were methodical and followed Points-To and Part-Of relations on the OOG to validate or falsify their assumptions. For instance, E4 used a Points-To relation between `p:Position` and `command:MoveCommand` (Table 5.8). Also, E3 and E5 used Part-Of relations to check whether there was a container-containee relation between `boardDrawing` and `boardFigure` (Table 5.9).

Table 5.9: Responses of the E participants to QX.4 (Table 4.7).

P	Answer
E3	"so [boardDrawing] maybe thats where the container is so let's take a look at there so property map thats container an `ArrayList` of figures thats a container this looks like what I'm looking for `FigureChangeEvent` array alright so `boarddrawing` contains a list of figures I think `boarddrawing` is really the object I was looking for okay so `boardDrawing` has a list of figures and it observes the individual pieces"
E4	"so let's see so I would think this is probably must be `boardFigure` is either a piece or `boardDrawing` observer listener [reading labeling types] since this is the MODEL this is [`boardDrawing`] got to be I'll assume the let's see your VIEW is the graphical representation your CONTROLLER is your actual code that does something so in your MODEL you've got to have the ... I would think that this class right here [`boardDrawing`] would have in it the representation of the current state of the board which is which pieces are in which places the view is the graphical representation"
E5	"okay I'll try so based from this [`Breakthrough::VIEW`] it says view so based from that I'm gonna say okay some of the UI stuff belong here okay so the board would be .. okay so this is the MODEL its just the data so the data for the board would be this board drawing ... alright let's go for the other ones a `boardFigure` ... let me go deep into this ... does `boardFigure` mean that its the piece? ... I can go here [`boardDrawing:BoardDrawig`] and check if it has objects called `boardFigure` ... okay this is probably the big board; that `boardDrawing`"

H3.2. Developers who have access to OOGs follow shorter paths to complete their tasks. Hypothesis 3 follows from Hypothesis 2 since a developer who

Table 5.10: Responses of the C participants to question QX.4 (Table 4.7).

Q	P	Answer
QX.4	C2	"`boardFigure` is the piece I think"
		"I think this is the graphical representation [`boardDrawing`] the actual drawing part where this [game] is more like the logic of it or maybe its the other way around, but it's checking wether it's valid so it seems that it's the logic here [game]"
	C3	"I think this is the display [`boardDrawing`] I mean the chessboard"

Table 5.11: Responses of the C participants to QX.1(Table 4.7).

P	Answer
C1	"so now just looking at classes that seem to be a reasonable candidate to have that type of code there."
	"It organizes it [package structure]...I mean with a limited time frame its difficult to really conceptualize where this specifically would be"
C2	"the theory is we have to figure out somewhere in here wherever the movement is occurring prior to whatever we have to do here [boardfigrue.performAction()] we have to check if basically there is somebody in toX toY"
C3	"I want to understand the total architecture, I'll just go through the classes and the interfaces OK since we need to validate the movement then we have two classes related to that Position and ChessboardPositioningStrategy I think that would be the right place to start"
C4	"can I look at your images [reverse engineered class diagrams] to see how the classes are interacting with each other"

explores fewer code elements or code elements that are highly relevant to a task spends less time completing the task. Only participant E1 ended up exploring many code elements and spending a relatively long time compared to the other participants in his group (Table H.1). This participant received the tutorial on the OOG and was encouraged to use it. Instead, he refused to use the diagrams and insisted on browsing the code in Eclipse, which reflected negatively on his performance.

E4 spent a long time on activity T1.b since he followed two paths (Table 5.8). In one path, he explored the code randomly and finally gave up:

> "so at this point I'd probably go to your viewer and I'll probably look again and try to figure out."

In the other path, he referred to the OOG Viewer which helped him be more systematic and follow a certain line of reasoning to get to the answer in less time:

> "So it's not the position, I know it's not `moveCommand`, so I'd probably look

Table 5.12: Responses of the E participants to QX.1 (Table 4.7).

P	Answer
E1	"Actually I want to know the whole project thing [package structure]"
E2	"so we will have the board with pawns over there and then we will use move command then from `[moveCommand:MoveCommand]` we will access the [`p:Position`] of the pawn and then after we update the position of the pawn here [movecommand:MoveCommand] I think we will go back here [game:GameStub] and then here to update the [drawing:BoardDrawing] from the change here [(FigureChangeListener) label on drawing] I think we will have the figure change listeners here and these guys [figurechangelisteners] will render the changes maybe by using the [window:MiniDrawApplication]"
E3	"I would imagine there is got to be some representation of the board as a whole boardFigure sounds more like drawing. Game might have central knowledge so let's look inside the game to see what is in there"
E4	"I'm thinking just based on MODEL because what this thing represents is a game so if it is a game then your controller would be things that would be actually doing transactions. The view is the events like dragging a dropping"

inside `boardFigure` because I'm still thinking this is the actual thing itself but may be not. So what do I do in order to look inside this? Ok, so I would show the internals OK so . . . a list of figures aha!"

This scenario can be compared to C1 (Table 5.7). While E4 took 7 min overall to get to the answer using the OOG, C1 took 12 min overall to answer the same question and was distracted twice. The time difference was because E4 used the OOG to obtain answers about his questions regarding key object relations. As a result he followed a much shorter navigation path with a successful outcome. He explored only relevant code, i.e., classes related to the task. E4 used most of his time interpreting the OOG to learn about object relations instead of browsing the code. Much of the time that C1 wasted was in exploring irrelevant code and resulted in him discarding two navigation paths. Since this participant did not have access to the OOG, it was difficult for him to determine how to get to an object. These are questions that have the code How-To-Get-X. Instead, he started looking for associations in the class diagrams.

5.3.4 H4: Developers who have access to OOGs are more successful on their tasks

We had two observations that provide direct evidence to each of the success criteria for this hypothesis.

H4.1. Developers who have access to OOGs are more focused and follow fewer unsuccessful paths through the code. This hypothesis is a logical consequence of the above two hypotheses. As discussed in H1.2, the C participants used alternative methods such as Eclipse features and class diagrams in order to answer their questions. Our results indicate that these methods resulted in them exploring irrelevant code or taking a long time to complete their tasks. When they finally did the tasks, their implementation was either a hack or introduced a bug, and they did not have enough time to test their implementation or refactor their code.

To illustrate how this hypothesis depends on H2.1, we compare between two participants from each group while doing the same activity (Tables 5.7 and 5.8). The participants followed different navigation paths, with one focusing on objects while the other navigating classes where needing to investigate further within the class to answer a question. Using the OOG helped E4 focus more, and navigate only relevant elements in the code to successfully answer his question, while C1 was distracted twice.

H4.2. Developers who have access to OOGs make correct assumptions and have enough time to complete their tasks. Our results identified that several of the C participants made wrong assumptions about the code. For example, when C1 provided his plans to implement Task1, he assumed that there were two different maps, one for holding the positions and the other for holding the pieces on the board. After 1.5 hours, when he started doing the actual implementation, he discovered that

his assumption was wrong and that he had to find another way to get the initial positions of a piece when it first moves.

> "Maybe the assumption that we can get the coordinates from here was false! Well I thought I can get the list of coordinates from this function here [generatePieceMultiMap()] or maybe I thought I had the coordinates perviously, because this is the function that I thought I can get the coordinates from and based on this I was gonna do the rest and since this in fact does not provide the coordinates then ..."

Another example is C2. This participant built all of his plans for the three tasks on the assumption that he will be able to get the drawing object inside the `GameStub` class. After 1 hour and 22 minutes, he said:

> "the initial assumption that you can get game from board drawing is wrong. Essentially everything hinges on being able to access `boardDrawing`, so you've to be able to get it somehow. Especially in `BreakThroughPieceFactory`"

Even this falsification of assumption is not correct, since the participant relied on code hinting in the IDE which is not always the right way to get fields or methods defined in a certain class since these could be private members. The E participants were able to navigate from an edge between game and drawing which took them to a field of type `BoardGameObserver` inside `GameStub` which corresponds to a drawing object. They used the type hierarchy to understand this relation better when they found that a `boardDrawing` class implements the `BoardGameObserver` interface. C3 made a wrong assumption and built on it and thus got a wrong behavior in the piece movement, while testing the tasks on the GUI.

Based on participants answers to QX.1, most of the E participants were aware of how the objects across run-time tiers were communicating compared to only C2 . In fact, C3 used the class diagram to look for classes where a movement of a piece

could take place, and he assumed that `Position` and `PositionStrategy` could be related. Then he found the method `adjustFigurePosition` while he was browsing the code and assumed that this is the method that is being triggered, so he added a print statement there and did some testing through the GUI. The method was triggered, so he did the modification there, but his assumption was wrong. These observations mean that most of the C participants did not fully understand the big picture compared to the E participants who used the OOG to provide their plans about where to implement a certain feature (based on the existence of an object inside a run-time tier), and how to get objects from other objects (based on Points-To relations between these objects).

5.3.5 Modifications Performed by Participants

All the participants were able to go through all the tasks by providing their plans and implementations, but three of the E participants were able to test their modifications for the three tasks compared to only one C participant. In fact, C3 finished his plans and implementations but was unable to test the modification due to a run-time exception, so he spent the time debugging. This participant got a run-time exception since he created another `figureMap` object instead of getting it from `BoardDrawing`.

To implement the tasks, all the participants modified two packages: `minidraw.boardgame` and `minidraw.breakthrough`. To implement Task3, only a few of them modified the `minidraw.standard` package. We captured the modifications that the participants did during the planning and the implementation phases in all tasks using the (`//TODO: Task X`) comments in the code (Appendix G). Looking at the modifications implemented in Task 1 (Tables G.1 and G.2), all the C participants performed the validation inside the method `MoveCommand.execute()`. The E participants agreed that the validation should be inside the method `GameStub.move()`. None of the C participants changed the method `BoardDrawing.pieceMovedEvent()`,

while two of the E participants modified this method.

We believe changing the `minidraw.boardgame` and `minidraw.breakthrough` packages is logical since they are not core framework packages. However, changing the `minidraw.standard` package is not the most elegant change since it is a core framework package (Figure 5.14(c)). The participants changed this class to implement Task3 since they thought adding the `menubar` should be inside a class that extends a `JFrame`. Actually, the C participants searched for the "JFrame" keyword to be able to get to this class. The E participants, on the other hand noticed that `window:MiniDrawApplication` object is inside the `VIEW` domain, so they traced to this instance in the code. The traceability link took them to a `window` instance inside `BreakThrough` class where they found the `init()` method and thought they just need to cast that `window` instance into `JFrame` and attach the `menubar` to it. The participants who added the `menubar` to `BreakThrough` thought that was better since they had access to all the objects they needed to complete the implementation of the undo logic.

5.3.6 Questionnaires

Based on participants answers to the recurring questionnaire and the exit interview questions, we found the following: The participants found the tasks realistic and the average answer to question I1 (Table 4.8) was that the tasks were reasonable and took 3 hours.

Value of diagrams. Based on answers to question I2, we classified our participants into three groups: developers who do not use diagrams at all and rely on IDEs, developers who use class diagrams, and developers who use class diagrams in addition to object diagrams. We classified our participants based on their diagram usage (Table 5.14). G1 represents developers who used OOGs. The other group includes

developers who did not use OOGs which can be further split into two sub-groups: G2 represents developers who did not rely much on diagrams, and G3 represents developers who used class diagrams and found them useful. For example, E1 had access to OOGs, but he preferred to use the IDE. Most of the C participants compared to one E participant fall under G2 as evidenced by answers to question QX.2 (Table 5.13).

Table 5.13: Responses of participants to question QX.2 (Table 4.7).

Q	P	Answer
QX.2	C1	"typically like it is just digging through the code first you know diagrams are helpful supplement, but you know this one [CD6] is a little bit I mean it looks fancy, but its . . . you know as a high level view its helpful but some of the ones that are broken down are more helpful for specific purpose
	C2	"usually when you change a software you don't get these. You actually go digging through the code itself thats what I'm used to. I do that all the time"
	E3	"I don't have that much practice with the diagrams mostly I just read the code i find most of the diagrams hide two much information you really need the details of the code"

Our findings also show how developers who do not rely on diagrams are more likely to make wrong assumptions and build on wrong assumptions especially when they are required to modify a framework in a limited time. When it comes to framework programming, a developer is required to do the change that fits within the existing design, and most of the time this is not optional since the framework implementation relies on facts such as programming to an interface.

So developers need to think in terms of object relations and which objects can be exposed and thus keep in mind that they make changes that fit within the design instead of implementing quick fixes that could be hacks, e.g., changing modifiers on method return values or field declarations. They should also make sure that they use the existing design patterns instead of adding new code elements that could be duplicating existing functionality such as creating new instances or methods declarations.

OOGs are complementary to class diagrams. In several instances, we identified that the OOG was necessary to answer questions that the class diagram could not answer for developers. We also noticed that the converse was true. In fact,

Table 5.14: Categorization of participants based on diagram usage.

Group	P	What helped most
G1	E2	Eclipse, OOGs
	E3	Eclipse, OOGs, and experience
	E4	Eclipse, OOGs, and experience
	E5	Eclipse, OOGs
G2	C1	Eclipse, Class diagrams
	C2	Eclipse, Class diagrams, and experience
	E1	Eclipse, OOGs, and experience
G3	C3	Eclipse, Class diagrams
	C4	Eclipse, Class diagrams
	C5	Eclipse, Class diagrams

several participants did not realize that the `observer` instance declared inside the `GameStub` class was also a `BoardDrawing` until they saw, either on the class diagram or using the Eclipse type hierarchy feature, that `BoardDrawing` implements the `BoardGameObserver` interface. Also, all C participants were able to figure out that the map of figures and positions, `figureMap`, was inside `BoardDrawing` and that this is the one that has the pieces. However, they either lucked out or spent a long time, and they never saw this on the class diagram, even though there were aggregations. On an OOG, this would be obvious from the beginning, the E participants first used the OOG to formulate their hypotheses by using object relations on the diagram to describe a certain scenario of which objects will be communicating in doing a certain task. After that, they moved to the IDE to browse the code or traced to code from the OOG Viewer.

Most questions that developers in the C group asked are about object relations. Even when they looked at class diagrams, C participants were looking for object relations:

> "yeah class diagrams and charts are helpful for seeing associations" (C1, T1.a)

Our findings reflect how they mostly referred to CD6, since it shows associations and dependencies with the BreakThrough class (Table 5.6 for an example). This diagram was the closest to an OOG, but still did not show distinct instances. Having

access to the OOG helped E participants think about distinct object instances, for example, E2 asked whether there was only one `Position` object and E5 said there could be a pair. The OOG Viewer enabled them to trace to multiple instances of Position, i.e. from, to, etc. (Figure 4.2). OOGs helped them also know which objects are accessible from which objects and help them trace to the instance declarations directly in the code by narrowing the scope by displaying a limited number of traceability options. E4 was searching for the object representing the chess board and was looking for a starting point and thought that position would be the best to start from. This is what he found:

> "the question is there is got to be a position, so if i'd go to somewhere ... so what references the position ah okay alright so that actually helps looks like there are just two objects" (E4,T1.b)

On the other hand, most of the time the C participants searched for all possible references of a certain type in Eclipse which displayed many occurrences to select from. They also used the file search which often displayed JavaDoc comments in the code. They often picked the first occurrence which led them to wrong assumption most of the time.

OOGs also help developers keep grouping objects in logical tiers so they keep in mind that instances of `Game`, `Command`, and `Tool` are in the `CONTROLLER` tier and instances of `Drawing` are in the `MODEL` tier. All participants including the ones in the control group were experienced and aware that some classes were responsible for the `LOGIC` part and some objects were responsible for the `DATA` part which represent the `CONTROLLER` and the `MODEL` in our case. This means that developers could get this information even without using an OOG. However, none of the C participants seemed to connect this idea with how these objects were communicating even though they were provided with class diagrams explaining class relations in each role. This could possibly mean that making this piece of information visually obvious while the

developer is looking at object relations across tiers can help better respect the design and think within that scope. We provide details of how E participants used the OOG to provide their hypotheses in the form of object communication scenarios:

> "so we will have the board with pawns over there and then we will use move command then from `moveCommand:MoveCommand` we will access the `p:Position` of the pawn and then after we update the position of the pawn here `moveCommand:MoveCommand` I think we will go back here `game:GameStub` and then here to update the `drawing:BoardDrawing` from the change here [(FigureChangeListener) label on drawing] I think we will have the figure change listeners here and these guys [FigureChangeListeners] will render the changes maybe by using the [window:MiniDrawApplication]"(E2,T1)

All participants received the same instructions in the form of a tutorial on MiniDraw design. All were told that MiniDraw is built using the MVC design pattern. They also had access to the class diagrams showing relations between classes that represented these three different roles. They were also encouraged to use these diagrams. However, the C participants started thinking in terms of design only during the planning phase. When they started with the implementation phase, they seemed to ignore the design and started to get things up and running in any way possible, which caused them to introduce many hacks. Admittedly, some C participants made elegant modifications and were aware of the design, but that could be due to background knowledge and pervious professional experience. For example, C2 had several years of experience in programming and was familiar with Swing, frameworks and design patterns, yet he did not make the best code modifications and struggled with getting the position object. This also applies to some E participants. For example, C2 said: "when you first explained it to me I understood it, then I forgot everything".

Developers perception of the OOG and the OOG Viewer. The main source of information for the E participants was the OOG, but they were also using the

OOG Viewer to search for certain objects, expand object substructures, and navigate to the code from a Points-To edge between two objects. Unfortunately, in a few cases there were issues in the OOG and the OOG Viewer that hindered developers from answering some of their questions and affected their performance negatively.

For instance, the zooming and panning was counter intuitive which caused frustrations sometimes. We captured all the issues in the OOG Viewer that arose in the experiment and classified them by type and severity level, and how we were able to overcome them during the experiment. Some of them were not solved during the experiment, so we provide a future solution. We list some of these issues (Table 5.15). The *Notation: object label* shows participants' complaints about notation. The *Tool usability* shows usability issues. The participants also required new features to be added to the tool (*Tool feature*). We believe fixing these issues will yield even more significant results.

Our experiment also identified several instances when developers asked questions about the OOG related to soundness, precision, and graphical notation (Table 5.15). Some participants thought that some objects could be missing from the graph especially since they were searching for certain keywords to implement certain tasks. For example, one participant was searching for the editor object on the OOG. However, in the code the concept was implemented using different names (window, editor, etc.). On a class diagram, the closest concept to an editor would be the DrawingEditor interface, but the OOG did not have the editor as a name of an instance, so they were wondering if there is an instance in the code called editor, then why it did not appear when they searched the ownership tree or the object labels on the graph. Another example would be the adjuster object. Some participants thought that the graph could be more explicit in showing that `BoardDrawing` instance inside `GameStub` is of type `BoardGameObserver` which is implemented by `BoardDrawing`. Finally, the participants asked questions about OOG notation. Answers to question I5 and I7

Table 5.15: Issues in the OOG Viewer reported by some E participants.

Type	Question
Notation: object label	"do you show the interface or the class"
Notation: labeling types	"Ok, so lets see boardDrawing can we go to BoardGameObserver?[developers think that they can trace to labels]
Notation: hiding root object	"I'm searching for the starting point and it does not show in your diagram"
Graph: Edge lifting	"yeah its [expanding an object to see a solid edge] more work and the fact that its indirect usually isn't that important if you can get there you can get there."
Graph precision	"this is confusing! from and to positions are different."
Graph precision	"so the object adjuster is not only instantiated inside boarddrawing, but it is also instantiated somewhere else higher than boarddrawing? In the same variable name?"
Graph soundness	So why do you hide drawing editor although they are here. Developers want to display local variables within method bodies so they can watch the data flow
Tool usability: panning and zooming	"This is painful! I'm not gonna lie. I'm not trying to be bad, I'm trying to be honest!"
Tool usability: traceability	The traceability tool developers to the wrong line or was missing
Tool feature: reverse traceability	"can we go from the object in the IDE to the graph?"
Tool feature: traceability	"let's drill in to this guy [boarddrawing] so it implement Boardgameobserver so this guy is a BoardGameObserver! it was not clear to me that this guy. So it implements BoardGameObserver but I want specific things out of him. From the design of the tool id probably make it a little bit more explicit you just have to know that there is only one BoardGameObserver and it happens to be that specific concrete class"

(Table 4.8) helped us get developers' overall impression on the OOG and the tool:

> "I found it a little bit hard just navigating. The diagram is a lot of information.
>
> It's very concentrated"(E3, Exit Interview)

Despite these issues, the E participants thought the OOG and the Viewer were useful. We identified several scenarios where they expressed how they liked the OOG and certain features in the OOG Viewer and thought they were useful. For example, they used Points-to and Is-In-Tier facts about objects on the OOG to answer their questions. They also used traceability links and exploring object hierarchies in the ownership tree to verify their assumptions about Is-Owned and Is-Part-Of relations. Below are some quotes from the think-aloud of the participants:

> "so where is boardfigure? Now I'm gonna look and see who looks at the board-

> Figure. This is cool! actually I'm starting to like your tool okay so I have a moveCommand and I don't think that's it. I was just there so I'm gonna go to Mr. tool"(E4,T1.b)

> "so I'm gonna go inside this one [boarddrawing], even though I did not think it had anything to do with it because its name doesn't make any sense to me. So AbstractFigure::owned I was there and it wasn't there aha [figureMap] position daddaa"(E4,T1.b)

> "Is board figure the piece? or [you know what?], I'll look inside drawing to see if it has figures to make sure it is the board"(E5,T1.b)

The C participants were tutored on the OOG during the last 20 minutes of the study. They also expressed that the OOG could have saved them some time:

> "Ah that's why you have that [OOG]! thats very helpful."(C2, Exit Interview)

Chapter 6: Related Work

In this chapter, we discuss related work. We first discuss previous work on evaluating OOGs (Section 6.1). We then organize the discussion around diagrams of the code structure (Section 6.2), diagrams of the run-time structure (Section 6.3) and diagrams of other program representations (Section 6.4). We then mention some related work in the area of empirical evaluations (Section 6.5).

6.1 Previous evaluation of OOGs

6.1.1 Exploratory Study

As part of our work in evaluating the usefulness of the diagrams of the run-time structure, we conducted an exploratory study [2]. For the exploratory study, we used the JHotDraw framework [27] and had only three pilots. The tasks changed slightly between the pilots, so our analysis there remained qualitative.

6.1.2 Case Study

We also conducted a case study [1]. In the case study, we evaluated whether the extracted OOG was useful for code modification tasks, so we refined the extracted diagram to reflect the developers mental model. In that study, the architectural extractor had to do the refinement which caused the two parties to work separately. Furthermore, both the architectural extractor and the developer were building their knowledge about the code incrementally and refining the OOG accordingly which could mean that the developer was feeding the diagram with useful information not the other way around. Therefore, to avoid running a long study that could be wasting the developers time fixing the OOG which seemed to happen in the previous study, we conducted the case study discussed in Chapter 3, where the developer refined an

OOG extracted by the architectural extractors to convey the design intent in the code before any code modification takes place.

For the case study, we used the DrawLets subject system [17] and involved one participant (the author of this thesis), which limited the external validity of our previous findings.

6.1.3 Field Study

In the Scholia approach [3], the extracted diagrams had previously been evaluated by comparing them against a target architecture, to analyze conformance and communication integrity. The humans in the loop only generated the target architecture and interpreted the analysis results. In our work, we evaluated the usefulness of the extracted diagrams to developers in the context of code modification tasks.

Abi-Antoun et al. previously conducted a field study to observe how developers understand object relations, and what tool features they need to convey their mental model of the system [2, 7]. They provided a professional developer with an initial object graph and refined it to convey his design intent, but they did not have the professional developer use the refined diagram to make any code modifications.

6.2 Diagrams of the Code Structure

6.2.1 Code Structure Exploration Tools

Several code exploration tools display the code structure differently from textual editors. A lab study of several code exploration tools found no significant benefits to any of them [14]. In our work, we focus on the runtime structure which, for object-oriented code, is quite different from the code structure.

6.2.2 Studies on Usefulness of UML Diagrams

Several researchers have evaluated empirically the usefulness of UML diagrams [22, 18, 10]. Hadar et al. [22] have studied the contribution of UML diagrams in program comprehension. They found that developers often need all types of UML diagrams and integrate the information they get from each one to understand and analyze the program. They also found that developers even sort diagrams by the type of information they can get from them. Developers know to use sequence diagrams to understand the dynamic behavior and class diagrams to study static relations. These evaluations, however, focus on class diagrams and partial views such as sequence diagrams. To our knowledge, our study is the first to evaluate global object diagrams of the entire system, which had been difficult to obtain using technology prior to Scholia.

6.2.3 Enhanced Class Diagrams

Riehle worked on the use of "collaboration roles" to enhance class diagrams with information about design patterns. Riehle used a class diagram of the core classes in JHotDraw and added a role modeling interpretation to that diagram [44]. He found that the role modeling adds more information to the existing documentation. In our approach, we present design patterns in the existing code, however, we do not add any extra information to the existing diagram; we extract diagram that reflects the code as is and thus convey information that already exists in the code.

6.3 Diagrams of the Run-time Structure

We distinguish between diagrams that are extracted using static analysis, from those that are extracted using dynamic analysis.

6.3.1 Use of Object Diagrams

Dekel and Herbsleb [15] conducted an observational study and confirmed that developers do employ object diagrams, together with class diagrams. Their study, however, focused on design-time diagrams. We assessed the usefulness of object diagrams for coding tasks.

Recent empirical work is paying greater attention to understanding the run-time structure of an application. Lee et al. [35] report on an empirical study where a participant expressed the need to understand "how objects connect to each other at run-time when I want to understand code that is unknown: an object diagram is more interesting than a class diagram, as it expresses more how [the system] functions".

Points-to and shape analysis. The hierarchical object graph extracted by SCHOLIA is a points-to graph. Similarly, many static analyses extract points-to or shape graphs, with the stated goal of aiding program comprehension. Most of these graphs are non-hierarchical, however. Also, to our knowledge, their results have not been evaluated empirically with developers asked to perform coding tasks.

6.3.2 Statically Extracted Diagrams

Fully Automated Object Graph Extraction. A number of existing approaches extract, fully automatically, various non-hierarchical object graphs, including WOMBLE [26], AJAX [38], PANGAEA [49]. While these approaches can be useful for showing object interactions, they share a fundamental scalability limitation: for programs of any size, they will produce a diagram with so many objects that, in practice, the diagram will be barely readable by humans. To our knowledge, these tools have never been evaluated in a laboratory study in the context of code modifications.

Annotation-Based Object Graph Extraction. [31] proposed a type system and a static analysis whereby developer-specified annotations guide the static abstraction of an object model by merging objects based on *tokens*. Their approach supports a fixed set of statically declared global tokens, and their analysis shows a graph indicating which objects appear in which tokens. Since there is a statically fixed number of tokens, all of which are at the top level, an extracted object model is a top-level architecture that does not support hierarchical decomposition, thus limiting the scalability of the approach to large systems. The SCHOLIA [3] approach extends Lam and Rinard's both to handle hierarchical architectures and to support object-oriented language constructs such as inheritance.

Visualizing object-oriented programs using Reflexion Models. Walker et al. [51] developed an approach for visualizing the operation of an object-oriented system at the architectural level. Their approach builds on the Reflexion Models technique, but uses the running summary model rather than the complete summary model. They allow developers to flexibly define the structure of interest, and to navigate to the resulting abstracted views of the system's execution. Approaches that rely on static information can often rely on the iterative mapping approach, and their approach relied on dynamic information which limits iteratively updating the mapping. Richner et al. [43] proposed a complementary approach to Walker's work that uses both static and dynamic information to answer developers questions about object-oriented code. Their study focused on reverse engineering HotDraw and trying to understand it, but did not involve any code modification task.

6.3.3 Dynamically extracted diagrams

Several researchers used dynamic analysis to extract information about object relations.

Dynamic Object Process Graphs. Quante evaluated in a controlled study Dynamic Object Process Graphs (DOPGs) [40]. A DOPG is quite different from an OOG. A DOPG is a statically extracted inter-procedural Control Flow Graph (CFG), shown from the perspective of one object of interest, with the uninteresting parts of the CFG removed based on a dynamic trace. So, a DOPG is closer to a partial call-graph than to a points-to graph. Quante found that the DOPG helped for concept location tasks on one code base but not another. Quante presented only success and time numbers, so it is unclear how participants were using the diagram or why exactly it helped only sometimes.

Diagrams designed to capture object relationships. Demsky and Rinard used dynamic analysis to extract two types of role-based object diagrams, role transition diagrams and role relationship diagrams [16]. Role transition diagrams help developers understand the different roles that the different instances of a class play in a computation. Role relationship diagrams help developers understand relationships between objects of different classes. While an OOG provides developers with a global view of the object structures of the system, the role-based object diagrams consider only one execution path in the program.

Tools to visualize ownership structures. Hill et al. [25] and Potanin et al. [39] built tools to visualize ownership structures. Mitchell et al. [37] built the Yeti tool for diagnosing storage inefficiencies and lifetime management bugs. These tools use dynamic analyses and have not been evaluated for their usefulness in assisting developers with code modification tasks. By definition, a diagram extracted using dynamic analysis is partial, so developers cannot base their decisions on such a diagram to make all of their modifications.

6.4 Diagrams of Other Program Representations

6.4.1 Call graphs

Many tools focus on call graphs, which seem like an intuitive model that enables developers to understand interactions between different parts of the code [24, 33]. For instance, LaToza and Myers [33, 34], in their Reacher tool, statically extract path-sensitive call graphs. However, Reacher does not track objects, to avoid an exponential blowup in the number of paths to display. We consider call graph visualizers to nicely complement tools which show object diagrams. Call graphs focus on the control flow structure. Object diagrams, with points-to edges, focus on the object reference structure.

6.5 Other empirical studies

6.5.1 Studies evaluating design patterns

Some studies have evaluated design patterns [19]. The emphasis was on the time-to-complete a task, not on whether the participants made changes that respected the design or followed good object-oriented design principles. In this thesis, we did not focus on specific design patterns in isolation, since the same object may be involved in the realization of multiple design patterns at once.

6.5.2 Studies on object-oriented frameworks

Modifying an object-oriented framework is especially challenging, since frameworks typically involve many design patterns. This is backed by previous empirical evidence [47, 28, 32], including our own exploratory study [2].

Kirk et al. [28] recognized the importance of understanding the object structure in object-oriented frameworks: “understanding the dynamic behavior of a framework

is more challenging, particularly given the separation of the static and dynamic perspectives in the object-oriented paradigm". Shull et al. concur that both "the static and dynamic structures must be understood and then adapted to the specific requirements of the application [...] For a developer unfamiliar with the system to obtain this understanding is a non-trivial task. Little work has been done on minimizing this learning curve" [47]. This controlled experiment demonstrates that OOGs seem to be a step in the right direction to help developers understand and reuse object-oriented frameworks.

6.6 Summary

We were heavily inspired by previous empirical work evaluating the usefulness of diagrams for program comprehension, and in particular, diagrams which participants used in the context of making code modifications. To our knowledge, our work is the first controlled experiment evaluating the usefulness of global, hierarchical object diagrams depicting the run-time structure, and which are statically extracted. These diagrams and our work focus on the questions or facts about objects and their relations that developers need to understand while evolving object-oriented code.

Chapter 7: Discussion and Conclusion

The goal of this thesis was to evaluate usefulness of OOGs to developers performing code modifications on object-oriented code. We designed and performed a controlled experiment to investigate whether having access to OOGs, in addition to available tools and diagrams such as Eclipse and class diagrams, can improve developers performance.

In this chapter, we discuss how we validated our hypotheses (Section 7.1), some perceived threats to validity and how we mitigated those threats (Section 7.2). Then, we discuss some limitations in our experimental design (Section 7.3). Finally, we mention some future work (Section 7.4) and conclude.

7.1 Validation of Hypotheses

We measured three variables to assess developers performance: code explored, time spent, and success on a task. Our results indicate that having access to OOGs in addition to other sources of information improves developers' performance. Two of the developers who had access to diagrams of the run-time structure completed the three tasks compared to only one developer who had access to only class diagrams. On average, developers who had access to diagrams of the run-time structure performed their activities in less time (22%–60%), and by browsing less irrelevant code (10%–60%). Our results did not indicate a significant difference across all tasks, but we found that for some of the activities in which the developers engaged, the difference was significant. The significance may be due to the fact that those activities required thinking about relations between object, which was always easier to find by using OOGs than by only browsing the code in the IDE or looking at class diagrams. Maybe selecting tasks designed more specifically to measure usefulness of OOGs would give more significant findings.

7.2 Threats to Validity

Our experiment may have several threats to validity. In this section, we discuss how we tried to mitigate them.

7.2.1 Construct Validity

Three of our hypotheses were measured using three dependent variables, i.e., success rate, code explorations, and time to task completion. We used automated tests to measure the significance in the mean difference between two groups for the second and third variables. While the data for the last two tasks do suggest that the time to task completion is longer in the case of the C group, the lack of statistical significance across the three tasks and the high standard deviation of the C group and sometimes the E group make it difficult to say whether this depends on the participant or the group as a whole. Overall, the results indicate that there were outliers in each group. This is probably because our sample size was small, so the analysis treated some individual participants as outliers. We used box plots mainly as extra support to show whether there were any indicators of dispersion (skewness) and outliers in our data. However, these plots are usually preferred when the number of observations is large. If we increase the sample size, the number of outliers may be less.

We could not achieve statistical significance across all tasks for both variables. However, our analysis included not only tasks but also activities within tasks related to object relations, and how these activities were accomplished with or without having access to OOGs. The results suggest that the difference is significant in the first task for the second measure and in the third task for the third measure. The significance was even higher when we analyzed the activities in these tasks. These results could be because those activities were directly related to seeking answers for questions related to object relations, and having access to OOGs made these relations visually obvious.

Also, some features in the OOG Viewer made it easy for developers to trace directly to field declarations in the code.

To avoid being biased, we recruited participants from other research areas and we did not include any students who had previous knowledge of OOGs and ownership annotations.

One might argue that the study design was biased because the architectural extractors refined the OOG and that we should have provided the participants with the extracted OOG as-is, before refining it. OOGs are extracted semi-automatically from the code using static analysis. We make an OOG relevant for code modification tasks by making it convey design intent and reflect the developers' mental model of the system (see Section 3.6.1). Admittedly, it may be a better idea to give the participants the ability to interactively refine the OOG to make it more useful to them. This is an area of future work (Section 7.4.2).

7.2.2 Internal Validity

According to our experimental design and procedure, we may have several extraneous variables that could have influenced the participants other than the independent variable alone. For instance, the level of experience in programming, and in Java in particular, and familiarity with Eclipse varied among participants (Table 4.2). Three of the E participants and one of the C participants had a previous experience that could have affected their performance on the task (Table 5.14). Nevertheless, our findings show that one of the three E participants had a bad performance compared to his colleagues (E1) and the other two (E3 and E4) still struggled with modifying a framework designed by others and needed the OOG to answer their questions. This indicates that evolving an application implemented using an object-oriented frameworks is challenging regardless of experience. Actually, the experiment revealed that most developers including experienced professionals know about object-oriented

frameworks and design patterns, but they do not seem to apply them in their everyday coding activities. Several participants complained that the framework might have been over-engineered:

> "At this point I just want to know what the person who implemented the class meant by it"(E4,T1)

> "but you said it's heavy in design patterns and some times when you go heavy in design patterns you can make it more complicated that necessarily needs to be"(E4,T3)

We tried to mitigate the differences between experience levels among participants by randomly assigning participants to both groups. It could be that we had more professional developers in the E group, but a closer look at the level of experience in Java justifies this. We had three professional developers in the OOG group, but some of the participants in the control group had the same level of experience in Java. For example, one participant in the C group was an undergraduate student, and had 3 years of experience programming in Java. Also, one participant in the E group was a professional developer who had a bachelor degree with 3 years programming in Java. Another participant in the C group had over 20 years of experience. So overall, we had two professional developers in the E group with over 20 years of experience and one professional developer in the C group with a similar level of expertise. We also had 2 developers who had the same experience, but one is professional and the other is not. Also, the participants had access to several resources such as the IDE, class diagrams, documentation in addition to OOGs. As a result, to avoid any possible confounds to the study results, we classified the different parts that influenced the participants efficiency into three categories as shown in Table 5.14.

Second, there could be threats to validity due to the experimenter treatment. The experimenter tried to instruct the participants in the same way by providing

the same information. Still there were questions from each participant which were different. The goal of the experiment was clear to participants, i.e., usefulness of certain types of diagrams, but the participants were not aware of the specific outcomes the experimenter expected from the experiment, e.g., speed of modifications, success rates, navigation paths, etc.

Also, it could be that a 20-min tutorial on the OOG and the tool was too long, but we wanted to mitigate the challenges that the participants face when dealing with an unfamiliar tool. The evaluation approach that we followed in the experiment could have provided opportunities for participants to incrementally learn how to navigate the tool in related tasks. One could argue that these exercises were triggering some potential uses of the OOG. However, from our previous experience, it could take a professional developer many months to understand the idea or rationale behind OOGs and the concept of ownership domains in general. Also, given that we had a control group, this should not be an issue since we are measuring the usefulness of OOGs and the difference in performance between having access to OOGs vs. other standard sources of information. Therefore, the purpose of the hands-on exercises was to ensure that the participants knew how to use the OOG effectively and were not just playing with a new tool.

Also, we demonstrated the OOG tutorial on MiniDraw, but the experimenter chose random objects on the OOG rather than specific objects related to BreakThrough (Table 4.6). Admittedly, we should have demonstrated it on another, unrelated application. Still, our findings do not suggest that the participants benefited from the tutorial to understand MiniDraw. In fact, some participants stated this explicitly:

> "What I'm seeing right now is am trying to learn two things: the visualization paradigm and the structure of the code." (E5,OOG Tutorial)

We could have given the impression that giving the C group 10 class diagrams was overwhelming compared to only 2 OOGs and an interactive viewer. However, one of

the class diagrams helped the C participants get a global view of class dependencies and associations with the BreakThrough class. The other 5 diagrams were just broken down versions of this diagram per package and were less cluttered. The remaining 4 diagrams were manually generated by MiniDraw designers to explain the class relationships in the Model, View, and Controller roles. In fact, the C participants found some of these diagrams useful and incorporated them while they performed the modifications as explained in Section 5.3.

Another technique that the experimenter used is the questionnaires between the tasks. It could be that these questionnaires were triggering questions about object relations, and one might argue that the developers did not ask the questions about object relations or structure themselves. This could be true, but the participants were still asking those questions in indirect ways and sometimes using different terminology. It was also clear from their navigations that the participants were seeking certain relations related to the object structures rather than the type or call hierarchies. Also, these questionnaires measured their level of comprehension as they made their changes and provided them with little guidance especially since we did not correct their answers; we only observed how they changed these answers or navigation paths as they discovered new relations using the OOG with the help of the OOG Viewer.

Finally, although we asked the participants to do the tasks in sequence, we gave them the instruction sheet which included description of all the tasks. As a result, one might think that the participants thought about some of the tasks beforehand which could have affected their behavior. However, all of the participants, except one in the E group (E4), planned then implemented each task in sequence, for the three tasks. In a future study, it may be a better idea to provide the participants with a task description only after they finished the previous one.

7.2.3 External Validity

Several factors could affect the generalizability of our findings. First, the subject system itself may not be representative of all code bases. As we discussed in chapter 4, MiniDraw is a framework that can be customized to implement several applications such as drawing applications. In fact, when we annotated MiniDraw we had several versions the extracted OOG each representing a different application, e.g., RectangleDraw, LogoPuzzle, etc. The OOGs of these applications are slightly different from the OOG of BreakThrough. For example, the `selectionTool` object and its corresponding tracker objects make more sense in the case of the RectangleDraw application. Also, different tasks could be designed to modify RectangleDraw such as implementing a view-specific undo feature. Therefore, conducting the study using the other applications of MiniDraw could yield different results.

Second, our tasks may not be representative of real maintenance tasks since they were crafted by the experimenter, and we did not extract them from a bug tracking system. Our results could have been more generalizable had we used real bugs or feature requests. However, the tasks were not carefully crafted to specifically illustrate the usefulness of OOGs. Admittedly we dropped some of them after the pilot, but that was either due to timing constraints or because the task was trivial and did not require asking any questions about object relations.

Third, our experiment mainly targeted graduate students, but we had 4 professional developers which gives external validity to our results. Still, some of our participants came from a C# or C++ background. Probably the results would be more powerful if we recruit only experienced Java developers, but this is a limitation shared by many published empirical studies.

Finally, for maintenance tasks, usually the maintainer of the system is knowledgeable about the code base. This could be an issue in controlled experiments which cannot avoid the artificial laboratory setting. A field study could give better results

especially since we are measuring the OOG usefulness for code maintenance.

7.3 Limitations in the Experimental Design

Our experimental design had some limitations that could have interfered with our results. In this section, we discuss some of these limitations.

7.3.1 Disabling the Feature to Inspect Types

The OOG Viewer has the ability to display the inheritance hierarchy of the types of the field declarations in the program that an object on the OOG merges. We disabled this feature for the study since we wanted to avoid the situation of the participants using the OOG Viewer mainly to generate class diagrams. We wanted the usage of the OOG Viewer to count as dealing with the runtime structure. We believe that making this feature available would have made the tool even more useful to the participants. We know this from a previous field study [7], where an experienced programmer told us that he loved this feature. Together with the trace to code feature, it allows relating the runtime structure back to the code structure better. It is often helpful to understand why the OOG merged objects of disparate types (it often meant that all these types were related by inheritance). In particular, it helps developers to see the classes that may be behind an interface.

7.3.2 Stripping the Annotations from the Code

To avoid additional confounds to the study, we stripped the annotations from the code that we gave to the participants. Stripping the annotations caused the traceability to code feature, in some cases, to take the participants to the wrong line of code, but to an area around it. This was because the participants were adding more code to the files and formatting their code all the time. Presumably, keeping

the annotations in the code could have produced better results, but the participants could have found the annotations more useful than the OOG itself.

7.3.3 Focusing on the distinction between private and public domains

When we provided developers with the OOGs, we had hoped that they might benefit from the distinction between public and private domains (Section 2.1) to make changes that do not violate the design. In this study, we did not find strong evidence that developers do benefit from that distinction, except in a few cases.

The instruction sheet did describe the difference between the two types of domains. However, because we provided the participants with a version of the code with the annotations stripped out, that distinction may be too subtle to grasp just by looking at an OOG. Without the annotations in the code that are checked with the type checker, a participant would have to notice that an object is in a private domain in the OOG (thick bordered domain) and ensure that he did not break the encapsulation by adding a public method that returns an alias to a private field in a private domain. Giving the participants the code with the annotations may have brought this distinction to the surface. It could also be that visual distinction between private and public domains on an OOG is not very clear, and that we may need to fix this in future work.

7.3.4 Focusing on the Notion of Object Hierarchy

The notion of object hierarchy, i.e., which object is part of which other object, is not explicitly expressed in object-oriented code. In Scholia, we use annotations to express that notion, and the OOG attempts to convey that information visually. Our findings show that developers benefit from hierarchy to answer their questions, but

these findings were not true across all developers. It could be that we need to explain this notion more carefully in our tutorial.

7.3.5 Focusing on Advanced Features in the Diagram

Another useful information we provide to developers is the points-to edges between objects. This was useful, as our results indicate. However, the developers in our experiment were not able to use the direction of communication between objects across architectural tiers to explain useful scenarios. They fully understood that the diagram showed points-to relations, i.e., how to get from one object to another. But they seemed to not grasp some of the other information that the diagram can convey such as labeling types (showing listener interfaces) to show how objects are communicating using listeners. It could be that we need to enhance our visualization to reflect these concepts more explicitly. We may not have provided enough tutoring on how a labeling type on an object, e.g., `DrawingChangeLister` on `stdViewWithBackground`, together with a points-to relation, e.g., a list of `DrawingChangeListeners` inside the `boardDrawing` object, can help explain that the `stdViewWithBackground` is registered as a listener to the `boardDrawing` object.

7.3.6 Selecting Tasks That Require Heavy Knowledge About Object Structures

When we designed our tasks, the main purpose was to add the game logic to BreakThrough which was missing. The only thing that was implemented in the version that we gave to our participants was the board as a drawing with the pieces on it. In fact, some of the functionality already existed which made it hard for the participants to implement the change since they had to search for the existing code instead of duplicating what is already there. For example, the participants were able

to move pieces by dragging them on the board the participants, which made them wonder whether the movement was partially implemented. Also, the concept of a player was not implemented, so they had to use drawing components, e.g. color, to do the change.

However, two of the tasks were related, which explains why the difference was significant for the second measure in the first task but not in the second. There was a difference in the first measure increased when they moved to the third task, presumably, because this task was different from the other two tasks. In general, we believe that some tasks required less information from the OOG than the others. We could have obtained better results by designing tasks that specifically require asking questions about the objects, questions that would be hard to answer just by looking at the class diagrams or by browsing the code in Eclipse.

7.4 Future Work

We foresee the following areas of future work.

7.4.1 Richer Information in the OOG

Currently, the OOG has edges which show only one kind of information, i.e., points-to relations between objects. In recent work, Abi-Antoun and Rawshdeh have enriched the object graph with some usage edges, namely dataflow edges [42, 41]. It may be instructive to conduct another experiment to re-evaluate the usefulness of OOGs with the additional information.

7.4.2 Interactive Refinement of the OOG

The Scholia approach does not currently support the interactive refinement of the OOG by direct manipulation. Developers using the OOG have to rely on the

architectural extractors to refine the OOG for them. It may be useful to give the developers the ability to edit an OOG on the fly, rather than just view it, in order to make it reflect their design intent. In recent work, Abi-Antoun and Selitsky have already developed a front-end for an interactive editor for the OOG [6, 46]. The main challenge when allowing for the iterative refinement of an OOG would be maintaining the diagram's soundness. It may be instructive to conduct another experiment in which the participants would have the ability to refine the OOGs and potentially make them more useful or relevant, while they undertake code modification tasks.

7.4.3 Improving Usability of the OOG Viewer Tool

This experiment, in addition to our previous studies [2], gave us several insights on how to improve the OOG Viewer to avoid issues that could have interfered with the usefulness of OOGs to developers. Future work will include fixing cognitive issues in the current visualization that may have hindered diagram understanding. Enhancing the features dealing with traceability and search could also improve the tool's usability.

7.5 Conclusion and Broader Impact

In this thesis, we provided the first empirical evidence that having access to global object diagrams improves developers' performance. We observed 10 developers, organized into an experimental and a control group, use the diagrams to perform realistic code modification tasks. Two of the developers who had access to diagrams of the run-time structure completed the three tasks compared to only one developer who had access to only class diagrams. On average, developers who had access to diagrams of the run-time structure performed their activities in less time (22%–60%), and by browsing less irrelevant code (10%–60%) than developers who had access to

only class diagrams or who did not use diagram at all.

Our findings are promising even though they did not indicate a significant improvement in developers' performance overall, which does not contradict our research question. Still, our results provide evidence that global object diagrams, which make explicit the relations between objects across run-time tiers, significantly improve developers' performance in some of the developers activities, namely ones which required answering questions about relations between objects. When asked to do maintenance tasks on object-oriented code with which they were unfamiliar, developers benefited from global object diagrams to locate and perform the changes much faster than by just browsing the code in the IDE or by looking at class diagrams. We also captured individual differences across all developers based on diagram usage. Most of the developers used the diagrams as maps to locate where to make the change especially when they explored the code for the first time. Those who did not use diagrams performed worse than their colleagues.

Given the considerable costs of software maintenance, diagramming tool support becomes a necessity for developers which justifies researchers efforts in developing diagramming tools. However, a slight improvement in developers performance justifies moving the focus away from supporting mainly class diagrams, and instead, investing more effort in research and tools to extract object diagrams, and improving the tools' usability and usefulness.

Appendix A: Pre-screening Material

A.1 Object-Oriented Programming Test[1]

Title of Study: *Assessing the usefulness of diagrams of the runtime structure for code modification tasks*

Practical Test on Basic OO Skills Using Eclipse (15 min)

Part 1: Objects and Classes

Consider a game-like predator/prey simulation:

Rabbit eats Grass
Fox eats Rabbit

The procedural code to represent this simulation is:

```
For each animal "A"
if (A is a rabbit) eatGrass(A)
if (A is a fox) eatRabbit(A)
```

1. Write the corresponding object-oriented code:

```
For each animal "A"
```

Consider adding an element to our simulation:

Eagle eats Fox and Rabbit

Notice that there will be a significant impact on the existing simulation, since we must change all code that does different things for different animals (eating, moving, breeding, etc.).

2. Use object-oriented design to solve this problem.

```
For each animal "A"
      if (A is a rabbit) eatGrass(A)
      if (A is a fox) eatRabbit(A)
if (A is an eagle) eatSmallAnimal(A)
```

[1]The test reuses slides on object-oriented topics by Jonathan Aldrich.

Appendix B: Main classes in MiniDraw

MiniDraw is based on the Model-View-Controller (MVC) architectural pattern, commonly used in graphical user interface applications. MiniDraw's architecture uses many design patterns. This is common for frameworks as frameworks aim to achieve a high degree of flexibility to allow customization. In MiniDraw, `Drawing` is the Model in the MVC pattern with some additions to handle modifications of the set of figures and the selection (see Fig B.1). `DrawingView` plays the central role in the MVC pattern (see Fig B.2). It defines four layers of graphics, drawn in order: background, drawing, contents, and it responds to change events from its associated `Drawing` and ensure redrawing forward all mouse and key events to the editor's associated tool. `Tool` is the controller (see Fig B.3). `DrawingEditor` is the main class of a MiniDraw application, that is the editor must instantiate all parts of the application (see Fig B.4).

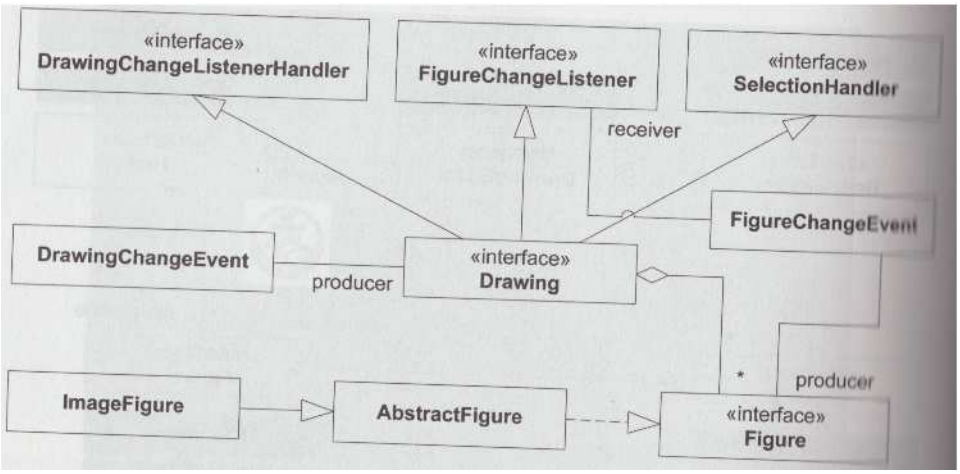


Figure B.1: UML class diagram of the roles in the model part [13].

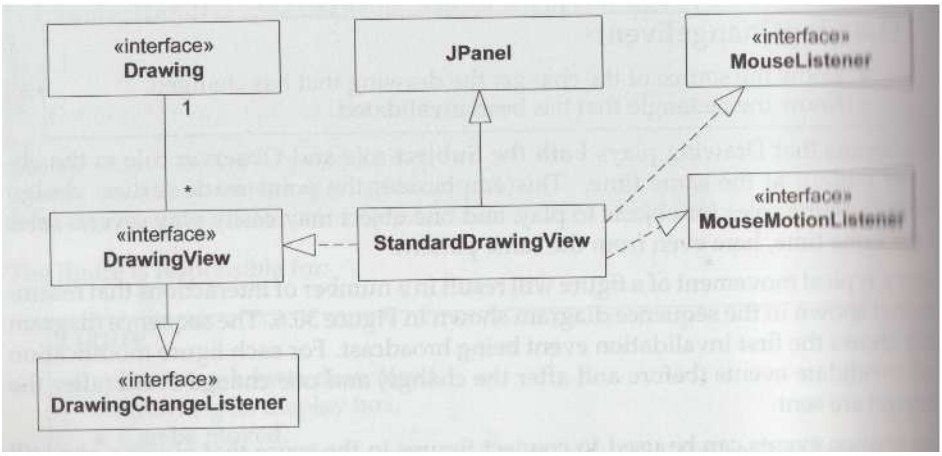


Figure B.2: UML class diagram of the view part [13].

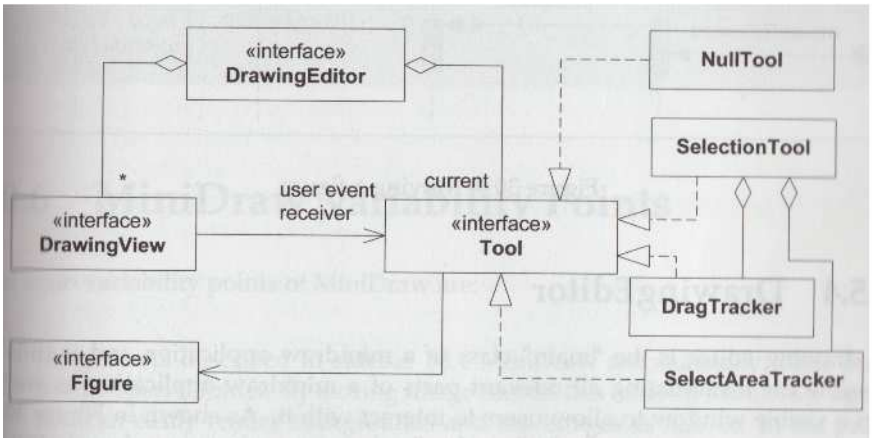


Figure B.3: UML class diagram of the tool part [13].

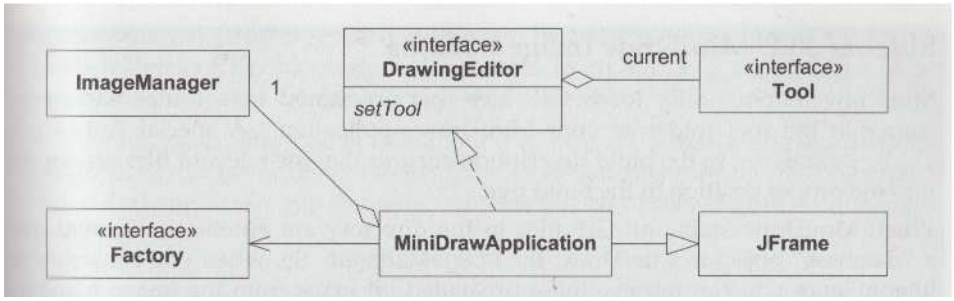


Figure B.4: UML class diagram of the editor part [13].

Appendix C: Eclipse Navigation Tutorial

[1]

Eclipse features a number of sophisticated features for navigating through source code. In this brief tutorial, you will try out several of the most useful features.

1. Press CONTROL-SHIFT-T and type "breakthrough" to open up the breakthrough class.

2. Press CONTROL-O to open the method outline and type `addObserver()` to navigate to the method `addObserver()`.

3. Hold down control and click on `addObserver` (the method call on `gameStub`, not the method declaration). This moves you to the method declaration.

4. Go back to the previous method you were looking at by clicking on the left arrow second most from the right edge of the toolbar. Alternatively, you can use ALT-LEFT-ARROW.

5. Place the cursor over the `addObserver()` method declaration, right click, and select "Open Call Hierarchy". At the bottom of the screen, a tree view shows either the Callee Hierarchy or the Caller Hierarchy. Switch between them by clicking a button to the right of the Call Hierarchy tab.

6. Place the cursor on `gameStub`. All references to this field in the current Java file are now highlighted with a grey highlight both in the text editor and next to the scrollbar, allowing you to quickly find all of the references in the file.

7. Find all of the methods that assign the field `gameStub` by placing the cursor on top of it and selecting from the "Search" menu at the top of the screen "Write Access" and then "Project".

[1] We reused a tutorial by Thomas LaToza.

8. Find all references to the `Game` type by selecting Java from the Search menus, typing in `Game`, and selecting "Type" in Search For.

Appendix D: Instruction Sheet

You will be asked to perform three change tasks. You will have 2.5 hours (beginning after you finish reading the actual task description) to work on all the tasks. The tasks have been designed to be challenging, so you will likely not have time to investigate or design as much as you could with unlimited time constraints. Your goal is to complete each task with a well designed implementation that respects the existing architecture of the system as much as possible.

You will be trying to fix bugs and add features to the existing code base that you will have little time to explore. ***So do not feel discouraged if you are not able to accomplish as much as you might hope you could accomplish***. To help you go through all the tasks, you may do the modification in two phases:

First, locate where the change should go in the code by adding a comment. The comment should include the task number and a pseudocode or a brief description of how you are planning to do the modification. Notice that you do not have to provide the complete algorithm or the implementation details. However, you do need to provide all the objects that will take place and the method call hierarchy. The code below is an example of an acceptable comment:

```
//Task1: TODO: get object obj1 from obj2
// call method m1 on obj1
```

After locating the change in all tasks, you need to implement the modification. You may not be familiar with some technologies used to implement the existing system (e.g., Java Swing), so you can refer to the sample code in the appendix.

You will be provided with a copy of Eclipse 3.5. You may use any feature of Eclipse including running the program. But you may not use any other application (including a web browser). You should perform the tasks in the order specified. After each task, the experimenter will ask some questions about your understanding of the system,

how you were working, and decisions you made in creating your implementation.

To understand how you are working, you will be asked to "think aloud" — talk about what you are thinking while you work. You should describe what you are trying to accomplish and what you are considering doing to accomplish your goals. If you find information that you have been seeking, you should make sure that you say something about what you found. If you get distracted or forget to think aloud, the experimenter will prompt you to continue to think aloud by asking you about what you're doing. To allow us to analyze how you work, your computer screen and audio will be recorded using a recording program. You may not disable the recording during the study.

Now take a few minutes reading the task description.

Note: If you are not familiar with Java Swing, you may find the following code snippet useful:

```
JFrame frame = (JFrame) window;
JMenuBar menuBar;
JMenu menu, submenu;
JMenuItem menuItem;
menuBar = new JMenuBar();
menu = new JMenu("Menubar");
menuBar.add(menu);
menuItem = new JMenuItem("MenuItem");
menuItem.addActionListener( new ActionListener() {
public void actionPerformed(ActionEvent e) {
// your code should go here
}
});
menu.add(menuItem);
frame.setJMenuBar( menuBar );
```

Appendix E: All Diagrams Provided to Participants

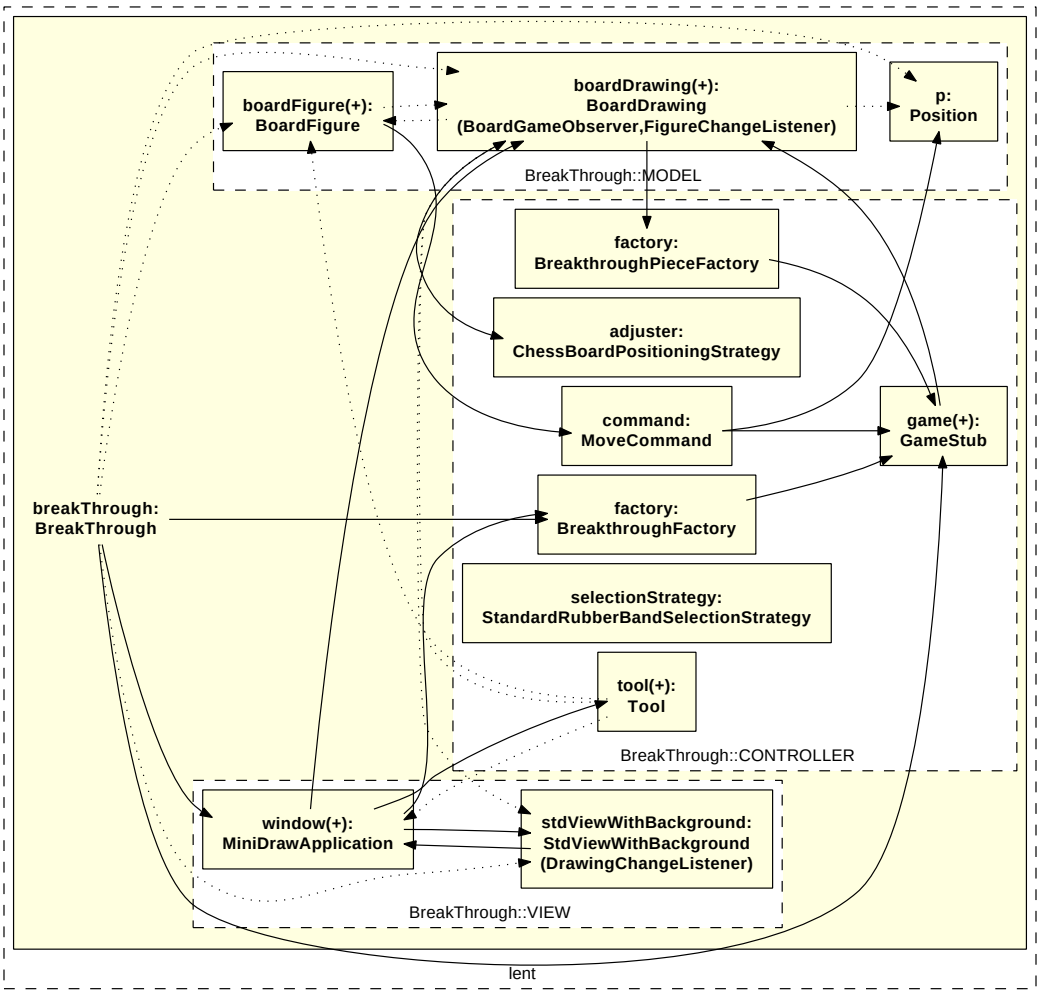


Figure E.1: An OOG for BreakThrough.

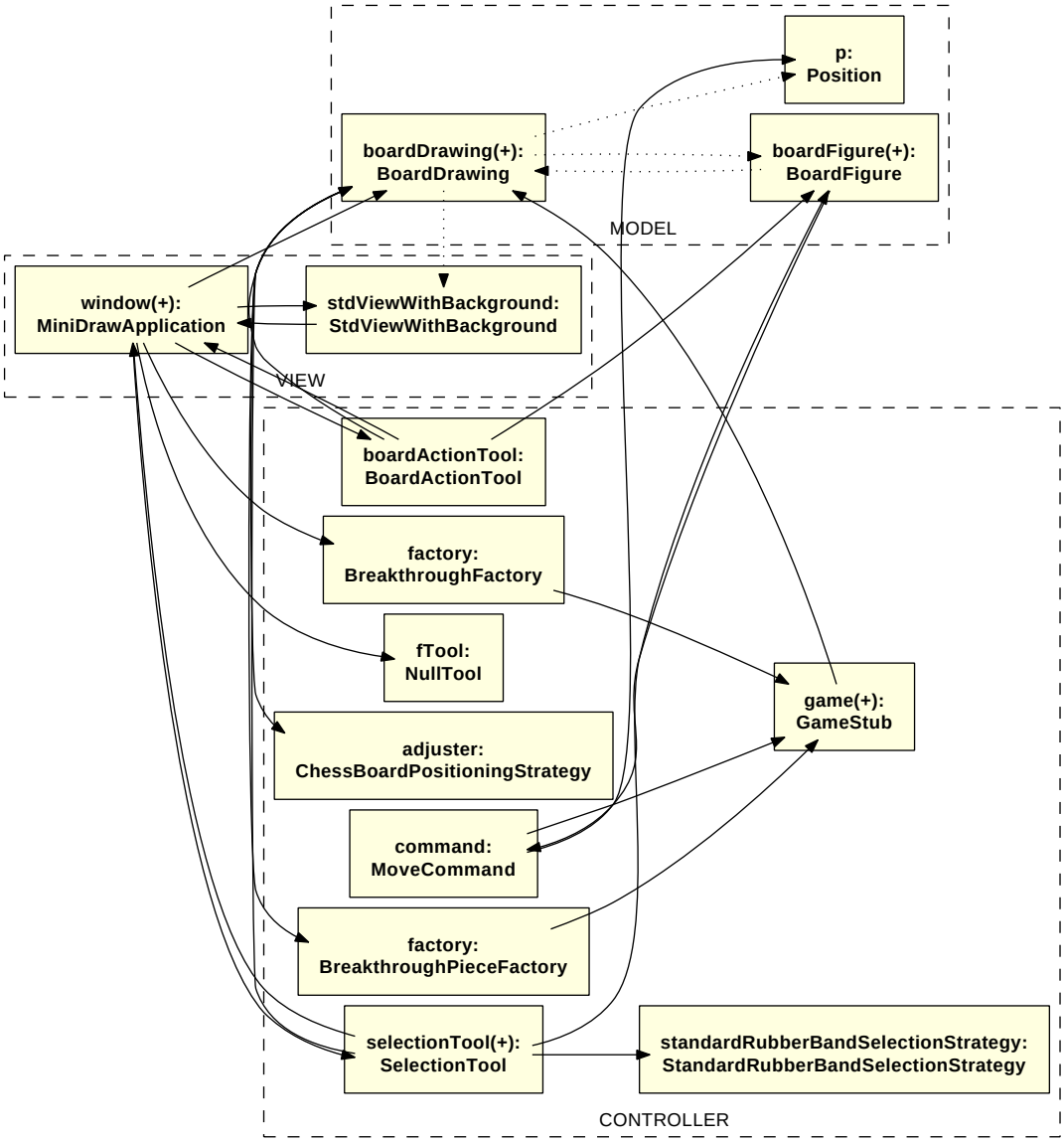


Figure E.2: An OOG for BreakThrough (less abstract version).

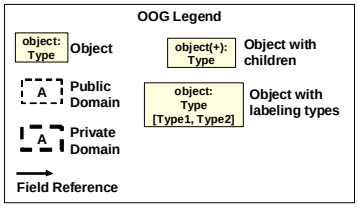


Figure E.3: OOG notation.

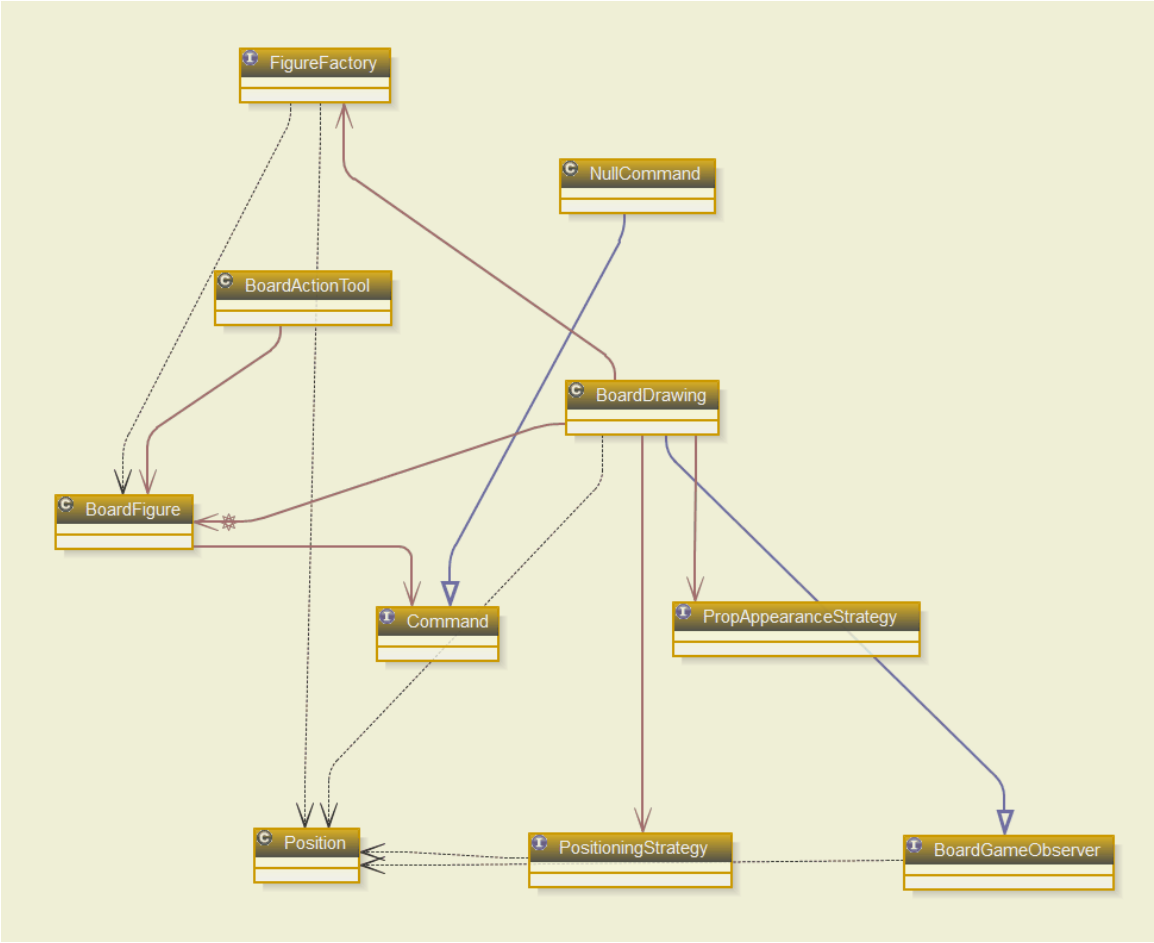


Figure E.4: Class diagram of classes in the boardgame package.

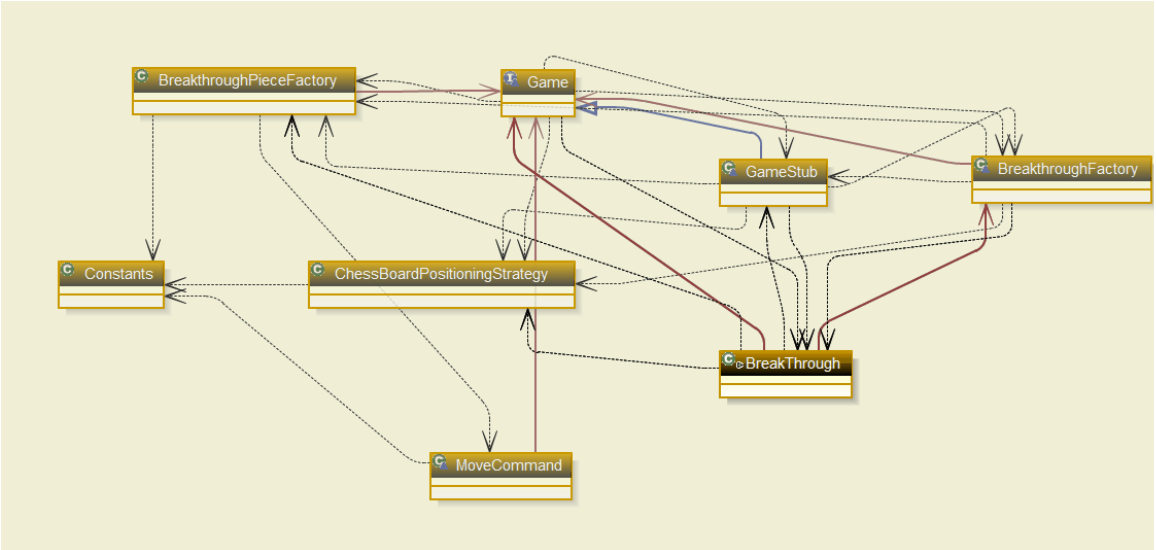


Figure E.5: Class diagram of classes in the breakthrough package.

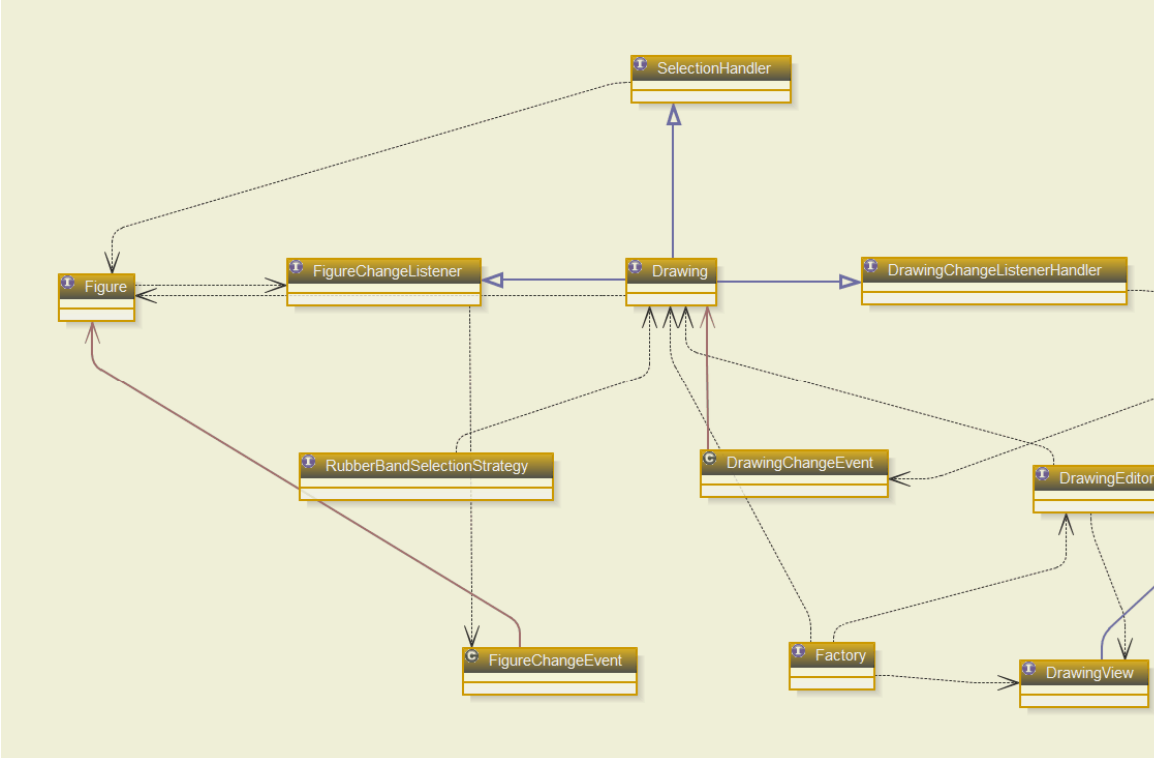


Figure E.6: Class diagram of classes in the framework package.

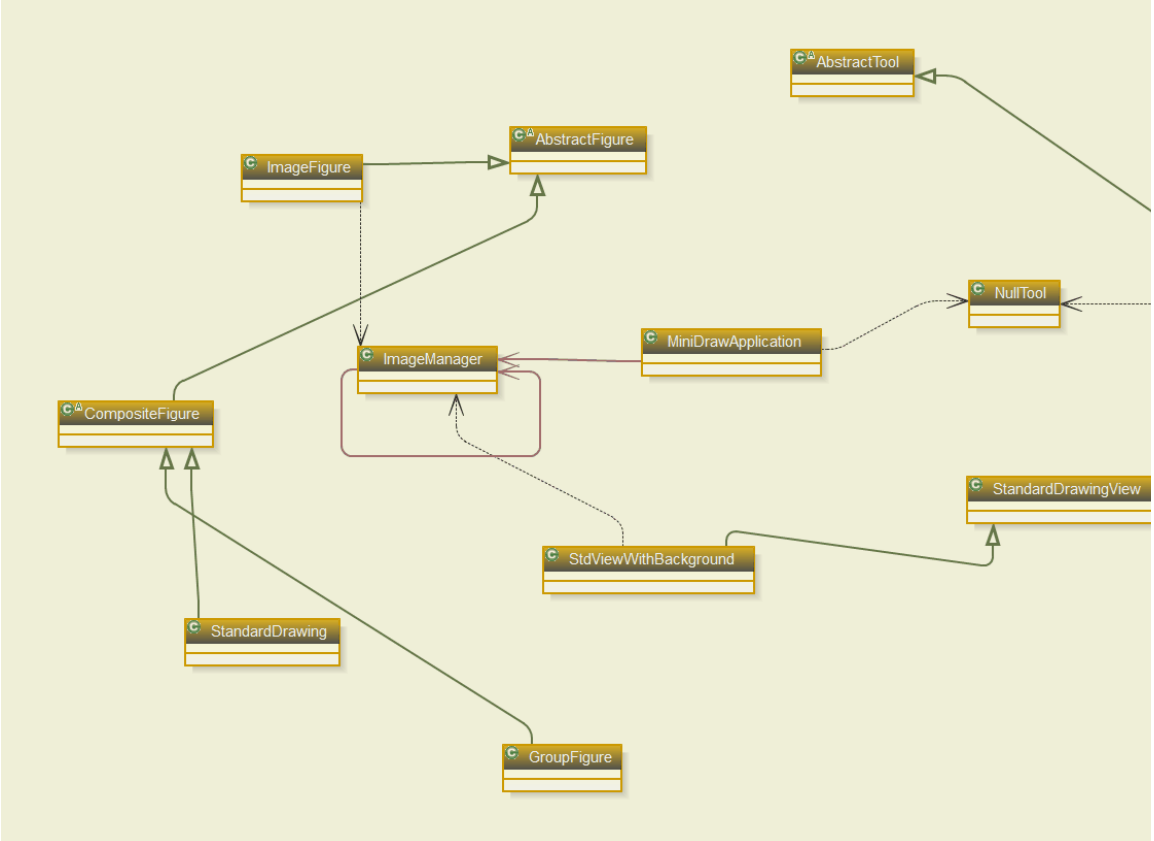


Figure E.7: Class diagram of classes in the standard package.

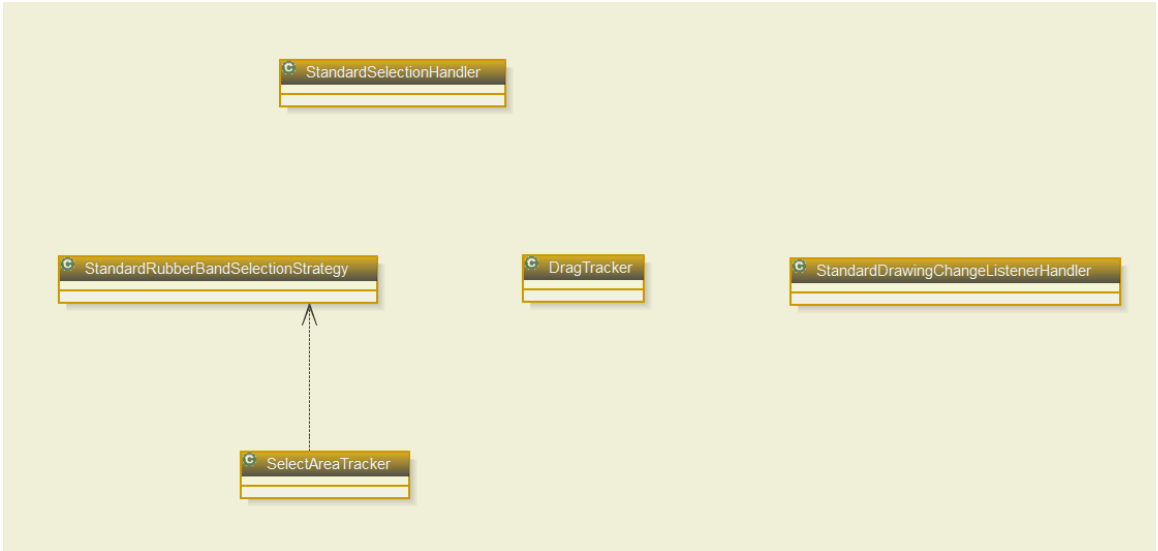


Figure E.8: Class diagram of classes in the handlers package.

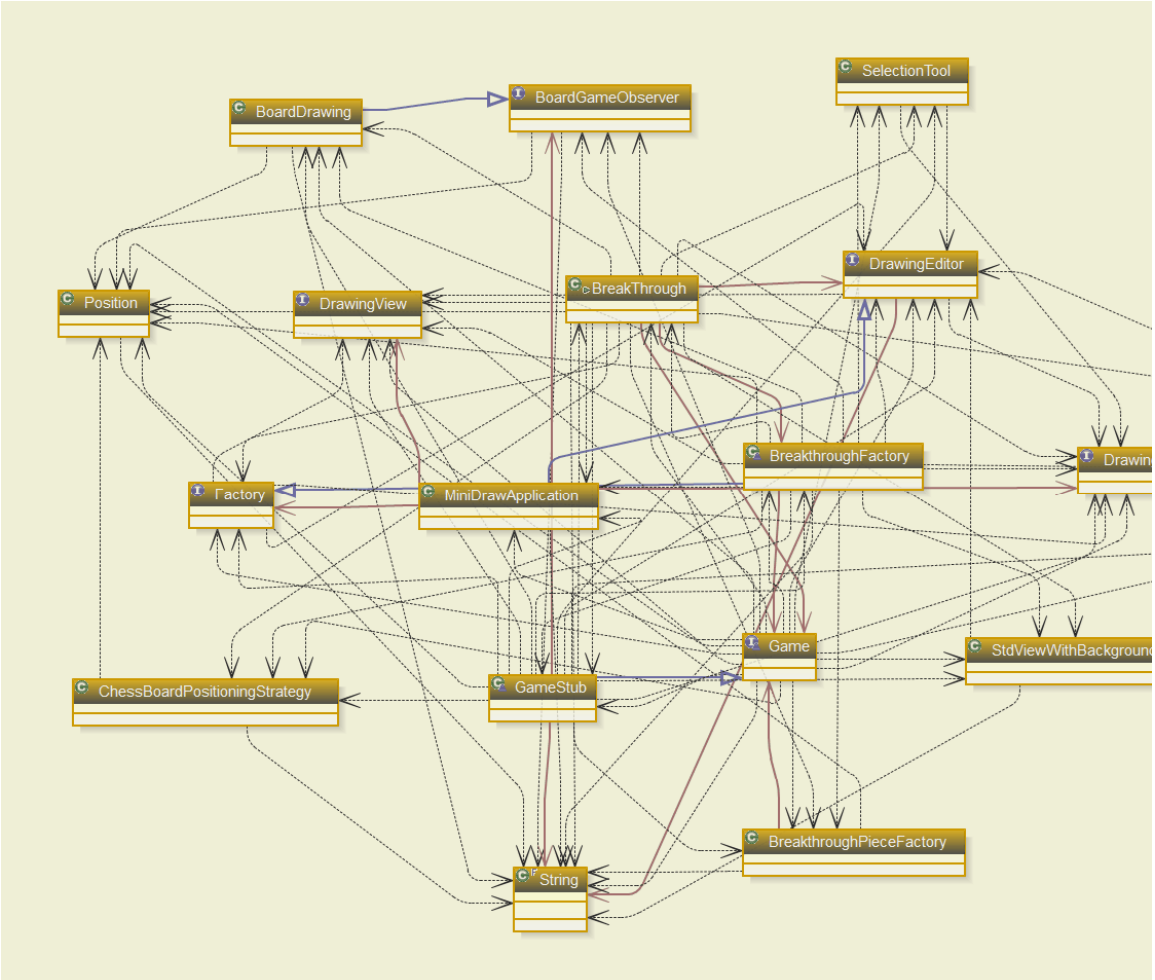


Figure E.9: Class diagram of class dependencies on the BreakThrough class.

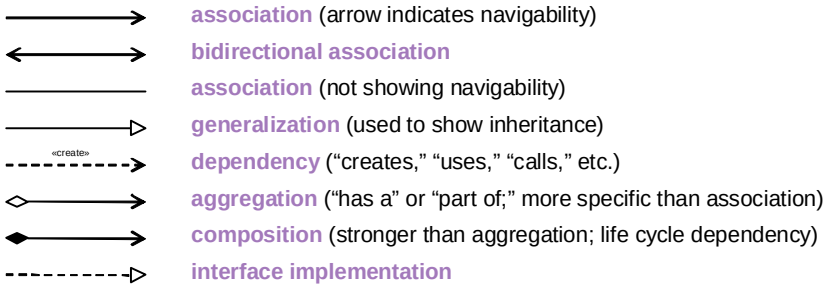


Figure E.10: Basic UML notation.

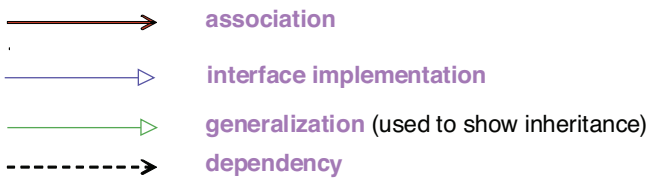


Figure E.11: AgileJ notation.

Appendix F: OOG Viewer Navigation Tutorial

In this brief tutorial, you will try out several of the most useful features in the OOG Viewer.

F.1 Navigating the Graph View

The displayed graph reflects the object structure of MiniDraw, and gives you an idea on how the objects interact at run-time. The three dashed boarders on the graph represent three public domains `MODEL`, `VIEW`, and `CONTROLLER`, and illustrate how the core objects in MiniDraw are organized into these domains. Notice that the domain names are qualified by the name of their declaring class (e.g. `BreakThrough::CONTROLLER`). The yellow boxes represent objects and the edges represent points-to relations between these objects. Notice that some objects have labels between parentheses (e.g. `stdViewWithBackground` has the label `(DrawingChangeListener)`). These labels are automatically generated based on the name or type of one of the references in the program that point to this object, so the `(DrawingChangeListener)` on `stdViewWithBackground` means that `StdViewWithBackground` type is a `DrawingChangeListener`. Ignore the (+) symbol for this part.

Based on your understanding, answer the following questions using Figure E.1 in the Appendix:

1. What do you think is the top-level structure/architecture of the system?
2. What is the root object? What is the declaring class of this object?
3. Try to navigate from any object on the graph using the "trace to code" feature. To which class does the traceability take you? What does this tell you?
4. How do the high-level objects of the system interact?

5. Can you annotate the Breakthrough application implemented in MiniDraw(see Fig 4.1) to map the graphical components on the user interface to their corresponding objects on the OOG? (Use the `objectname:ObjectType` notation).

6. Pick one object on the diagram that has a decorating label and explain how this information is useful to you.

7. Notice that some objects like `boardDrawing` has several decorating labels, and it is hard to tell how these types are related to the type `BoardDrawing`. In this case you can use the "inspect types" feature to investigate more the type hierarchy. Does this provide you with useful information?

8. Pick one object on the diagram and explain how you can get to this object. For example, we can get the `tool` object from the `window` object.

F.2 The Hierarchical nature of the OOG

An OOG provides abstraction by ownership hierarchy when it shows architecturally significant objects near the top of the hierarchy and data structures further down. Moreover, an OOG can provide abstraction by types by collapsing objects further according to their declared types.

In an OOG, an object can have two types of domains to contain it substructure. A public domain contains objects that are still accessible to the outside. A private domain contains objects that are strictly encapsulated within the parent object and cannot be accessed by outside objects (see Fig 2.1). Graphically, our visualization distinguishes between private and public domains, by showing a private domain with a thick dashed border, and a public domain with a thin dashed border. A (+) symbol indicates that an object has a collapsed sub-structure.

F.3 Navigating the Tree View

1. Expand the substructure of `drawing`. Can you explain the difference between domains? How is this useful?

2. As you can see, `widow` has reference to `boardDrawing` and `stdViewWithBackground`. Do you think it has access to their internal objects as well?

3. Expand the `window` object to see its substructure. As you can see `fImageManager` is inside a private domain called `owned`. Why do you think this is the case? (Hint. Use the trace to code feature.)

4. The tree view can be used to search for the nodes on the OOG by typing regular expressions which will do the filtration for you. For example, to search for all the possible edges between `boardFigure` and `command` in the tree you can type the expression (*–>*).

F.4 Exploring Objects

The other form of architectural abstraction is by types, so if two objects share a common supertype, they are collapsed into one object. As a result, each run-time object has exactly one representative on the graph. For example, the `tool` object represents multiple types of objects all ultimately of type `Tool` (`selectionTool`, `boardActionTool`, and `nullTool` in figure E.3). To investigate this, use SHIFT+double click near the border of the `tool` object to select it. This selects the object in the tree viewer. Right click on this object and choose trace to code. Notice that you have different classes to navigate to.

How many objects does the position object represent? (Hint: use the trace to code feature 4.2)

F.5 Exploring Edges

The edges on an OOG, correspond to all possible run-time points-to relations between the objects in MiniDraw. For example, the `command` object points to the `game` object, and the `game` object points to the `p` object. To explore the edge between `game` and `command`, use SHIFT+double click on the arrowhead. This causes the selection in the tree viewer where you can investigate the relation further. Right click on this edge and choose trace to code to navigate to the corresponding field declaration in Eclipse text editor. Deselect all current selections by using SHIFT + left click outside the boarders of the graph. Now explore the other relation between `game` and `p`.

Appendix G: Code Modifications

G.1 Task1

Table G.1: Code modifications performed by C participants in Task 1

P	Phase 1: Planning	Phase 2: Implementation		
	Activity	**Classes modified**	**Methods modified**	**Classes/Methods/Fields created**
C1	`//Task 1a: TODO: get list from factory`	BoardDrawing MoveCommand	 execute()	`Map<Position, List<BoardFigure>> list FigureFactory figureFactory`
C2	`//Task 1a: TODO: get figuremap then check for piece in position to`			`pieceMovedEvent2`
C3	`//Task 1a: TODO: compare cx and cy //with rx and ry`	BoardDrawing	adjustFigurePosition()	putBreakThrough()
	`//Task 1b : TODO: use window object and call function showStatus`	BreakThrough		
C4		BoardFigure BoarDrawing BreakThrough MoveCommand	changeImage() init() execute()	 getDrawing() setDrawing() isValid() isSameOwner()
C5				

Table G.2: Code modifications performed E participants in Task 1.

P	Phase 1: Planning	Phase 2: Implementation		
	Activity	**Classes modified**	**Methods modified**	**Classes/Methods/Fields created**
E1		BoardDrawing GameStub	pieceMovedEvent() move()	
E2		GameStub	move()	
E3	`//Task 1a: TODO: get BoardDrawing object` `//ask for figures at from and to locations`	BoardDrawing GameStub	pieceMovedEvent move()	 getPieceAtPosition() removePieceFromBoard() addwindow()
E4				
E5				

G.2 Task2

Table G.3: Code modifications for Task 2.

Participant	Classes modified	Methods modified	Code added
C1	BoardDrawing MoveCommand	 execute()	setFigureMap() removPiece() isCaptured()
C2	BoardFigure BoardDrawing	BoarFigure()	setIsBlack getIsBlack() piecemovedevent2()
C3	BoardFigure BoardDrawing	 adjustFigurePosition()	getColor()
C4	MoveCommand	isValid()	
C5			
E1	BoardFigure BoardDrawing GameStub	 pieceMovedEvent() move()	
E2	GameStub	move()	
E3	GameStub	move()	oppositePlayers()
E4			
E5			

G.3 Task 3

Table G.4: Code modifications for Task 3.

Participant	Classes modified	Methods modified	Code added
C1	BoardDrawing BreakThrough		setFigureMap() setHistory() getHistory()
C2	BoardDrawing	undoLast()	
C3			
C4			
C5			
E1	MiniDrawApplication		createMenuBar()
E2	GameStub MiniDrawApplication	move()	undo()
E3	MiniDrawApplication		
E4			
E5			

Appendix H: Excerpts From Transcripts

CONTROL						
P,T.A	**Activity**	**Explorations**	**Time**	**Paths**	**Successful**	**Outcome**
P1.T1.a	Locate where to implement the validation logic	11	10			Success
b	get list of all occupied spaces	10	12			Success
c	determine how to access the list	9	19			Success
d	which object knows about the statusbar	NA	NA	NA	NA	NA
e	get hold of the window object	NA	NA	NA	NA	NA
	Total	**10**	**45**			**Not tested**
P2.T1	P:locate where to implement the validation logic	11	6			success
	P:figure out who is keeping track of all pieces	4	2			success
	I:figure out how to access the list inside move command	3	4			success
	P:determine who knows about the message	2	2			success
	I: get hold of the window object	11	15			Fail
	Total	**41**	**43**			**Not tested**
P3.T1	P:locate where to implement the validation logic	16	11			Success
	verify that boarddrawing is the chess board	2	2			Success
	get drawing inside something else	1	1			Success
	P:locate where to show the message	2	4			Success
	I:get hold of the window object in drawing	5	4			Success
	Total	**25**	**21**			**Tested**

Figure H.1: Individual data of the C participants.

EXPERIMENTAL						
P,T	**Activity**	**Explorations**	**Time**	**Paths**	**Successful**	
P4.T1	locate where to implement the validation logic	3	6			
	locate the object representing the board	18	18			
	get drawing somewhere else	1	1			
	who knows about status bar	2	2			
	get window in piecemovedevent	NA	NA			
P4.T2	Locate where to implement the capture	1	1			
	which object represents a piece to compare it to an opponent	11	22			
	Get hold of that object inside the class responsible for handling captures	1	1			
	Figure out how to remove a piece from the game board	1	6			
P4.T3	locate where to put toolbar	2	17			
	where to implement the logic	NA	NA			
	acces undo logic inside ui	NA	NA			
P5.T1	locate where to implement the validation logic	3	10			
	i want to know what the positions of all the pieces	1	4	1	1	
	get hold of that object inside game	2	5	2	1	
	locate where to show the status message	5	6			
	get the window object	NA	NA			
P5.T2	Locate where to implement the capture	15	14			
	which object represents a piece to compare it to an opponent	2	2			
	Get hold of that object inside the class responsible for handling captures	2	4			
	Figure out how to remove a piece from the game board	1	2			
P5.T3	put toolbar	5	7			
	where to put undo logic	1	1			
	how to access that logic in the interface	1	1			
P6.T1	P:locate where to implement the validation logic	3	2			
	P:look for the container that holds all the pieces on the board	3	4			
	I:get drawing object inside gamestub	3	6	1	1	
	I:who knows about the status bar	2	1	1	1	
	how to get message sent to the window	4	11	2		
P6.T2	Locate where to implement the capture	1	1			

Figure H.2: Individual data of the E participants.

REFERENCES

[1] Abi-Antoun, M. *Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure.* PhD thesis, Carnegie Mellon University, 2010. Available as Technical Report CMU-ISR-10-114.

[2] Abi-Antoun, M., and Aldrich, J. A Field Study in Static Extraction of Runtime Architectures. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)* (2008).

[3] Abi-Antoun, M., and Aldrich, J. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2009).

[4] Abi-Antoun, M., and Ammar, N. A Case Study in Evaluating the Usefulness of the Run-time Structure during Coding Tasks. In *Workshop on Human Aspects of Software Engineering (HAoSE)* (2010).

[5] Abi-Antoun, M., Ammar, N., and LaToza, T. Questions about Object Structure during Coding Activities. In *Cooperative and Human Aspects of Software Engineering (CHASE)* (2010).

[6] Abi-Antoun, M., and Selitsky, T. Interactive Refinement of Runtime Structure. In *Workshop on Flexible Modeling Tools (FlexiTools)* (2010).

[7] Abi-Antoun, M., Selitsky, T., and LaToza, T. Developer Refinement of Runtime Architectural Structure. In *SHAring and Reusing architectural Knowledge (SHARK)* (2010).

[8] AgileJ. StructureViews. `www.agilej.com`, 2008.

[9] Aldrich, J., and Chambers, C. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object-Oriented Programming (ECOOP)* (2004).

[10] Bennett, C. J., Myers, D., Storey, M.-A., German, D. M., Ouellet, D., Salois, M., and Charland, P. A Survey and Evaluation of Tool Features for Understanding Reverse-Engineered Sequence Diagrams. *J. Softw. Maint. Evol. 20*, 4 (2008).

[11] Bennett, K. H., Rajlich, V., and Wilde, N. Software evolution and the staged model of the software lifecycle. *Advances in Computers 56* (2002), 3–55.

[12] Calcagno, C., Distefano, D., O'Hearn, P. W., and Yang, H. Compositional shape analysis by means of bi-abduction. In *Principles of Programming Languages (POPL)* (2009).

[13] Christensen, H. B. *Flexible, Reliable Software Using Patterns and Agile Development*. Chapman and Hall/CRC, 2010.

[14] de Alwis, B., Murphy, G., and Robillard, M. A comparative study of three program exploration tools. In *International Conference on Program Comprehension (ICPC)* (2007).

[15] Dekel, U., and Herbsleb, J. D. Notation and representation in collaborative object-oriented design: an observational study. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2007).

[16] Demsky, B., and Rinard, M. Role-based exploration of object-oriented programs. In *International Conference on Software Engineering (ICSE)* (2002).

[17] DrawLets. `www.rolemodelsoft.com/drawlets/`, 2002. Version 2.0.

[18] Dzidek, W., Arisholm, E., and Briand, L. A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. *IEEE Transactions on Software Engineering 34*, 3 (2008).

[19] Ellis, B., Stylos, J., and Myers, B. The factory pattern in api design: A usability evaluation. In *International Conference on Software Engineering (ICSE)* (2007).

[20] Eclipse Metrics Plugin. `http://metrics.sourceforge.net/`, 2010.

[21] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[22] Hadar, I., and Hazzan, O. On the Contribution of UML Diagrams to Software System Comprehension. *Journal of Object Technology 3*, 1 (2004).

[23] Hailat, Z., and Abi-Antoun, M. Adding Ownership Domain Annotations to and Extracting Ownership Object Graphs from MiniDraw. Tech. rep., WSU, 2011.

[24] Hill, E., Pollock, L., and Vijay-Shanker, K. Exploring the neighborhood with dora to expedite software maintenance. In *Automated Software Engineering* (2007).

[25] Hill, T., Noble, J., and Potter, J. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *Journal of Visual Languages and Computing 13*, 3 (2002).

[26] Jackson, D., and Waingold, A. Lightweight Extraction of Object Models from Bytecode. *IEEE Transactions on Software Engineering 27*, 2 (2001).

[27] JHotDraw. `www.jhotdraw.org`, 1996. Version 5.3.

[28] Kirk, D., Roper, M., and Wood, M. Identifying and Addressing Problems in Object-Oriented Framework Reuse. *Empirical Software Engineering 12*, 3 (2006).

[29] Ko, A. J., DeLine, R., and Venolia, G. Information Needs in Collocated Software Development Teams. In *International Conference on Software Engineering (ICSE)* (2007).

[30] Kollman, R., Selonen, P., Stroulia, E., Systä, T., and Zundorf, A. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In *Working Conference on Reverse Engineering (WCRE)* (2002).

[31] Lam, P., and Rinard, M. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *European Conference on Object-Oriented Programming (ECOOP)* (2003).

[32] Lange, D., and Nakamura, Y. Interactive Visualization of Design Patterns Can Help in Framework Understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (1995).

[33] LaToza, T. D., and Myers, B. A. Developers ask reachability questions. In *International Conference on Software Engineering (ICSE)* (2010).

[34] LaToza, T. D., and Myers, B. A. Visualizing Call Graphs. In *Visual Languages and Human-Centric Computing (VL/HCC)* (2011). To appear.

[35] Lee, S., Murphy, G., Fritz, T., and Allen, M. How Can Diagramming Tools Help Support Programming Activities? In *Visual Languages and Human-Centric Computing (VL/HCC)* (2008).

[36] MiniDraw. `www.baerbak.com/`.

[37] Mitchell, N., Schonberg, E., and Sevitsky, G. Making Sense of Large Heaps. In *European Conference on Object-Oriented Programming (ECOOP)* (2009).

[38] O'Callahan, R. W. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie Mellon University, 2001.

[39] Potanin, A., Noble, J., and Biddle, R. Checking Ownership and Confinement. *Concurrency and Computation: Practice and Experience 16*, 7 (April 2004).

[40] Quante, J. Do dynamic object process graphs support program understanding? - a controlled experiment. In *International Conference on Program Comprehension (ICPC)* (2008).

[41] Rawshdeh, S. A static analysis to extract dataflow edges from object-oriented programs with ownership domain annotations. Master's thesis, Wayne State University, 2011.

[42] Rawshdeh, S., and Abi-Antoun, M. A static analysis to extract dataflow edges from object-oriented programs with ownership domain annotations. *In preparation*, 2011.

[43] Richner, T., and Ducasse, S. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. In *International Conference on Software Maintenance (ICSM)* (1999).

[44] Riehle, D. *Framework Design: a Role Modeling Approach*. PhD thesis, Federal Institute of Technology Zurich, 2000.

[45] Rosenthal, R., and Rosnow, R. L. *Essentials of behavioral research: methods and data analysis*. McGraw Hill, 1984.

[46] Selitsky, T. A Front-End for an Ownership Object Graph Interactive Editor. Master's thesis, Wayne State University, 2010.

[47] Shull, F., Lanubile, F., and Basili, V. R. Investigating Reading Techniques for Object-Oriented Framework Learning. *IEEE Transactions on Software Engineering 26*, 11 (2000).

[48] Sillito, J., Murphy, G., and Volder, K. D. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering 34*, 4 (2008).

[49] Spiegel, A. *Automatic Distribution of Object-Oriented Programs.* PhD thesis, FU Berlin, 2002.

[50] Tonella, P., and Potrich, A. *Reverse Engineering of Object Oriented Code (Monographs in Computer Science).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.

[51] Walker, R. J., Murphy, G., Freeman-Benson, B., Wright, D., Swanson, D., and Isaak, J. Visualizing Dynamic Software System Information through High-Level Models. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (1998).

ABSTRACT

EVALUATION OF THE USEFULNESS OF DIAGRAMS OF THE RUN-TIME STRUCTURE FOR CODING ACTIVITIES

by

NARIMAN AMMAR

August 2011

Advisor: Dr. Marwan Abi-Antoun

Major: Computer Science

Degree: Master of Science

Software maintenance accounts for 50% to 90% of the costs over the life-cycle of a software system. And program comprehension is a major activity during software maintenance, absorbing around half of the maintenance costs. This is especially true when developers attempt to perform maintenance tasks such as bug fixes and feature requests on a well-designed framework. Therefore, during code evolution tasks, developers often consult high-level views of the system, i.e., design documents and diagrams, to help them understand the design intent before they attempt any code modification. Different diagrams are often used to describe the design intent in a software system, including diagrams of the code structure, e.g. class diagrams, and diagrams of the run-time structure, i.e. object diagrams. A class diagram describes class dependencies at compile time, whereas an object diagram describes interactions between different instances of a class at run-time. An exploratory study showed that 80% of the developers questions while doing coding activities required thinking about object relations rather than class dependencies. Thus, class diagrams are not enough for understanding the structure of the system and need to be combined with object diagrams to understand object interactions.

There is no shortage of tools that extract class diagrams. Also, previous research studied the benefits of class diagrams. Giving developers usable object diagrams, however, had typically been challenging. Recently, Abi-Antoun and Aldrich had proposed an approach to extract hierarchical ownership object graphs (OOG) for the entire system. An OOG can help developers address a typical software comprehension task at a variety of abstraction levels, by combining information about object communication at both the architectural level and implementation details hidden within interfaces such as data structures, an inherent feature in the object-oriented paradigm. But the usefulness of those diagrams had not been evaluated empirically. So we set out to investigate the following research question: do developers who have access to diagrams of the run-time structure, i.e. OOGs, in addition to diagrams of the code structure, i.e. class diagrams, perform code modification tasks more effectively than developers who have access to only diagrams of the code structure?

This thesis provides the first empirical evidence to evaluate the usefulness of OOGs. In a controlled experiment, we observed 10 developers perform maintenance activities on a well-designed object-oriented framework. We found that when developers attempt code maintenance tasks, they struggle mostly with activities related seeking answers for questions about the object structure. Two of the developers who had access to OOGs completed the three tasks compared to only one developer who had access to only class diagrams. On average, developers who had access to OOGs performed their activities in less time (22%–60%), and by browsing less irrelevant code (10%–60%) than developers who had access to only class diagrams or who did not use diagram at all.

AUTOBIOGRAPHICAL STATEMENT

NARIMAN AMMAR

EDUCATION

- Master of Science (Computer Science), August 2011
  Wayne State University, Detroit, MI, USA
- Bachelor of Engineering (Computer Systems Engineering), June 2006
  BirZeit University, Palestine

PUBLICATIONS

1. Abi-Antoun, M., and Ammar, N. A Case Study in Evaluating the Usefulness of the Run-time Structure during Coding Tasks. In *Workshop on Human Aspects of Software Engineering (HAoSE)* (2010).
2. Abi-Antoun, M., Ammar, N., and LaToza, T. Questions about Object Structure during Coding Activities. In *Cooperative and Human Aspects of Software Engineering (CHASE)* (2010).