Evaluation of Global Hierarchical Object Graphs for Coding Activities: a Controlled Experiment[1]

Nariman Ammar **Marwan Abi-Antoun**

December 2011

Department of Computer Science
Wayne State University
Detroit, MI 48202

Abstract

A diagram of the runtime structure shows objects and their relations and complements diagrams of the code structure. One such diagram, the Ownership Object Graph (OOG), is a global object graph that conveys architectural hierarchy based on annotations in the code. In this work, we ask the research question: do developers benefit from using OOGs, in addition to class diagrams, during their coding activities?
We conducted the first controlled experiment evaluating global object graphs in relation to class diagrams. We observed 10 developers, for 3 hours each, perform coding activities on a framework application. Developers struggled with many questions about the object structure. Two developers who used OOGs completed the three tasks compared to only one developer who used only class diagrams. Developers who used OOGs performed their activities in less time by 22–60%, and browsed less irrelevant code by 10–60%, compared to those who used only class diagrams.

[1]This technical report is based on: Ammar, N. *Evaluation of the Usefulness of Diagrams of the Run-Time Structure for Coding Activities*, M.S. thesis, Wayne State University, Department of Computer Science, August 2011. Available online: `http://digitalcommons.wayne.edu/oa_theses/91/`

Keywords: class diagrams, object diagrams, hierarchical object graphs, controlled experiment

1 Introduction

Software maintenance accounts for 50% to 90% of the costs over the life-cycle of a software system. One major activity during maintenance, program comprehension, absorbs around half of the costs [9]. To support comprehension, researchers have produced many tools to visualize the structure of the system. These tools are based on the widespread belief that diagrams are useful for program comprehension. For example, a high-level diagram can help a developer locate where to implement a change.

One problem related to diagrams and comprehension is understanding the run-time structure of object-oriented code, in terms of objects and their relations. With object-oriented design, it is at least as important—possibly more important—to understand the run-time structure as to understand the code structure dealing with source files, classes and packages. Moreover, for object-oriented code, the run-time structure is often quite different from the code structure. Thus, diagrams of the run-time structure can be highly complementary to diagrams of the code structure, and can answer several crucial questions that developers ask during code modifications.

Recently, Abi-Antoun and Aldrich [2] proposed a solution to statically extract a global, hierarchical Ownership Object Graph (OOG), that shows all the objects across the entire system, organized hierarchically. They also demonstrated the usefulness of an OOG by abstracting it into an as-built, run-time architecture, then analyzing the conformance of the implementation to an as-designed, run-time architecture. In this work, we de-emphasize conformance and focus instead on program comprehension. This paper's contribution is to evaluate in a controlled experiment if, as diagrams of the run-time structure, OOGs are useful to developers who are performing code modification tasks that require information about the run-time architectural structure. Our research hypothesis is:

> ***Research Hypothesis:*** *developers who use global, hierarchical, Ownership Object Graphs (OOGs), that depict objects and relations between objects, perform code modification tasks more effectively than developers who use only class diagrams or who just explore the code. The former complete more coding tasks, take less time, and explore less irrelevant code, compared to the latter.*

Outline. The rest of this paper is organized as follows. We first illustrate key differences between OOGs and other diagrams (Section 2). Next, we present our method (Section 4), data analysis (Section 5) and results (Section 6). We then mention some limitations and threats to validity (Section 7), discuss related work (Section 8) and conclude.

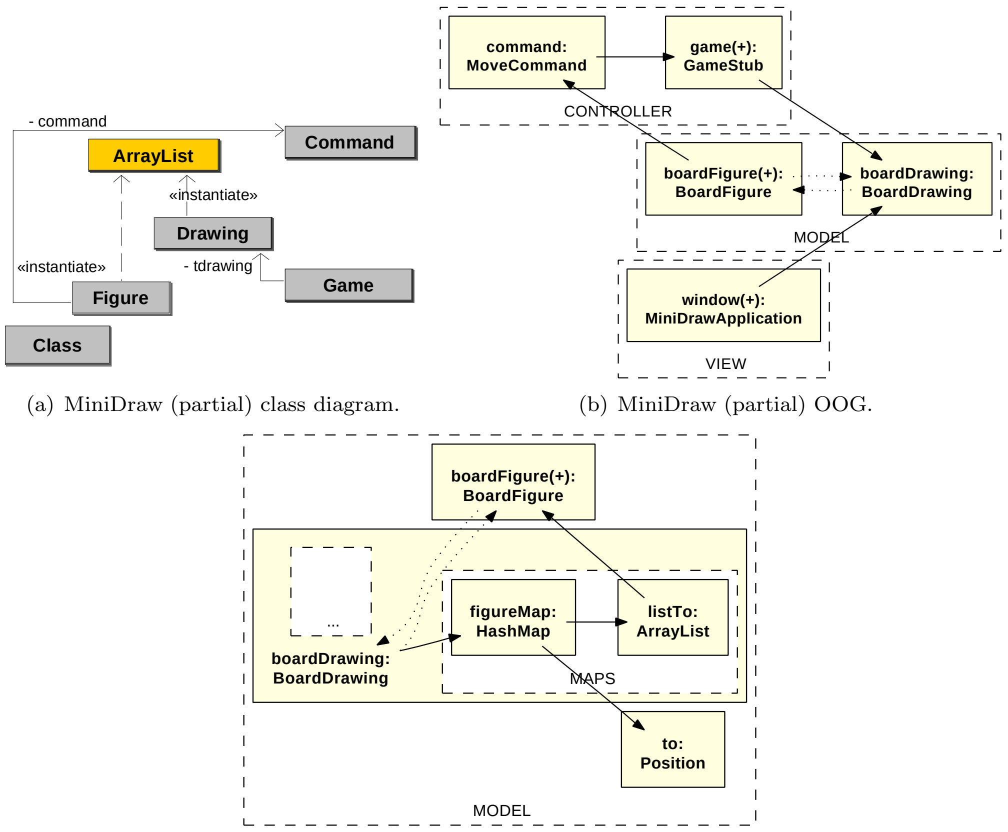
2 Background

We illustrate by example some key differences between class diagrams, object diagrams, flat object graphs and OOGs. Our example uses some core classes of MiniDraw, the subject system in our experiment (Section 4).

Class diagram. The object-oriented community, both in research and practice, uses *class diagrams* depicting the code structure of a system, in terms of classes and inheritance relationships. Today, many tools *automatically* extract class diagrams or other module views showing layers (packages) of the code structure. For example, a class diagram for MiniDraw (Fig. 1(a)) shows the classes `Drawing` and `Figure`, which depend on `ArrayList`.

Object diagram. Another important design diagram 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 diagram makes explicit the structure of the objects 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 [32]. Today, there are few tools that extract meaningful object diagrams, so most object diagrams are *manually* generated.

Manually drawn object diagrams illustrate how a few object participate in a design pattern. But they lack abstraction, and do not convey how conceptual groups of objects interact across the entire system. For example, they often show multiple instances of the same class, and how each instance has different field



(a) MiniDraw (partial) class diagram.

(b) MiniDraw (partial) OOG.

(c) Expanded `boardDrawing` sub-structure.

Figure 1: MiniDraw: partial class diagram vs. partial OOG.

values. A flat object diagram works when looking at a small number of objects, but does not scale when including all the objects in the system.

Flat object graph. To approximate object diagrams, several automated approaches statically extract flat object graphs [17]. A flat object graph mixes low-level objects that are data structures such as objects of type `ArrayList` or `HashMap` with architecturally-relevant objects from the application domain such as objects of type `BoardDrawing`, and a developer has no easy way to distinguish between these objects. Most extracted object graphs are flat because architectural hierarchy is often missing in code that is written in general-purpose programming languages.

Ownership Object Graphs (OOG). To shrink a large, flat graph, hierarchy is effective [30]. One idea is to use abstraction by object hierarchy, where architecturally significant objects appear near the top of the hierarchy and low-level objects are further down. For example, a `listTo` object of type `ArrayList` is a child of the `boardDrawing` object of type `BoardDrawing` (see the ownership tree in Fig. 3).

To supply the missing architectural hierarchy, developers use annotations and specify, within the code, their design intent as minimally invasive hints about object encapsulation, logical containment and architectural tiers. Furthermore, the annotations use language support for annotations, and are backward compatible with legacy code. The annotations then enable a static analysis to extract a global, hierarchical object graph, the OOG. Furthermore, since the OOG is extracted statically, it is sound and safely approximates all possible objects and their relations that can occur in any program run.

An OOG is abstract in that one "canonical object" on the OOG corresponds to one or more objects in the system at run-time. Further, an OOG groups related objects into *domains*, that are conceptual groups of objects. A domain roughly corresponds to an architectural runtime tier, a notion present in most architecture description languages. Also, an OOG does not pin things down to individual objects. Instead, it merges several objects of the same or similar types that are in the same domain. For example, MiniDraw creates many instances of the `Position` class at runtime. But the MiniDraw OOG (Fig. 1(b)) shows one `to:Position` object in the `MODEL` domain. Still, one can trace from the OOG to multiple corresponding lines of code (See Fig. 3).

For MiniDraw, the top-level domains, `MODEL`, `VIEW` and `CONTROLLER`, indicate that MiniDraw follows a three-tiered architecture. A user interface component such as `window` is in the `VIEW` domain. A `boardDrawing` object and related objects such as `boardFigure` are in the `MODEL` domain. In addition, the OOG is hierarchical: an object can have one or more nested domains, with other objects inside them. Hierarchy enables both high-level understanding and detail. To get high-level understanding, we can collapse the sub-structure of `boardDrawing` and hide its sub-structure (Fig. 1(b)). To gain detailed understanding, we can expand `boardDrawing` to reveal a nested domain, `MAPS`, which contains objects `figureMap` and `listTo` (Fig. 1(c)). Similarly, we can expand the sub-structure of `boardFigure`, to reveal data structures such as other `ArrayList` instances.

Graphical Notation. Our current visualization uses box nesting to indicate containment of objects inside domains and domains inside objects. 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 the "object `obj`" or a "`T` object" to mean "an instance of the `T` class." A (+) symbol on an object or a domain indicates that it has a collapsed sub-structure. A dotted edge is an edge that is lifted from a hidden child object to the nearest visible ancestor in the object hierarchy.

Are OOGs useful during coding activities? An OOG makes visually obvious several facts about the system's runtime structure. Compared to a class diagram, an OOG makes more explicit the sharing of objects in the system. On the MiniDraw class diagram, the edges might suggest that, at run-time, a `BoardDrawing` object and a `Figure` object share the same `ArrayList` object. But at run-time, different instances of the `ArrayList` class belong to conceptually different objects. Unlike the class diagram which shows one `ArrayList` class, the OOG shows different instances of `ArrayList` in different domains. For example, a `boardDrawing` object has its own `listTo` object of type `ArrayList` inside its `MAPS` domain. Though collapsed, the `boardFigure` object contains a different `ArrayList` object, inside its `owned` domain. Of course, whether outside developers can learn useful facts about the object structure through the OOG during their coding tasks requires a controlled experiment, this paper's contribution.

3 Extracting OOGs

We briefly discuss the process to extract OOGs for a system. It includes: adding annotations, extracting initial OOGs, and refining the extracted OOGs.

Add annotations to the code: We first add 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. We then use a typechecker to check the annotations, and manually fix any annotation warnings. The warnings produced by the type checker are alerts that the OOG may be unsound, so we tried to resolve most of these warnings before extracting the OOG.

Our concrete system uses available language support for annotations, which leads to verbose code constructs, but enables adding annotations to legacy code, after the fact, in order to reverse-engineer OOGs. The annotations still implement the Ownership Domains type system [7], and a typechecker checks that the annotations are consistent with each other and with the code. We describe the annotation language elsewhere [1, Appendix A].

To give the reader some intuition about the information that the annotations encode, we show some code with annotations, adapted from our subject system (Fig. 2). The example shows how architectural tiers, such as `MODEL`, `VIEW`, `CONTROLLER`, can be represented as top-level domains (line 1). Architectural hierarchy

```
 1 @Domains({ "VIEW", "CONTROLLER", "MODEL" }) // Declare top-level domains
 2 class BreakThrough { // Root class used as starting point for extracting OOG
 3   @Domain("CONTROLLER<VIEW,CONTROLLER,MODEL>") GameStub gameStub = new ...;
 4   @Domain("VIEW<VIEW,CONTROLLER,MODEL>") DrawingEditor window = new ...;
 5   @Domain("MODEL<VIEW,CONTROLLER,MODEL>") BoardDrawing drawing = new ...;
 6 }
 7 @Domains({ "owned", "MAPS" }) // Declare private domain “owned”; public domain “MAPS”
 8 @DomainParams({ "U","L","D" }) // Declare domain parameters
 9 @DomainInherits({"StandardDrawing<U,L,D>"}) // Handle inheritance
10 class BoardDrawing extends StandardDrawing ... {
11   // Map each location to the set of images positioned on it
12   @Domain("MAPS<D,MAPS<D>>") Map<Position, List<BoardFigure>> figureMap = new ...;
13
14   // Map graphical (x,y) positions to the props of the game
15   @Domain("owned<shared, D<U,L,D>>") Map<String, BoardFigure> propMap = new ...;
16 }
```

Figure 2: The root class used in MiniDraw to extract OOG.

is encoded using either logical containment or strict encapsulation. Logical containment is illustrated using a public domain such as `MAPS`, declared on the class `BoardDrawing` (line 7), to make the `figureMap` object conceptually *part of* a `boardDrawingObject` (line 12). Strict encapsulation is illustrated using a private domain, such as `owned`, declared on the class `BoardDrawing` (line 7), to make the `propMap` object *owned by* a `boardDrawingObject` (line 15).

Extract initial OOGs: once the annotations are in the code and type check correctly, we run a static analysis to extract some initial OOGs. The goal is to reduce the number of objects in the top-level domains in the OOGs. We tweak the annotations to try to obtain a less cluttered OOG by imposing conceptual hierarchy on all the objects in the system, such that architecturally significant objects appear near the top of the hierarchy and data structures are further down.

Refine the extracted OOGs: in this step, we usually work with the original developers of the system to make the OOG more relevant, and make it convey their design intent. In this study, however, we did not have access to the original MiniDraw designers, so the experimenter performed a few representative code modification tasks and gave us feedback on the OOG for further refinement.

4 Method

In this section, we describe the measures, experimental design, group assignment, and treatment.

Hypotheses and Measures. Based on our research question, we formulate the following null hypotheses:

H10 *Using OOGs in addition to class diagrams has no impact on developers ability to answer questions about the object structure.*

H20 *Using OOGs does not impact the number of code elements explored by developers.*

H30 *Using OOGs does not impact the time spent by developers on their tasks.*

H40 *Using OOGs does not impact the success rate of developers.*

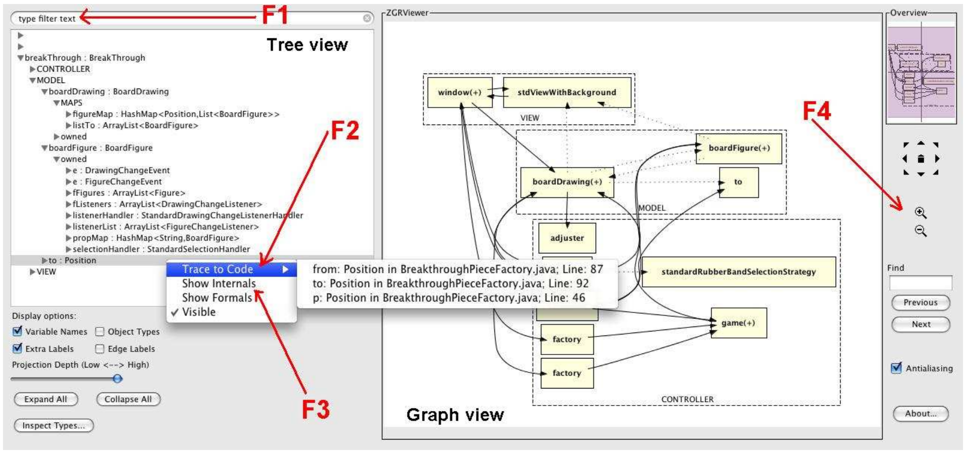
Thus, our corresponding alternative hypotheses are the following:

H1 *Developers who use OOGs answer questions about the object structure that they cannot answer using class diagrams.*

H2 *Developers who use OOGs explore fewer code elements.*

H3 *Developers who use OOGs complete their tasks faster.*

H4 *Developers who use OOGs succeed on their tasks.*



```
▼breakThrough : BreakThrough
  ▶CONTROLLER
  ▼MODEL
    ▼boardDrawing : BoardDrawing
      ▼MAPS
        ▶figureMap : HashMap<Position,List<BoardFigure>>
        ▶listTo : ArrayList<BoardFigure>
      ▶owned
    ▼boardFigure : BoardFigure
      ▼owned
        ▶e : DrawingChangeEvent
        ▶e : FigureChangeEvent
        ▶fFigures : ArrayList<Figure>
        ▶fListeners : ArrayList<DrawingChangeListener>
        ▶listenerHandler : StandardDrawingChangeListenerHandler
        ▶listenerList : ArrayList<FigureChangeListener>
        ▶propMap : HashMap<String,BoardFigure>
        ▶selectionHandler : StandardSelectionHandler
    ▶to : Position
  ▶VIEW
```

```
from: Position in BreakthroughPieceFactory.java; Line: 87
to: Position in BreakthroughPieceFactory.java; Line: 92
p: Position in BreakthroughPieceFactory.java; Line: 46
```

Figure 3: The OOG Viewer to interactively navigate an OOG. The tree view shows the ownership tree. The graph view shows the OOG using nested boxes. Developers can interactively navigate the OOG by searching for an object in the ownership hierarchy (F1), tracing from a selected object or edge to the code in Eclipse (F2), collapsing or expanding the sub-structure of an object (F3), and using other navigation features such as zooming, panning and scrolling (F4).

Experimental Design. We followed the between-subjects design. In the *control* group, C participants received class diagrams. In the *experimental* group, E participants additionally received OOGs. All the participants received an instruction sheet and a brief tutorial on MiniDraw which explained 4 class diagrams manually drawn by the MiniDraw designers [10]. They also received 6 class diagrams reverse-engineered using AgileJ [6]. The E participants received two printed OOGs In addition, the E participants received an OOG Viewer, to interact with the OOGs (Fig. 3). Since the OOG is a hierarchical representation, the OOG viewer allows expanding or collapsing an object's sub-structure, something that cannot be done on a printout.

The *independent variable* in our experiment was having access to OOGs. To measure the effect of the controlled variation of the independent variable, we used the following *dependent variables*: 1. the number of code elements explored to complete a task; 2. the time to task completion; and 3. the success on a task.

We used Camtasia to record the think-aloud of the participants as well as a screen capture of their navigations in Eclipse and whatever diagrams they used. The study materials are available [8, Appendix].

Subject System. We used MiniDraw [21], a pedagogical object-oriented framework that consists of around 1,400 lines of Java code, 31 classes and 17 interfaces. We added annotations to the MiniDraw code and extracted OOGs. Due to space limits, we discuss the annotation process elsewhere [8, Chap.3]. For the experiment, we used the BreakThrough framework application, which is a two-person game played on an 8x8 chessboard. We discussed the process of extracting and refining the BreakThrough OOG in Section 3.

Task Design. 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. We asked the participants to reuse the framework and implement the following tasks:

T1 *Implement validation on the piece movement.* A piece may move one square straight or diagonally in the case of capture.

T2 *Implement the capture of a piece.* When capturing, the opponent piece is removed from the board and the player's piece takes its position.

T3 *Implement an undo move feature.* Add a menu item "Undo move". In the cases where the move cannot be undone, display a message on the status bar.

Table 1: Overview of the experimental procedure (3-hour session).

Part I (25 min)	**Introduction**	2 min
	Eclipse Tutorial	3 min
	OOG Tutorial	20 min
Part II (2.5 hours)	**Planning phase**	**Implementation phase**
	T1	T1
	Questionnaire	Questionnaire
	T2	T2
	Questionnaire	Questionnaire
	T3	T3
	Questionnaire	Questionnaire
Part III (5 min)	**Exit Interview**	5 min

Table 2: The recurring questionnaire used between the tasks.

No.	**Question**
QX.1	Can you formulate a hypothesis? (Do you have a plan?)
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?

Procedure. Our experiment was in the form of a 3-hour session (Table 1). The experimenter briefly introduced MiniDraw. Then, for 5 minutes, she tutored the participants on the Eclipse navigation features (Table 12), using hands-on exercises on MiniDraw.

Since the concepts of OOGs and ownership are not general knowledge, the experimenter gave the E group a 20-minute tutorial explaining the OOG notation and the main features of the OOG Viewer (Table 11) using hands-on exercises (Table 14). To test whether having access to OOGs would have made a difference, the experimenter gave the C participants the OOG tutorial during the last 20 minutes, where she explained the OOG briefly and asked them if they could have done better on their tasks using the OOG and whether it could have helped them answer some questions with which they struggled.

In the remaining 2.5 hours, the experimenter asked the participants in both groups to read the instruction sheet and perform the tasks. The participants were encouraged to work in two phases: planning and implementation (Table 1, Part II). They were allowed to attempt the tasks the way that worked best for them to avoid the artificial setting. They could either plan for the changes before implementing them, or implement the changes while they planned for them. If they got stuck, the participants were allowed to comment out their changes and move to the next task. In the planning phase, the participants documented their plans in the form of pseudocode. The implementation phase involved mainly coding, testing, debugging, and refactoring. The participants were allowed to test their modifications 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 one of the questions from the questionnaire.

Questionnaires and Interview. The experimenter used a recurring questionnaire between the tasks (Table 2) to measure the level of program comprehension in a participant and how useful they perceived the diagrams to be. At the end, exit interview questions captured the participants' subjective feedback and any other free-form comments they had (Table 15).

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. We selected 10 participants (Table 3): 4 professional programmers, 3 Ph.D. students in their 4th year, 2 M.S. students, and 1 senior undergraduate. The median in programming experience was 8.5 years, while the median in Java experience was 4 years. All participants were familiar with Eclipse and UML, and all except one, with frameworks and design patterns.

Table 3: Participants' self-reported experience on a Likert scale 1(beginner) to 5(expert).

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

Table 4: Coding model for the questions or facts that developers ask or state about the system [4].

Code	**Question/Fact about...**
CD/OOG	the Class Diagram (CD) or the OOG
Is-A	an Is-A relationship (one class extends from another class, or implements an interface)
Has-A	a Has-A relationship
Is-Part-Of	is an object logically *part of* another object (inside a public domain)
Is-Owned-By	is an object strictly *owned by* another object (inside a private domain)
Is-In-Tier	is object in some runtime tier
Points-To	does an object point-to another object
Has-Label	the label of an element on the diagram
How-To-Get-X	how to get a reference to an object `x:X`
May-(Not)-Alias	if two variables `x` and `y` may (or not) refer to the same object at run-time

5 Data Analysis

We transcribed the recordings offline (Fig. 7). According to the transcripts, all participants divided the tasks into smaller activities to be able to solve them. The participants documented these activities as comments in the code. For each task, we studied the sequence of activities in which all the participants engaged (Table 5). The participants did not do the activities in the same order, but for simplicity and to be able to compare results, we list them in the order specified.

In the transcripts (Fig. 7), we studied the code explored and the time spent, among others, following standard protocol analysis [27].

Code explored. A *navigation target* (Fig. 7) is a class, a method qualified by the class name, an element on the OOG, a class diagram, or a feature in the OOG Viewer. For each activity, a *navigation path* followed

Table 5: Activities performed in each task. The code definitions are in Table 4.

Task	**Activity**	**Description**	**Question**
T1	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	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	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

by a participant is a sequence of navigation targets that lead a participant to an *outcome* (Table 8). A path has a length (number of navigation targets including repetition), the time spent following it, and an *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 that leads to a successful outcome. An *unsuccessful* path is one that a participant starts to follow, then abandons or finds distracting.

Time. Our analysis of the time to task completion includes the time a participant spent formulating hypotheses, answering the recurring questionnaire, providing the plans, implementing the change, and testing the modifications.

Success. 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 to test the correctness of their implementation by running the application.

Questionnaires. Our analysis of the participants' response to the questionnaires remained qualitative. While some participants did not answer all of the questions when prompted, they made assumptions which turned out to be either correct or incorrect. Either way, they spent time validating their assumptions. We use quotes from the participants' answers to the questions to support our hypotheses. Where applicable, we report at which point during the session a participant came to an answer.

6 Results

Our results indicate that the participants who used OOGs, in addition to class diagrams, performed better than those who used only class diagrams (Table 7). Two of the participants who used OOGs completed the three tasks compared to only one participant who used only class diagrams 6. On average, participants who used OOGs performed their activities in less time, by 22%–60% (Fig. 4(b)), and by browsing less irrelevant code, by 10%–60% (Fig. 4(a)). We summarize our results using box-and-whisker plots. The dots on each plot indicate that there were outliers in both groups for both variables (Fig. 5(a), 5(c), 6(a), and6(b)).

To measure the differences between the two groups, we also used tests for statistical significance. Due to our small sample size, we selected the non-parametric equivalent of Student's t-test, the Mann Whitney u-test. We display the results of applying the lower one-sided u-tests below each plot as p-values. The p-values indicate that the mean difference in the code explored was significant in only T1 (Table 7). For the time spent, we found that the mean difference was significant in only T3. Therefore, we cannot reject the null hypotheses across all tasks for the two measures.

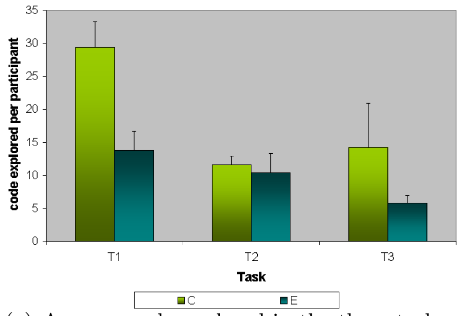
Table 6: Summary of participants' performance and relation to programming experience. Time is measured in minutes. Code explored is measured in units of code (class or method).

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

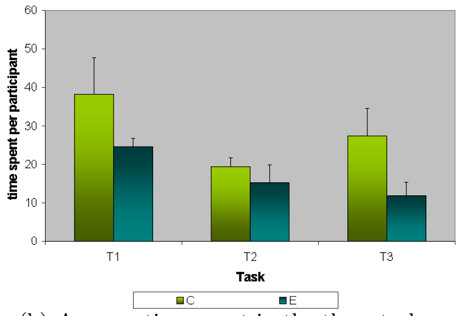
Qualitative analysis based on Questionnaires. In the rest of this section, we support each of our hypotheses with qualitative data including analysis of questionnaires. Tables referenced in this section are included in Appendix 9.

Table 7: Summary of the quantitative measures.

Variable	Task	p-value	Percentage Difference
Code	T1	0.00794	53%
Code	T2	0.264	10%
Code	T3	0.0687	60%
Time	T1	0.147	36%
Time	T2	0.232	22%
Time	T3	0.0476	60%



(a) Average code explored in the three tasks.



(b) Average time spent in the three tasks.

Figure 4: The mean difference for two variables in the three tasks.

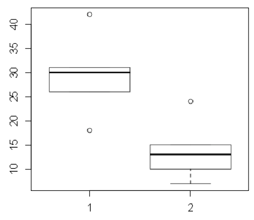
H1: Developers who use OOGs answer questions about the object structure that they cannot answer using class diagrams

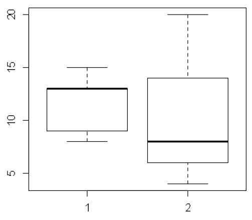
Observation1: Developers perform their tasks in a sequence of activities related to questions about the object structure. The participants engaged in activities that often involved questions about the object structure (Table 5), so we classified the questions in each activity using our coding model (Table 4). The participants performed some of these activities during the planning phase, and wanted to know to which tier an object belonged (Is-In-Tier). For example, in T1 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 in our coding model. Some other activities during the implementation phase often involved questions about how to get an object inside the class where the modification should be implemented. For example, for Activity T1.e, we used the code How-To-Get-X. Table 13 shows excerpts from the participants thought process on which we relied to come up with these activities and codes.

Observation2: Developers use relations between objects on an OOG to answer their questions about the object structure. The OOG helped the E participants answer their questions about the object structure most of the time (Table 16). Since the C participants did not have access to OOGs, they were either unable to find answers or struggled using other means. According to the transcripts and the answers to QX.1 (Table 2), the C participants used the following resources: class diagrams, Package Explorer (Eclipse), file search (Eclipse), documentation (JavaDoc), or reading through code (Table 17).

H2: Developers who use OOGs explore fewer code elements

Observation3: Developers use OOGs to understand object relations, and navigate fewer paths through the code. According to the transcripts, the E participants explored fewer code elements compared to the C participants especially on T1. To illustrate, we compare between two participants from each group while they were doing the same activity (Tables 8 and 9). The participants followed different navigation paths, with one path focusing on objects while the other path focusing on classes, then investigating further within the class to answer a question. C1 followed 3 paths. In Path1, he navigated methods and classes in Eclipse but did not succeed. In Path2, he referred to class diagrams but did not succeed. In Path3, he succeeded by reading through the code. On the other hand, E4 followed 2 paths: in Path1, he navigated


(a) T1 (p = 0.008)


(b) T2 (p = 0.26)

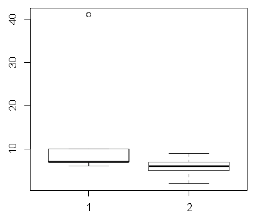
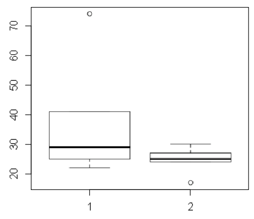
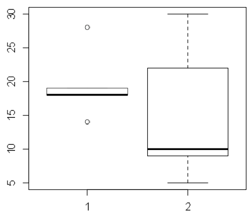

(c) T3 (p = 0.07)

Figure 5: 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 for the E group. Only the code explored in T1 was significantly less for the E group (p=0.007).


(a) T1 (p = 0.15)


(b) T2 (p = 0.23)

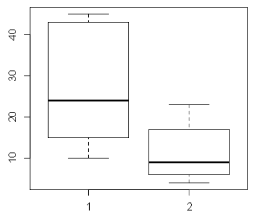

(c) T3 (p = 0.05)

Figure 6: 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 obvious. The E participants completed T3 in significantly less time (p=0.04).

some methods and classes, then he referred to the OOG where he studied the edge between `moveCommand` and `boardDrawing` (Fig. 1(b)), then he expanded the `boardDrawing` object where he found the `figureMap` object that he was looking for (Fig. 1(c)).

Observation4: Developers use traceability links on the OOG to navigate to relevant code. Tables 8 and 9 illustrate how E4 noticed an edge between two objects on the OOG, then traced to the corresponding field declaration to answer his question. C1, on the other hand, explored irrelevant code before he answered his question, and was distracted twice. According to the transcripts, C1 used the file search which disappointed him since the search result pointed to a JavaDoc comment (Tables 16 and 17). Based on Tables 16 and 17 and answers to QX.1 (Tables 18 and 19), the C participants seemed to be navigating the code randomly without having a specific methodology, which resulted in them being distracted especially during the first hour.

Moreover, all the participants in the study narrowed the scope of navigations down to 7 classes for all tasks. However, the C participants spent along time exploring much irrelevant code before they narrowed the scope compared to the E participants. For example, when C2 started implementing T1, he realized that he should not have explored many classes, and should have focused on only three.

All the E participants, on the other hand, worked within the scope of these 7 classes from the beginning. They also followed common navigation paths in certain activities. For example, to see if there was a way to get the data structure that holds all pieces in certain positions on the board, they referred back to the OOG to investigate further. They often expanded the `boardDrawing` or `boardFigure` objects to answer this question. They found an Is-Part-Of relation between `boardDrawing` and `figureMap` which indeed answered there question as `figureMap` was the data structure (a `HashMap`) for which they were looking (Fig. 1(c)).

H3: Developers who use OOGs complete their tasks faster

Observation5: Developers who use OOGs take less time to understand the global object structure. In general, all participants took their time to understand the code then perform the implementation.

Table 8: Two navigation paths followed by C1 for activity T1.b.

Path	Time	Navigation Target (Class/Method)	Outcome

1	6	`BoardFigure`
		`ImageFigure`
		`AbstractFigure`
		`PositioningStrategy`
		`PositioningStrategy.CalculateFigure...()`
		`PositioningStrategy`
		`ChessBoardPositioningStrategy`
		`Position`
		`PositioningStrategy`
		`FigureFactory`
		`Position`
			Failure

2	4	CD7 CD1 CD6 CD1	Failure

3	2	`FigureFactory`
		`BreakThroughPieceFactory`
		`BreakThroughPieceFactory.generate...()`
			Success

Table 9: Two navigation paths followed by E4 for activity T1.b.

Path	Time	Navigation Target (Class/Method/Object)	Outcome

1	6	`CompositeFigure.add()`
		`Figure`
		`CompositeFigure`
			Failure

2	1	`moveCommand:MoveCommand`
		`boardDrawing:BoardDrawing`
		`AbstractFigure::owned`
		`BoardDrawing::MAPS:figureMap:HashMap`
			Success

However, we observed a difference in the time spent among the participants to answer QX.6 (Table 2). Overall, the E participants took less time to understand how the objects communicated. Also, QX.4 (Table 2) required the ability to map GUI components to their corresponding code elements. Answers to QX.4 (Tables 20 and 21) indicate that some of the C participants knew immediately that `BoardDrawing` represented the chess board. One C participant made a correct guess about `BoardDrawing` but spent a relatively long time in validating his assumption of where to perform the validation logic (Activity T1.a).

On the other hand, 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` (Fig. 1(b)). Also, E3 and E5 used Part-Of relations to check whether there was a container-containee relation between `boardDrawing` and `boardFigure` (Table. 16).

Admittedly, some E participants spent a long time on certain activities, but they spent extra time investigating the Eclipse Type Hierarchy. For example, E5 and E3 spent a long time compared to their colleagues only to find out that `BoardDrawing` is a `BoardGameObserver` and can be safely cast to that type. Thus, all their time was spent effectively unlike the C participants who wasted their time being distracted.

Observation6: Developers who use OOGs follow shorter paths to complete their tasks. Only E1 explored many code elements and spent a long time compared to the other E participants, presumably since he refused to use diagrams and insisted on browsing the code. As another example, E4 spent a long time on activity T1.b since he followed two paths (Table 9). 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." (E4, T1)

In the other path, he used the OOG Viewer which helped him be more systematic and get to the answer faster:

"It's not the position. . . it's not `moveCommand`. I'd probably look inside `boardFigure` because I'm still thinking this is the actual thing itself, but maybe not. What do I do in order to look inside this? I would show the internals [F3, Fig. 3]. Okay, a list of figures, aha!" (E4, T1)

This scenario can be compared to C1 (Table 8). 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 most of his time interpreting the OOG to learn about key object relations instead of just browsing the code. Since C1 did not use the OOG, it was difficult for him to determine how to get to an object (How-To-Get-X). Instead, he looked for associations in the class diagrams, and was left guessing. As a result, E4 followed a shorter navigation path, and explored only code relevant to the task.

H4: Developers who use OOGs succeed on their tasks

Observation7: Developers who use OOGs follow fewer unsuccessful paths. The alternative techniques that the C participants used to answer their questions (Observation2) resulted in them following more unsuccessful paths through the code or taking a relatively long time to complete their tasks. The example discussed in Observation 6 above indicates that E4 followed fewer navigation paths with unsuccessful outcome as compared to C1 who followed two navigation paths with unsuccessful outcomes before reaching a successful outcome.

Some C participants completed the tasks, but their implementation was either a hack or introduced a bug, and they did not leave enough time to test their implementation or refactor their code. Participant C4, for instance, encountered a bug (null pointer exception) and struggled with accessing the `figureMap` object. To solve his problem, he created a public static getter method to get the `figureMap` object because he could not answer a How-To-Get-X question (Table 17).

Observation8: Developers who use OOGs make correct assumptions before implementing their tasks. Several C participants made wrong assumptions about the code during the planning phase. When they started the implementation, they falsified their initial assumptions. Other C participants made wrong assumptions, based their implementation on the wrong assumptions, and encountered the errors during testing. For example, most of the participants struggled to understand the relation between `game` and `boardDrawing`. The C participants assumed that they will be able to get a `drawing` object from the `game`, but when they implemented the change they often relied on code auto-completion in Eclipse which was not always the right way to get to fields or methods since they could be private members:

"Maybe the assumption that we can get the coordinates from here was false!" (C1, T1, 1 hour 30 min)

"The initial assumption that you can get `game` from `boardDrawing` is wrong!" (C2, T1, 1 hour 22 minutes)

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 (Table 18). Then he found the method `adjustFigurePosition()` while he was browsing the code and assumed that this is the method being triggered based on debugging techniques (Table 17).

The E participants, on the other hand, traced from an edge between `game` and `drawing` on the OOG (Fig. 1(b)) which took them to a field of type `BoardGameObserver` inside `GameStub`, which corresponds to a `drawing` object. They used the Eclipse Type hierarchy to understand this relation better when they found that a `boardDrawing` class implements the `BoardGameObserver` interface.

Also, based on the answers to QX.1 (Table 2), most E participants were aware of how the objects communicated compared to only C2. Most C participants did not fully understand the big picture compared to the E participants who used the OOG to locate where to implement a certain feature, based on Is-In-Tier facts, how to get objects from other objects, based on How-To-Get-X facts, and how to dig for a data structure based on Is-Part-Of or Is-Owned facts.

7 Discussion

7.1 Study Design Issues

Our quantitative analysis lacks statistical significance across all three tasks. However, the study exposed some issues in our task design. We designed the tasks to have the participants add the missing game logic to BreakThrough. The tasks T1 and T2, however, were related, and that could have influenced our findings. The results suggest that the difference is significant in the case of T1 for the code measure and in the case of T3 for the time measure. The difference in the time measure increased with T3, presumably because T3 was different from the other two.

In general, some code modification tasks may require less information from the OOG than others. Not all code modification tasks require information about how objects communicate. Admittedly, we could have obtained stronger results by selecting tasks that specifically trigger questions about objects, questions that would be hard to answer by looking at the class diagrams or by browsing the code. We could have even picked tasks that are highly crafted to trigger highly specific navigation of the OOG. But we did not do so, to ensure that the tasks are plausible code modification tasks.

The version of the OOG Viewer that we provided to our participants was not as useful as it could be. We purposely disabled a feature that enables a developer to display partial UML class diagram with the inheritance hierarchy of the types of the field declarations in the program that an object on the OOG merges. We wanted the usage of the OOG Viewer to count as dealing with the object structure rather than generating class diagrams. Enabling this feature would have made the tool more useful, since the participants would have focused intensely on the OOG, instead of having to switch back to Eclipse to invoke the Type Hierarchy feature.

7.2 Threats to Validity

The lack of statistical significance in both measures in some tasks, the high standard deviation in some cases in both groups, and the presence of outliers affected the validity of our conclusion. We consider this experiment as the first to evaluate OOGs, and we believe an external replication of the results with a larger sample size would be necessary to obtain a higher statistical significance across all maintenance tasks.

Our study may have several threats to internal validity. First, the participants had varying experience. Three E participants and one C participant had previous experience that could have affected their performance on the task. However, one of the three experienced E participants performed badly compared to his colleagues. The other two struggled while performing the tasks and needed the OOG to answer their questions. One of the experienced C participants made a wrong assumption, so he took a long time and explored much irrelevant code. Also, we tried to even out the differences between experience levels among participants by randomly assigning them to groups. The E group may have more professional developers, but a closer look at the participants' Java programming experience justifies this assignment. For example, one C participant was an undergraduate student with three years of Java programming experience. This makes him roughly comparable to E5, who is a professional developer with a bachelor's degree and three years of Java experience.

Second, we wanted to prepare the participants to face an unfamiliar tool, and understand the concepts behind ownership and OOGs, but the OOG tutorial may not have been enough. In fact, from our own experience, it takes professional developers several months to master these concepts. Also, by tutoring the OOG and the tool through hands-on exercises, we encouraged the participants to learn to use the tool effectively. Still, the exercises may have prompted the participants to think in terms of the object structure more frequently. Also, the experimenter demonstrated the OOG tutorial on MiniDraw, but using objects that were unrelated to the tasks that the participants had to complete. Admittedly, we should have demonstrated the OOG on an unrelated application. Still, our findings suggest that the tutorial did little to help the participants to understand MiniDraw.

Third, all of the participants might have been overwhelmed by the 10 class diagrams. However, one class diagram helped the participants get a global view of class dependencies and associations with the

BreakThrough class. In fact, the main purpose of this diagram was to show dependency relationships. According to UML specification [23], a dependency relationship is different from an association in a sense that it shows whether two classes depend on or use each other in some way; not necessarily an association. AgileJ treats such diagrams as dependency diagrams [6]. The other five diagrams were less cluttered than this dependency diagram, and were organized by package. The remaining four diagrams were manually generated by the MiniDraw designers, and explained the different roles of the Model, View, and Controller.

Finally, the questionnaires between the tasks may have triggered questions about the object structure that developers would not normally ask on their own. Still, the participants often asked such questions, but in indirect ways or using a different terminology. Also, our questionnaires measured their comprehension level as they made their changes, but provided them with little guidance otherwise, since we never corrected their answers. We only observed how they changed their answers or navigation paths, as they discovered new facts while implementing the tasks.

Several factors could affect the generalizability of our findings. First, the subject system itself may not be representative of all code bases. Second, our tasks may not be representative of real maintenance tasks since they were selected by the experimenter rather than from a bug tracking system. Our results could have been more generalizable had we used bugs or feature requests submitted by outside framework developers. Still, our tasks such as T3 ("implement an undo feature") are fairly broad software engineering tasks that were not specifically crafted to favor OOGs. Third, our participants were mostly graduate students. Although four of them were professional developers, they came from a C# or C++ background and were not proficient in Java. We could perhaps obtain better results by recruiting only experienced Java developers.

7.3 Why Does the OOG Help during Coding Activities?

Understanding the object structure is *fundamental* to the program comprehension of object-oriented code. As a diagram of the run-time structure, an OOG may help answer key program comprehension questions.

Instances matter in object-oriented code. 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 [13], 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 an instance.

Do specific instances really matter? An OOG does not pin things down to individual objects. Instead, it abstracts objects by domains and types. For example, it merges several objects of the same or similar types that are in the same domain. If the code creates many instances of the `BoardFigure` class at runtime, and the latter are all in the `MODEL` domain on the `BreakThrough` object, the OOG shows one `boardFigure` object in the `MODEL` domain (Fig. 1(b)). Still, developers can select a canonical object on the OOG and trace to all the lines of code that may create such an object.

If despite merging objects, OOGs hold enough precision and are still useful for program comprehension, as our study indicates, an instance may not matter in terms of "the particular object". It seems enough to pin things down just to objects of a type that are within a domain. So this leads us to refine the question.

Does information about *types + ownership + domains* answer key questions in program comprehension? It seems that what really matters is the *role* an instance is playing, and information about *types + ownership + domains* give us a richer language for describing that role than type alone. For instance, we can express facts like *"an object of type* A *in domain* D *in an object of type* B", that we show as triplets $\prec$`A,D,B`$\succ$. So the question becomes: *how often does the ability to distinguish the role of instances not just by type, but by named groups (domains) or by position in the run-time structure (ownership), matter for code modification tasks?*

We conjecture that the OOG is useful during coding tasks because it conveys that information. We also conjecture that there are situations where types are not enough, but domain and ownership information gives developers exactly what they need for some task. The following is an excerpt from the think-aloud transcript of a participant in our study:

> *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.* Participant C5 (control group)

Indeed, the code uses many `HashMap` instances, and the one that the developer really needs is the `HashMap` in the `MAPS` domain in the `boardDrawing` object (of type `BoardDrawing`). This fact can be represented as the triplet $\prec$`HashMap, MAPS, BoardDrawing`$\succ$ and is visually obvious on the OOG (Fig. 1(c)). We also conjecture that such examples are reasonably common in object-oriented code. For MiniDraw, we were able to count at least 10 such instances.

8 Related Work

Several diagrams and models have been proposed by researchers to aid in program comprehension. Some of these diagrams have been evaluated empirically to measure their usefulness for program comprehension. In this section, we give an overview of each of these diagrams and how they are different from OOGs. We also discuss how widely, we believe, the usefulness of different types of diagrams have been evaluated empirically by discussing which aspects of these diagrams have been investigated in detail, and how they are different from our evaluation method described in this paper.

8.1 Our previous studies on evaluating OOGs

As part of our work in evaluating the usefulness of OOGs for code modification tasks, we previously conducted an exploratory study in a laboratory setting. We observed three participants perform several code modification tasks while using OOGs [4]. The tasks changed slightly between the participants, so our analysis remained qualitative. Based on our study, we developed a taxonomy of questions that developers ask about objects and their relations (Table 4).

We also conducted a case study [3] that provided qualitative data that an OOG does help developers with program comprehension. During the study, the OOG answered several questions the developer had about the object structure (in Table 4, their codes are Is-In-Tier, Is-Owned-By and Is-Part-Of). The case study, however, involved one participant, which limited the external validity of our findings. The study in this paper overcomes this limitation.

8.2 Evaluating object-based diagrams

We are not the only ones emphasizing the importance of instance-based diagrams. Quite a few researchers have worked on evaluating the usefulness of instance-based models, and proposed solutions to complement the current object-oriented modeling approaches.

UML static structural object-based diagrams. To our knowledge, only few researchers studied UML static object diagrams. Torchiano et al. [28] conducted a controlled experiment and an external replication to evaluate the usefulness of static object diagrams for program comprehension as compared to class diagrams. They found that static object diagrams are significantly more useful for program comprehension when combined with class diagrams than using only class diagrams. Their study was only questionnaire-based and did not involve any code modifications. Tonella et al. [31], on the other hand, compared static object diagrams to dynamic object diagrams.

UML dynamic behavioral object-based diagrams. Much of the research work on object-based diagrams was done on dynamic, behavioral views such as sequence diagrams and collaboration diagrams, in comparison to class diagrams [14, 5]. A study by Hadar et al. [16] on the usefulness of different types of UML diagrams identified that developers consult class diagrams to study static relations, but when they want to understand the dynamic behavior they need diagrams like the sequence diagrams.

Thus, those studies focus on partial views such as sequence diagrams and object diagrams that describe specific scenarios and were manually generated. Our study is the first to evaluate global hierarchical object graphs that are statically extracted from the code, which had been difficult to obtain using prior technology.

8.3 Extracting diagrams of the object structure

Several researchers worked on producing diagrams of the object structure, which were either statically extracted or dynamically extracted from object-oriented Java code.

Statically extracted object graphs. Several approaches extract flat object graphs from object-oriented code either automatically, including WOMBLE [17], AJAX [22], and PANGAEA [29]) or using annotations [19]. While these approaches can be useful for showing object interactions, they share a fundamental scalability limitation. For example, [19] 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, for programs of any size, these approaches will produce a diagram with so many objects that, in practice, the diagram will be barely readable by humans. The SCHOLIA [2] approach extends Lam and Rinard's both to handle hierarchical architectures and to support object-oriented language constructs such as inheritance. Also, to our knowledge, none of the above approaches were evaluated empirically by having developers use flat object graphs during code modification tasks.

Dynamically extracted object graphs. Dynamically extracted object graphs, on the other hand, consider specific executions of the system. Quante evaluated in a controlled study Dynamic Object Process Graphs (DOPGs) [24]. A DOPG is quite different from an OOG. A DOPG is a statically extracted interprocedural 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 callgraph than to a points-to graph. Quante found that the DOPG helped for concept location tasks on one codebase 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. Demsky and Rinard also used dynamic analysis to extract two types of role-based object diagrams, role transition diagrams and role relationship diagrams [12]. Dynamically extracted diagrams reflect only a few program runs, while an OOG is sound and reflects all possible objects and relations that may occur in any program run. In order to make decisions related to code modification tasks, developers should be able to base their decisions on a sound diagram.

8.4 Evaluating class-based diagrams

Studying the effectiveness of design-time or compile-time diagrams, such as UML class-based diagrams has been the focus of many research studies.

UML structural class-based diagrams. Ricca et al. [25] and Gabriele et al. [11] studied, in controlled experiments, the usefulness of UML class diagrams for program comprehension. However, their studies were only questionnaire based and focused on different aspects. For example, [25] studied the effect of stereotypes for web application design comprehension, whereas [11] studied whether class diagrams were better than Entity-Relationship (ER) diagrams for data model comprehension. Therefore, the diagrams that they used were manually generated and were not reverse-engineered from the code and so there was no direct correspondence between the elements on the diagrams and the code elements.

Enhancing class-based diagrams. Most of the studies on UML diagrams have identified several weaknesses in the currently available UML diagrams and notations and that we have to move our attention to instance-based rather than only class-based models. Torchiano et al. [20] have proposed an extension to the class-centered model to hierarchical instance models based on a schema extracted from the class model. Their approach is manual and describes a system model like a business process. Our approach is based on automatically reverse-engineering a model from the code and thus provides developers with models that are consistent with the code. Other researchers have proposed possible enhancements to UML diagrams such as integrating both UML dynamic and static views of the source code. Lallchandani et al. [18] worked on combining information extracted from sequence diagrams along with those from class and state-machine diagrams into a model dependency graph (MDG). Also, Gogolla et al. [15] proposed the use of layered graphs to describe both type graphs and instance graphs. Riehle et al. [26] used “collaboration roles” to enhance class diagrams with information about design patterns. He found that the role modeling adds more information to the existing documentation.

9 Conclusion

We designed and conducted the first controlled experiment to evaluate global, hierarchical, Ownership Object Graphs (OOGs). Our experiment provided strong evidence in some of the tasks to support the main research hypothesis that, as diagrams of the runtime structure, OOGs are useful for coding tasks. On average, we found that OOGs improve developers ability to answer questions about the object structure by taking less time and exploring less code. Our experiment provided statistical generalization for some of the tasks, and was preceded by preliminary work that provided analytical generalization [3]. We plan to replicate the results of this experiment by recruiting a larger number of participants to gain a higher statistical significance.

Given the considerable costs of software maintenance and evolution, a measured improvement in developers' performance on code modification tasks justifies de-emphasizing class diagrams, which are reasonably mature, and instead, focusing on tools that extract complementary diagrams of the run-time structure. We are enhancing the OOG to show, in addition to points-to relations, richer data-flow communication [33]. We are also mining the usage data we gathered in this study to improve the tools' usability and usefulness.

References

[1] M. Abi-Antoun. *Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure.* PhD thesis, Carnegie Mellon University, 2010.

[2] M. Abi-Antoun and J. Aldrich. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *OOPSLA*, pages 321–340, 2009.

[3] M. Abi-Antoun and N. Ammar. A Case Study in Evaluating the Usefulness of the Run-time Structure during Coding Tasks. In *Workshop on Human Aspects of Software Engineering (HAoSE)*, pages 1–6, 2010.

[4] M. Abi-Antoun, N. Ammar, and T. LaToza. Questions about Object Structure during Coding Activities. In *Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 64–71, 2010.

[5] S. Abrahao, E. Insfran, C. Gravino, and G. Scanniello. On the effectiveness of dynamic modeling in UML: Results from an external replication. In *Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2009.

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

[7] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, pages 1–25, 2004.

[8] N. Ammar. Evaluation of the Usefulness of Diagrams of the Run-Time Structure for Coding Activities. Master's thesis, Wayne State University, 2011.

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

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

[11] A. De Lucia, C. Gravino, R. Oliveto, and G. Tortora. Data Model Comprehension: An Empirical Comparison of ER and UML Class Diagrams. In *ICPC*, 2008.

[12] B. Demsky and M. Rinard. Role-based exploration of object-oriented programs. In *ICSE*, pages 313–324, 2002.

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

[14] M. Genero, J. A. Cruz-Lemus, D. Caivano, S. Abrahão, E. Insfran, and J. A. Carsí. Assessing the Influence of Stereotypes on the Comprehension of UML Sequence Diagrams: A Controlled Experiment. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 280–294, 2008.

[15] M. Gogolla, J. marie Favre, and F. Bttner. On squeezing M0, M1, M2, and M3 into a single object diagram. Technical report, 2005.

[16] I. Hadar and O. Hazzan. On the Contribution of UML Diagrams to Software System Comprehension. *Journal of Object Technology*, 3(1):143–156, 2004.

[17] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *TSE*, 27(2):156–169, 2001.

[18] J. Lallchandani and R. Mall. Integrated state-based dynamic slicing technique for uml models. *Software, IET*, 4(1):55 –78, feb. 2010.

[19] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP*, pages 275–302, 2003.

[20] G. B. Marco, M. Torchiano, and R. Agarwal. Modeling complex systems: Class models and instance models, 1999.

[21] MiniDraw. `www.baerbak.com`.

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

[23] OMG. Unified Modeling Language (UML). `www.omg.org/`, 2010.

[24] J. Quante. Do dynamic object process graphs support program understanding? - a controlled experiment. In *ICPC*, pages 73–82, 2008.

[25] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato. How Developers' Experience and Ability Influence Web Application Comprehension Tasks Supported by UML Stereotypes: a Series of Four Experiments. *TSE*, 36(1), 2010.

[26] D. Riehle. Junit 3.8 documented using collaborations. *SIGSOFT Softw. Eng. Notes*, 33:5:1–5:28, March 2008.

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

[28] G. Scanniello, F. Ricca, and M. Torchiano. On the effectiveness of the UML object diagrams: a replicated experiment. *IET Seminar Digests*, 2011(1):76–85, 2011.

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

[30] M.-A. D. Storey, H. A. Müller, and K. Wong. Manipulating and Documenting Software Structures. In P. Eades and K. Zhang, editors, *Software Visualization*, 1998.

[31] P. Tonella and A. Potrich. Static and Dynamic C++ Code Analysis for the Recovery of the Object Diagram. In *ICSM*, pages 54–63, 2002.

[32] P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code (Monographs in Computer Science)*. Springer-Verlag, 2004.

[33] R. Vanciu and M. Abi-Antoun. Extracting Dataflow Communication from Object-Oriented Code. Technical report, WSU, 2011.

APPENDIX

Most of this appendix is available in Ammar's thesis [8]. The appendix includes the following:

1. Different versions of class diagrams provided to participants (Table 10).
2. The OOG viewer tool features (Table 11).
3. Excerpts from the transcripts (Fig. 7).
4. Eclipse features used by the participants during the experiment (Table 12).
5. Questions about the object structure common to all participants (Table 13).
6. Hands-on exercises used to tutor the OOG (Table 14).
7. Exit interview questions (Table 15).
8. Some of the methods followed by E participants to answer questions (Table 16).
9. Some of the methods followed by C participants to answer questions (Table 17).
10. Responses of the C participants to QX.1 (Table 18).
11. Responses of the E participants to QX.1 (Table 19).
12. Responses of the C participants to question QX.4 (Table 20).
13. Responses of the E participants to QX.4 (Table 21).

Table 10: Different versions of class diagrams provided to participants.

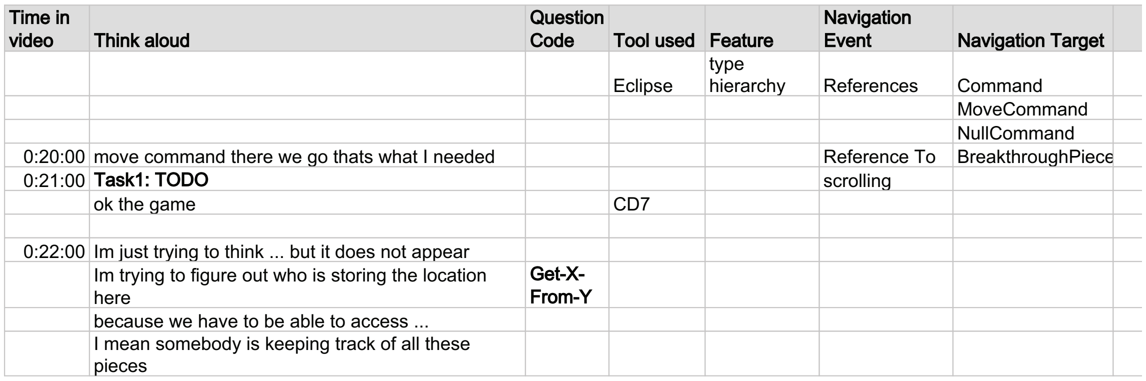
Diagram	**Source**	**Purpose**
CD1	Reverse engineered using AgileJ	classes in the `minidraw.boardgame` package
CD2	Reverse engineered using AgileJ	classes in `minidraw.breakthrough` package
CD3	Reverse engineered using AgileJ	classes in `minidraw.framework` package
CD4	Reverse engineered using AgileJ	classes in `minidraw.standard` package
CD5	Reverse engineered using AgileJ	classes in `minidraw.standard.handlers` package
CD6	Reverse engineered using AgileJ	dependency diagram of `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`

Table 11: 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.

Table 12: 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
Ec.F10	Debugger	



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

Table 13: 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 [generatePieceMultimap()] 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
	"see command is local to the boardfigure and that's not necessarily good because we need to access from anywhere"	Is-Owned
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

Table 14: 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 15: 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?

Table 16: Some of the methods followed by E participants to answer questions about the object structure while doing Task 1. The numbers in the Tool column refer to the features listed in Table 11.

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	"A `boardFigure`. . . Let me go deep into this. 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 17: Some of the methods followed by C participants to answer questions about the object structure. The numbers in the Tool column refer to the features listed in Table 12.

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 not 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
	"you know as a high level view its helpful but some of the ones that are broken down are more helpful for specific purpose"		CD6		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
	"I didn't assume any kind of a method to be implemented. I just assumed that maybe searching for capture might narrow my search down a little bit"	Has-Label		Ec.F5	Fail
C2	"okay so observer.pieceMovedEvent() who impelents this BoardDrawing? BoardGameObserver. Back to the begining! Its just cyclical? i just wanna see who is calling you again? Gamestub, round and around again!"			Ec.F1	Fail
	"who's calling editor? . . . very convoluted here!"	Points-To		Ec.F5	Fail
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
C4	"It's not right to do this! Maybe I should debug. I guess I don't need to recreate it. I just need to refer to the one who created it somewhere else." "It's easier for me later I want to access this `figureMap`. So it will save me the hassle of finding where this drawing object is created. "Is there a quick way, fast way to find out where there are people creating that `boardDrawing` object?"	May-Alias How-To-Get-X How-To-Get-X		Ec.F10	

Table 18: Responses of the C participants to QX.1(Table 2).

P	Answer
C1	"I'm just gonna poke around different classes seeing which one might be I know thats probably not the most efficient method"
C2	"... [after 1 hour 30 min] so basically in boarddrawing because we got figuremap in there so that's like a centralized location which is what I thought about initially instead of wandering all that nonsense!"
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"

Table 19: Responses of the E participants to QX.1 (Table 2).

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”

Table 20: Responses of the C participants to question QX.4 (Table 2).

P	Answer
C1	“trying to narrow the scope down. . . 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()]”
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”
C4	”I actually need another parameter that represents the current game board like at this position which piece is there.. ok i want to look at the game.move() to see how exactly the move works its kind of related. Ok Im not exactly sure what the list is but looking from the code I guess the list should have the information i need...ok i guess now i got some clue”

Table 21: Responses of the E participants to QX.4 (Table 2).

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`”