Empirical Evaluation of Diagrams of the Run-time Structure for Coding Tasks

Nariman Ammar **Marwan Abi-Antoun**

August 25, 2012

Department of Computer Science
Wayne State University
Detroit, MI 48202

Keywords: object diagrams, controlled experiment

Abstract

With object-oriented design, it is at least as important—possibly more important—to understand the run-time structure, in terms of objects and their relations, as to understand the code structure dealing with source files, classes and packages. Today, many tools and diagrams help developers understand the code structure. Diagrams of the run-time structure, however, are much less mature.

One diagram of the run-time structure is a statically extracted, global, hierarchical Ownership Object Graph (OOG). The OOG conveys architectural abstraction by ownership hierarchy by showing architecturally significant objects near the top of the hierarchy and data structures further down. In an OOG, objects are also organized into named, conceptual groups called domains.

We posit that types are not enough for object-oriented code comprehension, and that the OOG improves comprehension by giving developers the ability to distinguish the role that an object plays, not only by type, but also by named groups (domains) or by position in the run-time structure (ownership), i.e.,$types + ownership + domains$.

We evaluate, in a controlled experiment, whether an OOG, as a diagram of the run-time structure, improves object-oriented code comprehension by giving developers the ability to distinguish the role that an object plays, not only by type, but also by named groups (domains) or by position in the run-time structure (ownership). We observed 10 participants, for 3 hours each, perform three feature implementation tasks on a framework application. Our experiment identified that, on average, the participants who used information obtained from OOGs, in addition to information obtained from the code structure, i.e., class diagrams and browsing or reading the code in Eclipse, always outperformed the participants who used only information obtained from the code structure, i.e., class diagrams and browsing or reading the code in Eclipse. Our results indicate that, on average, the OOG had a positive effect of varying extents on comprehension that reduced the time spent by 22%-60% and irrelevant code explored by 10%-60%. The difference was significant ($p < 0.05$) for two of the tasks.

Contents

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 [13]. To support comprehension, researchers have produced many tools to visualize the structure of a software system. These tools are based on the widespread belief that diagrams are useful for 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. In object-oriented design patterns, for example, much of the functionality is determined by what instances point to what other instances [23]. For instance, in the Observer design pattern, 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. Moreover, for object-oriented code, the run-time structure is often quite different from the code structure. Thus, a Diagram of the Run-time Structure (DRS) can be highly complementary to a Diagram of the Code Structure (DCS) and can answer several crucial questions to developers while performing code modifications.

Currently, there are many widely supported DCS tools, but DRS tools are much less mature. We broadly include in DCS tools various code exploration features found in modern IDEs such as Eclipse, even if they do not display a diagram. For instance, they may show the classes in a project as a tree (Eclipse Package Explorer), or allow searching for strings in files and show the results as a list. But they are still showing code entities, as opposed to run-time entities.

Currently, many tools extract DCS, but tools to extract DRS are much less mature. One DRS recently proposed and formalized by Abi-Antoun and Aldrich [4] is a statically extracted, global, hierarchical Ownership Object Graph (OOG). The OOG is a sound approximation of all run-time objects across the entire system with all possible points-to relations between those objects. A DRS assigns different instances of the same type different roles according to the context in which they occur. In particular, on the OOG, the context is expressed using the notion of an *ownership domain*. A domain is a conceptual grouping of objects, and the domain name conveys design intent in the code based on *ownership domain annotations*. Due to space limits we describe the annotation language in [2, Appendix A]. In this paper, we deemphasize evaluating the effort of extracting and refining OOGs, which we have recently measured [6]. The interested reader can refer to [12] and [11, Chap.3] to read more about the process of extracting and refining OOGs for this experiment. This paper sheds more light on the inherent difficulties of object-oriented comprehension, and contributes the following:

- A theory in comprehension in terms of facts that developers can learn from a DRS; Our theory predicts that the ability to distinguish the role that an instance plays not just by type, but by named groups (domains) or by position in the run-time structure (ownership), enables developers to answer challenging questions about the RS.We illustrate how types are clearly not enough for object-oriented code comprehension, and how information content on a DRS (facts about $types + ownership + domains$) gives developers a richer language for describing that role than type alone.
- A preliminary classification of questions that developers ask about the run-time structure;
- A controlled experiment to investigate whether developers can use a DRS to answer questions about the run-time structure during code modifications more easily than developers who do not have access to such a diagram and rely on only type information provided by DCS tools.

Outline. In the rest of this paper, we give background on OOGs (Section 2). In Section 3, we explain our theory of comprehension (Section 3.1) and define a classification of facts about the run-time structure based on our theory, and we motivate the need for OOGs to fill gaps in todays diagrams and currently used DCS tools (Section 3.2). Next, we present our controlled experiment. describe our method (Section 4), our analysis (Section 5), and both quantitative and qualitative results (Section 6). We then discuss some limitations, threats to validity and lessons learned (Section 7). Finally, we discuss related work (Section 8) and conclude.

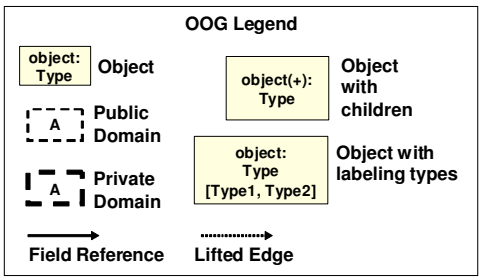


Figure 1: OOG graphical notation

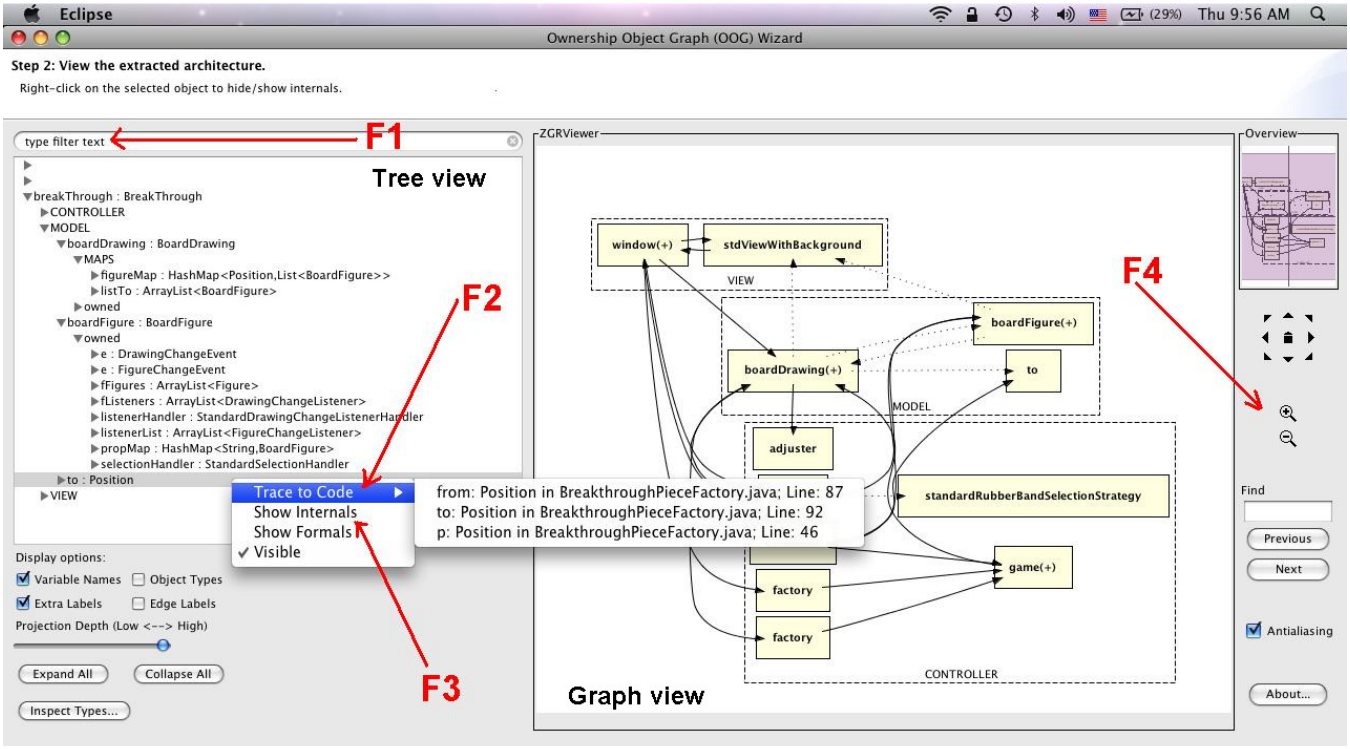


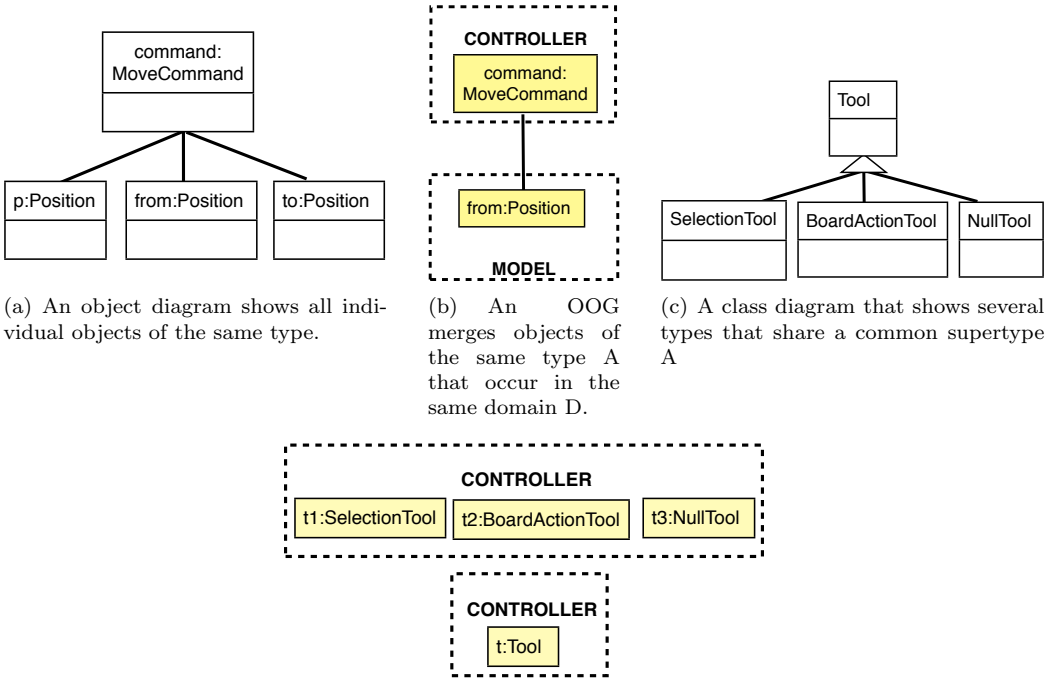
Figure 2: A viewer to interactively navigate the OOG. The graph shows the OOG with nested boxes indicating objects. The tree enables developers to search for an object (F1), trace from a selected object or edge to the code in Eclipse (F2), and collapse or expand an objects sub-structure (F3).

2 Background on OOGs

Most extracted object graphs are flat and as a result, cannot convey high-level understanding. Since hierarchy is an effective idea for shrinking a large flat graph [58], we use annotations to supply the missing architectural hierarchy. The annotations specify, within the code, the original developer's design intent as minimally invasive hints about object encapsulation, logical containment and architectural tiers. The annotations then enable a static analysis to extract a global, hierarchical ownership object graph, the OOG. In this section, we give some background on the key properties of OOGs (Section 2.1) and the OOG extraction process (Section 2.2).

2.1 Key Properties of the OOG

- **The OOG displays information about the run-time structure.** The OOG is a sound approximation of all run-time objects across the entire system with all possible points-to relations between those objects. On the OOG, solid-filled boxes represent objects and solid edges represent points-to relations between objects (Fig. 1). Even though the OOG shows some information about types such as the type of an object and optional labeling types (Fig. 1), it does not show classes, or static relations such as inheritance.
- **The OOG displays abstractions of run-time objects.** The OOG does not pin things down to individual objects. Instead, it abstracts objects by domain and by type. First, the OOG provides abstraction by ownership hierarchy. The hierarchical representation of the OOG enables both high-level understanding and detail, and promotes object-oriented comprehension by hiding irrelevant details and viewing them at different levels in the hierarchy. The architectural extractors use ownership domain annotations to place objects from the application domain in high level logical groups to express the high level architecture. Then they hide data objects such as data structures within high-level abstractions. Second, the OOG displays exactly one unique representative for each run-time object, i.e., if the code creates several instances of type `A` at run-time, and the latter are all in the same domain `D`, the OOG displays all those objects merged into one canonical object (Fig. 3) with the ability to investigate each distinct instance creation expression in the code (F2, Fig 2). Third, the OOG provides abstraction by types (ABT) by collapsing objects further—within a domain—based on the objects' declared types in the code, and optional input from the extractors indicating the architecturally significant types. If several objects share a common super type `A`, and they all occur in the same domain `D`, the OOG displays them as one object.
  However, the OOG is not too abstract to the point of being useless. To avoid excessive merging of objects, the extractors either place objects of the same type, that serve different purposes and so may not alias, in different domains. Another technique that the extractors follow to avoid merging is to place objects of similar types in the same domain but provide as input a set of design intent types to govern the level in the type hierarchy at which the merging should occur. For example, in some design patterns, several types share a common listener interface type but instances of some of these types are in the data tier while instances of other types are in the user interface tier. In that case, the extractors do not indicate the listener interface type as the common supertype to avoid loosing important details.
- **The OOG collapses object nodes based on containment, ownership and type structures,** not according to where objects are syntactically declared in the program, some naming convention or a graph clustering algorithm.
- **The OOG is sound.** Each runtime object has exactly one representative in the OOG. Also, the OOG has edges that correspond to all possible runtime points-to relations between those objects. As with any static analysis, reflection, dynamic code loading or native calls may introduce unknown objects and edges into the system. These external entities can be summarized using "virtual" fields in the annotations.
- **The OOG is statically extracted.** Developers use the OOG during code modification tasks by extracting it statically without having to run the program while they are busy coding. Using a dynamic analysis requires that developers both run the program and provide an extensive and thorough test suite to exercise the analysis. While a dynamic analysis can be more scalable and more precise than a static analysis, it suffers from a few problems. First, a dynamic object diagram may not reflect important objects or relations that show up only in other executions. Second, a dynamic analysis may not be repeatable, i.e., changing the inputs or executing different test-cases might produce different results.
  Developers who require actual relations between objects and actual multiplicity of the objects created can use a dynamic object diagram, e.g., [21, 47], which can show the exact number of instances and the actual relations in a given program run. Facts that may hold for one program run, however, may not hold for a different run.
- **The OOG groups conceptually related objects into domains.** A domain is a conceptual



(a) An object diagram shows all individual objects of the same type.

(b) An OOG merges objects of the same type A that occur in the same domain D.

(c) A class diagram that shows several types that share a common supertype A

(d) OOG abstracts objects that share a common super type A and that occur in the same domain D; abstraction by types (ABT)

Figure 3: OOG does not pin things down to individual objects. Instead, it abstracts objects by domain and by type

partitioning of functionality. The similar notion of a tier exists in many architectural description languages. Developers often think of their systems as following some tiered style such as Document-View, Model-View-Controller or Presentation-Logic-Data [23, Ch1]

- **The OOG groups objects into domains at multiple levels in the hierarchy.** Our current visualization uses box nesting to indicate containment of objects inside domains and domains inside objects (Fig. 1). Dashed-border, white-filled boxes represent domains. A (+) symbol on an object or a domain indicates that it has a collapsed sub-structure.
  An object on the OOG can have one or more nested domains, with other objects inside them; an object is in exactly one domain. Hierarchy enables both high-level understanding and detail. A top-level domain on the OOG corresponds to an architectural run-time tier. A low-level domain groups conceptually or functionally related objects together. Domains also identify accessibility of objects. Logical containment is achieved using a public domain, represented by a thin, dashed-border box (Fig. 1), to make an object conceptually *part of* another object. Strict encapsulation is accomplished using a private domain, represented by a thick, dashed-border box, to make an object *owned by* another object so the owned object cannot be accessed by other objects without going through the parent object.
- **The OOG is a global, rather than a task-specific view.** The OOG is a single global model, extracted by selecting a root class. Developers control the level of visual detail, by expanding or collapsing selected objects or domains (F3, Fig. 2). The OOG shows all the objects across the entire system, similarly to a global diagram of the run-time architecture. It is intended to be complementary to low-level object diagrams which often just illustrate some key interactions between a few selected objects. Developers interested in how a specific set of objects interact , within a specific method, can launch a highly precise shape analysis [53, 39].
- **The OOG conveys the design intent in the code and the annotations.** Since an OOG is

```
new BreakThrough<SHARED>(); // Root expression
class BreakThrough<OWNER> { // Root class used as starting point for extracting OOG
  public domain VIEW, CONTROLLER, MODEL; // Declare top−level domains

  GameStub<CONTROLLER,VIEW,MODEL> gameStub = new ...;
  DrawingEditor<VIEW,CONTROLLER,MODEL> window = new ...;
  BoardDrawing<MODEL,VIEW,CONTROLLER> drawing = new ...;
}
class BoardDrawing<D, U, L> // Declare domain parameters
      extends StandardDrawing<D,U,L> ... // Handle inheritance {
  domain owned; // Declare private domain
  public domain MAPS; // Declare public domain

  // The outer annotation is for the container itself
  // The inner annotations are for the contained elements (keys and values)
  // figureMap: map each location to the set of images positioned on it
  Map<MAPS, D Position, MAPS List<D BoardFigure>> figureMap = new ...;

  // propMap: map graphical (x,y) positions to the props of the game
  Map<owned, SHARED String, D BoardFigure<U,L,D>> propMap = new ...;
}
```

Figure 4: The root class used in MiniDraw to extract OOG.

extracted from the code, it can be assured against the code. To express the design intent missing from other diagrams, the architectural extractors follow a design-intent-based approach for object graph extraction using ownership domain annotations. The annotations supply missing design intent that cannot be inferred from the code. For example, to express the architecture of the system, the extractors group objects of core types into different architectural tiers. If the extractors want to convey the design intent that different instances serve different conceptual purposes, they must put them into different domains. To convey that an instance is logically contained by another, the extractors place it in a public domain. If the instance is strictly encapsulated by another, the extractors place it in a private domain. The extractors follow a type checker to make sure that the annotations are with each other and with the the code.

- **The OOG enables traceability to code.** 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." Solid edges on the OOG represent field references. A dotted edge is an edge that is lifted from a hidden child object to the nearest visible ancestor in the object hierarchy. Lifted edges between objects in top-level domains summarize field reference edges between objects inside these domains.
  Since the OOG is a hierarchical representation, we developed an interactive viewer of the OOG in Eclipse to allow expanding or collapsing the object's sub-structure, while browsing the code in Eclipse (Fig.2). Using the viewer, a developer can trace from each node or edge on the OOG to the underlying lines of code (F2, Fig.2). This way, the OOG can help the developer locate where to implement a change. First, each node that represents a "canonical object" on the OOG represents one or more run-time objects from which a developer can trace to the corresponding *instance creation expression* (`new A()`). In the case of object merging, developers can select a canonical object on the OOG and trace to all the lines of code that may create such an object. Second, for each object node on the OOG, a developer can explore all incoming or outgoing points-to edges then trace to the corresponding *field declaration* (`A f`).

2.2 Extracting OOGs

For this experiment, we used MiniDraw [40], a pedagogical object-oriented framework that consists of around 1,500 lines of Java code, 31 classes and 17 interfaces. We briefly discuss the process to extract OOGs for MiniDraw. 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.

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 [10], and a typechecker checks that the annotations are consistent with each other and with the code. In case of inconsistencies, the typechecker produces annotation warnings. The warnings produced by the type checker are alerts that the OOG may be unsound, so we fix the annotation warnings before extracting OOGs. We describe the annotation language elsewhere [2, 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. 4). The example shows how architectural tiers, such as `MODEL`, `VIEW`, `CONTROLLER`, can be represented as top-level domains (line 3). Architectural hierarchy is encoded using either logical containment or strict encapsulation. For example, the architectural extractor declares a public domain such as `MAPS` on the class `BoardDrawing` (line 11), to make the `figureMap` object conceptually *part of* an object of type `BoardDrawing`. He declares a private domain, such as `owned` on the class `BoardDrawing` (line 11), to make the `propMap` object *owned by* an object of type `BoardDrawing` (line 20) so it cannot be accessed by other objects without accessing the parent object.

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 refine the annotations to obtain a less cluttered OOG by refining the abstraction by ownership hierarchy, 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 convey their design intent. The goal of this experiment is not to optimize the process of extracting or refining OOGs. It is in evaluating whether they are useful for code modification, in order to justify additional research into tools to support them. Since we did not have access to the original MiniDraw designers, one of the authors of this paper performed a few code modification tasks and helped refine the MiniDraw OOG.

3 A Theory of Program Comprehension

Understanding the run-time structure is *fundamental* to the comprehension of object-oriented code. In this section, we define a theory for program comprehension of object-oriented code that types are not enough and that developers need more information to distinguish between different instances of the same type.

3.1 Theory: Types are not Enough; Instances Matter

We define a theory in comprehension in terms of facts about the run-time structure that developers can learn from an OOG (Table 1).

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 example, in the Observer design pattern [23], understanding "what" gets notified during a change notification is crucial for the operation of the system, but "what" does not usually mean a type, "what" means an instance.

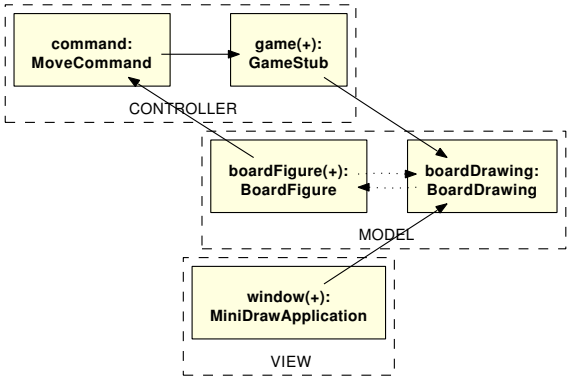


Figure 5: Top-level OOG with run-time tiers as top-level domains.

Do specific instances really matter? The OOG merges instances of the same type that are in the same domain, e.g. `Position` (Fig. 2). In addition, an OOG with abstraction by types merges instances of related types. If despite merging objects, OOGs hold enough precision and are still useful for comprehension, as our experiment will demonstrate later, 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 (May-Alias,Table 1).

Does information about *types+ownership+domains* on the OOG answer key questions in program comprehension? We believe that what really matters is the *role* an instance is playing, and information about *types + ownership + domains* gives us a richer language for describing that role than type alone.

Our theory predicts that an OOG can answer developers questions about the run-time structure during code modifications more easily by providing them with the ability to distinguish the role that an instance plays not just by *type*, but by named groups (*domains*) (Is-In-Tier) or by position in the run-time structure (*ownership*) (Is-Owned/Is-Part-Of,Table 1).

3.2 State of the Art in OO Diagrams of the System Structure

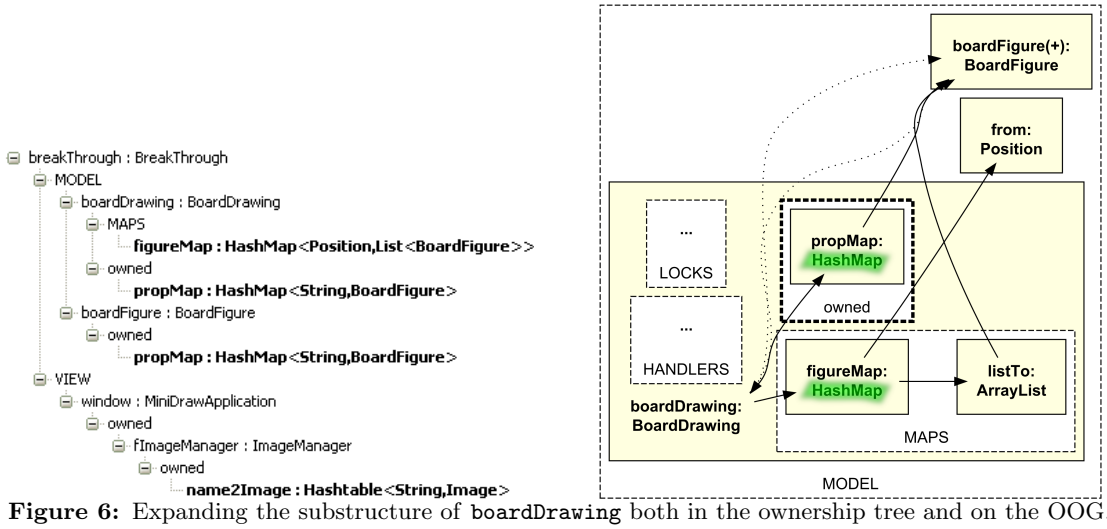


Figure 6: Expanding the substructure of `boardDrawing` both in the ownership tree and on the OOG.

There are various ways to graphically represent different aspects of software, including its structure, its execution, and its evolution. We focus on the approaches that have been proposed to describe the structure of a software system. We present how these diagrams meta models currently describe OO systems (Table 3)

and any expressiveness or scalability problems (Table 2) that they may have.

The most widely used DCS is a class diagram. A class diagram summarizes all instances of the same type as one box, e.g., `BoardFigure`, and shows one association with that type. Also, if a field is declared using an interface type, a class diagram shows an association with the interface type, e.g., an edge from `BoardFigure` to `Command`. In contrast, a DRS such as an OOG distinguishes between different instances of the same type that are created in different contexts. Also, by tracking object allocation expressions in the code, and by using a more precise lookup of types based on reachable domains, an OOG shows objects of a subset of all possible concrete types. Thus, an OOG can help developers understand the possible concrete classes that are hiding behind an interface, which is one of the difficulties in object-oriented code comprehension.

An object diagram distinguishes between different instances of the same type, but there are no tools to automatically extract object diagrams. Thus, partial views are often manually drawn (Table 3) to illustrate specific scenarios. The naive approach for extracting a DRS produces a flat object graph (Table 3), that mixes low level objects with architecturally significant objects from the application domain. For example, in MiniDraw, objects of the core type `Drawing` should appear in the `MODEL` domain. Since plain Java lacks the notion of a tier, we supply this missing design intent, using annotations (Table 3). Thus a top-level domain on the OOG represents a run-time architectural tier.

The OOG conveys architectural abstraction by ownership hierarchy using ownership domains. Domains on the OOG are not global; every object contains domains, which in turn contain objects. Thus, the OOG displays architecturally significant objects, e.g., objects of type `BoardDrawing` near the top of the hierarchy and data structures, e.g., objects of type `ArrayList` further down (Fig. 2). Also, an object on the OOG can contain multiple domains, to express design intent. For example, one container can be placed in a *private* domain of an object, while another container can be placed in a *public* domain of the same object. This way, only objects in public domains are considered part of the object's *visible state*. For example, the object `boardDrawing` contains two different `HashMap` instances, one is in a public domain, `MAPS` and the other is in a private domain, `owned` (Fig. 2).

4 Method

4.1 Experimental Design

We conducted a controlled experiment in a laboratory setting. We followed the between-subjects design [51] by having two groups **C**ontrol and **E**xperimental. The C group worked with only DCS tools, i.e., class diagrams and Eclipse while the E group was also provided with an OOG.

4.2 Selected Variables

The *independent variable* in our experiment was having access to the OOG. To measure the effect of the controlled variation of the independent variable, We used as *dependent variables* the number of code elements explored and the time spent by a participant on each task.

Research Hypothesis: Our *research hypothesis* is: for some code modification tasks that require knowledge about the run-time structure, developers who use a DRS require less comprehension effort, explore less irrelevant code, and spend less time than developers who use only DCS tools.

4.3 Hypotheses

Based on our research hypothesis, we formulate the following null hypotheses:

H10: Using a DRS does not impact the number of code elements explored by developers while performing code modifications.

H20: Using a DRS does not impact the time spent by developers while performing code modifications.

The corresponding alternative hypotheses are as follows:

H1: Developers who use a DRS explore fewer code elements (in terms of only relevant code) to answer challenging questions about the run-time structure while performing code modifications than developers who use only DCS tools.

H2: Developers who use a DRS spend less time to answer challenging questions about the run-time structure while performing code modifications than developers who use only DCS tools.

4.4 Participants

We advertised the study around the Computer Science Department at Wayne State University using flyers, mailing lists, and by word of mouth. We had 14 respondents, which we pre-screened for 1 hour each. We selected 10 participants (Table 4): 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 [1], while the median in Java experience was 4 years. All were familiar with Eclipse and UML, and all except one, with frameworks and design patterns. All of our participants passed a preliminary Java skills exam and an object-oriented programming test using Eclipse. One of them passed an advanced software engineering course discussing frameworks and design patterns.

4.5 Tools and Instrumentation

Both groups worked with Eclipse 3.4 and received an instruction sheet with 4 manually drawn, partial, class diagrams by the MiniDraw designers that explain the roles of the main classes in MiniDraw [15]. Both groups also received 6 diagrams that we reverse-engineered using AgileJ [9]. Five of these diagrams explained class relations in the five packages of MiniDraw, while the last diagram described dependencies and associations with the main class `BreakThrough`. In addition, the E group received a printed OOG. Since the OOG is hierarchical, a print out will show objects that are collapsed or expanded. So, we installed an interactive viewer of the OOG in Eclipse (Fig. 2) to allow the E group to interactively expand objects (Fig. 2-F3) or search the ownership tree (Fig. 2-F1). We used Camtasia to record the participants' think-aloud as well as a screen capture of their navigationsin Eclipse and the diagrams that they used. The study materials are available on our online appendix [1]. Table 5 shows class diagrams given to all participants. Table 7 shows the main Eclipse features with which all participants worked. Table 6 shows the features used by the E group to interactively navigate the OOG. We use the numbers in these tables later in our qualitative analysis section.

4.6 Subject System

We used MiniDraw [40], a pedagogical object-oriented framework that consists of around 15,00 lines of Java code, 31 classes and 17 interfaces. We added annotations to the MiniDraw code and extracted OOGs. We discussed the process of extracting and refining the BreakThrough OOG in Section 2.2. For the experiment, we used the BreakThrough framework application, which is a two-person game played on an 8x8 chessboard.

4.7 Task Design

For the experiment, we used the BreakThrough framework application of MiniDraw, which is a two-person game played on an 8x8 chessboard. 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 designed three tasks that serve to implement the game logic. We asked the participants to reuse the framework and implement the following features:

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.

[1] There are some inconsistencies in the table, but we recorded the numbers reported by participants.

T3 Implement the undo move feature. Add a new menu item. In the cases where the move is prohibited or cannot be undone, display a message on the status bar.

4.8 Procedure

Our experiment was in the form of a 3-hour session conducted in several settings in the same lab. The experimenter briefly introduced MiniDraw, then she tutored the participants on the basic navigation features in Eclipse. Since the concepts of OOGs and ownership are not general knowledge, She gave the E group a 20-minute tutorial explaining the OOG notation and how to interactively navigate the OOG. Both Eclipse and OOG tutorials were on arbitrary classes and methods from MiniDraw. In the remaining 2.5 hours, the participants read an instruction sheet and performed the tasks in order. Since the C group did not receive the OOG tutorial at the beginning, the experimenter spent the last 20 minutes in the C group introducing the OOG to them and asking them if it could have helped them answer some of their questions.

The participants were encouraged to plan their modifications by adding informal comments in the code. However, to avoid the artificial setting, the participants were allowed to attempt the tasks in the way that worked best for them. If they got stuck, the participants were allowed to comment out their changes and move to the next task. The participants often reasoned about the modification before they started the actual implementation, which involved mainly coding, testing, debugging, and refactoring, and tested their modifications by running the program as needed. To be able to capture their think-aloud, the experimenter prompted the participants by asking them "what are you trying to do?" Also, instead of interrupting them, the experimenter preferred to ask the participants one of the questionnaire questions whenever they struggled with the tasks.

4.9 Questionnaires and Interview

The experimenter used a recurring questionnaire between tasks (Table 9) to measure the level of comprehension in a participant. We designed the recurring questionnaire to measure level of comprehension at both high-level and details (Table 11). QX1, for example, measured top-level understanding since it asked about collaborating classes QX2 asks more about objects and their relations. QX3, which required the ability to map code elements to GUI components, required more detailed understanding Questions QX4 and QX5 aimed at determining how useful the participants perceived different sources of information to be. At the end, exit interview questions (Table 10) captured the participants' subjective feedback and any other free-form comments they had.

5 Data Analysis

We transcribed the video recordings and screen captures offline. In the transcripts (Fig. 13), we studied the developers think aloud from the video recordings, their navigations from the screen captures, and the time in the video, among others, following standard protocol analysis [51]. The transcripts included all the data that we needed, and we used the collected data to measure each of our dependent variables after the experiment.

In the transcripts, we recorded what tools or diagrams as well as the specific features the participants used to answer their questions. We use the term *information sources* to mean anything used by a participant to modify code, ranging from IDE features to available documentation to available design diagrams to source code itself. In the rest of the paper, we will use the term *strategies* to mean all the features that the participants used either in Eclipse or using the OOG viewer. We use the term *facts* to mean the information that the participants obtained from DCS tools, i.e., class diagrams and Eclipse or from the OOG to answer their questions (Table 13).

In the transcripts, we also recorded navigation targets. A *navigation target* is a code element to which a participant navigated while exploring the code, which could either be a class `B`, a field of type `A` in a class `B`, a method in a class `B`, or a local variable of type `A` declared in a method of a class `B` (Table 14). Therefore, the

code explored depends on a navigation action used by a participant. For example, if a participant opened a file using the open type or open declaration features in Eclipse, we counted all classes to which a participant navigated. Also, if a participant used the open call hierarchy feature in Eclipse on a method, he navigated from one method to another up or down in the call hierarchy. If a participant switched back and forth between files that are already open, we also record them to show what classes he ended up focusing on for a task. We also considered nodes on an OOG to be navigation targets since they represent objects in the code. Thus, on an OOG, a code element could be an object of type `A` declared in a domain `D` in an object of type `B` since the E participants ultimately traced to the field corresponding to that node (Table 15). For each activity, a *navigation path* followed by a participant is a sequence of navigation targets that lead a participant to an *outcome*, which includes finding an answer to either a question from the questionnaire or a question that the participant wanted to answer. For some of the activities, the participants made several attempts to complete those activities. In each attempt, they followed a different navigation path using a different strategy. 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. We define the completion of a certain activity as achieving a successful outcome. Tables 14 and 15 show two variations of navigated paths followed by participants in each group.

Code explored. We measured the difference between code elements explored by participants in both groups by counting all the navigation targets to which the participants navigated in each task.

Time spent. In the transcripts, we recorded the starting and ending time of each task to measure the time spent on a task, and compare the difference between groups.

It is worth mentioning that our experiment involved actual code modification as well as testing the modification. Therefore, we measured each variable while the participants thought about the modification as well as when they started the implementation. 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. Our analysis of time spent and code explored included the time a participant spent answering the questionnaire questions, thinking about the task, implementing the change, and testing the modifications. For the purposes of qualitative analysis, we also recorded the starting and ending time and the code explored of each activity . Since the participants followed several paths using different strategies to accomplish an activity, we also recorded the time spent and code explored in each path.

6 Results

In this section, we present our results with respect to the null hypotheses defined in Section 4.3. We then qualitatively discuss the results, and we briefly discuss the results of the questionnaire and the exit interview. We conclude this section by a discussion of the influence of different co-factors on the main factor.

Our results indicate that, on average, participants who used the OOG, in addition to DCS tools, i.e., class diagrams and exploring or reading the code in Eclipse, explored fewer code elements, and spent less time than participants who used only DCS tools (Fig. 7).

We analyzed our results using statistical hypothesis tests. For the code and time variables, we used the one-sided Wilcoxon Rank Sum test, since we expected a positive effect of OOGs on comprehension. Since statistical tests do not provide enough information about the practical significance [**?**], we also estimated unstandardized effect sizes in terms of raw units rather than relying on only standardized effect size, Cohen's d. To this aim, we used the mean percentage difference[2]. We also used non-parametric effect size (Cliff's delta)Cliff's delta is interpreted as the degree to which one population differs from another, i.e., the proportion of values that are higher in one sample minus the reverse proportion (i.e. 0 means no difference, +1 means all scores in one population are higher than the ones found in the control, and -1 means the opposite). Delta is a distribution-free measure of effect size and directly pertains to the distributional (non-)overlap between the samples. The CI, on the other hand, only gives the minimum and/or maximum non-overlap with 95%

[2]The mean percentage difference Mean2-Mean1*X%=Mean2

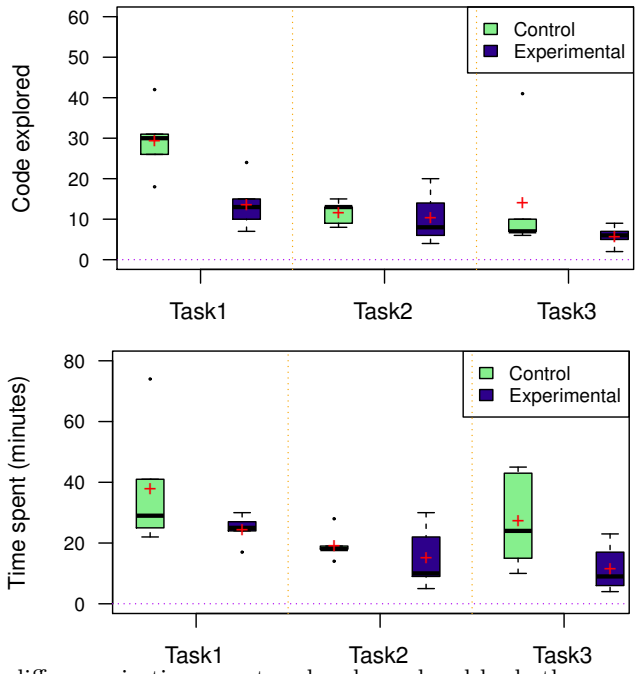


Figure 7: Box plots of the difference in time spent and code explored by both groups by task. The red (+) signs show the mean values. The plots indicate that the on average, the E group always performed better on the tasks than the C group.

probability. along with the corresponding confidence intervals with a 95% confidence level. Cliff's Delta is bound to be negative as, generally, lower values are found in the experimental groups [16].

6.1 Code explored

Our results indicate that the participants who used information about both RS and CS, i.e., OOGs, in addition to class diagrams and browsing or reading the code in Eclipse, explored fewer code elements than those who used information about only the CS.For T1, the mean reduction in code explored achieved using OOGs is 53% (Table 16). The median of the code explored using OOGs is 30, and without it, it is 13 (Table 17). The difference is statistically significant with a p-value of 0.008. The median of the code explored in T2 is 13 using OOGs, and 8 without. However, the reduction is only 10%, and the difference is not statistically significant. Using OOGs resulted in 60% reduction in T3. The median is 7 using OOGs, and 6 without, but the difference was not statistically significant. Therefore, we cannot reject the null hypothesis H20. The effect is very impressive for T1 (d=-0.92, 95%CI:[-0.99,-0.52]). For T2, the effect is small (d=-0.24, 95%CI:[-0.82,0.59]) and it is medium for T3 (d=-0.56, 95%CI:[-0.90,0.22]).

6.2 Time spent

Our results indicate that the participants who used information about both RS and CS, i.e., OOGs, in addition to class diagrams and browsing or reading the code in Eclipse, spent less time than those who used only information about CS. The mean reduction in time spent achieved using OOGs is 36% for T1, 22% for T2, and 60% for T3(Table17). In T1, the median time spent using OOGs is 29 min, and 25 min without. In T2, the median is 18 min using OOGs, and 10 min without. In T3, the median is 24 min using OOGs, and 9 min without. The effect size is medium for T1 (d=-0.4,95%CI:[-0.85,0.4]), T2 (d=-0.28,95%CI:[-0.85,0.59]), and T3 (d=-0.68,95%CI:[-0.94,0.10]). Not all the differences are statistically significant, so we cannot reject the null hypothesis H20.

Table 1: Facts about the run-time structure provided by an OOG. Fn refers to a feature in Fig. 2

Fact	**How to use the OOG**
Is-In-Tier	A developer can look for a top-level domain that corresponds to a run-time tier (e.g. `MODEL`), then pick an object object of type `A`(e.g. `from:Position`) in that domain (Fig. 2)
May-Alias	An object on the OOG may represent more than one run-time object. A developer can pick an object of type `A`, e.g., `Position` and trace to all possible `new Position()` expressions (Fig. 2-F2)
Points-To	A developer can explore all incoming points-to edges (solid arrows, Fig. 2) to an object of type `A`. If an edge is lifted (dotted arrows, Fig. 2), he can expand the object to identify a solid edge
Is-Owned Is-Part-Of	If an object is not in top-level domains, a developer can search in different domains in the ownership tree (Fig. 2-F1) or expand an object of type `B`, e.g. `boardDrawing` (Fig. 2-F3) looking in different domains for different objects of type `A` that are strictly encapsulated or logically contained in that object

Table 2: Problems in current approaches/tools/diagrams used to visualize the structure of OO programs.

Diagram Tool Approach	**Global [vs. Elided/ task-specific]**	**Hierarchical [vs. Flat]**	**Instances [vs. Types/Layers]**	**Sound [vs. Unsound]**	**Concrete [vs. Abstract]**	**Traceability to Code**	**Uses annotations**
Automatically extracted CD		N/A			X	X	
Manually drawn CD		N/A					
Manually drawn OD			X				
Statically extracted OG			X	X	X	X	
Dynamically extracted OG			X	X	X	X	
Scholia (OOG)	X	X	X	X	X	X	X

Table 3: State of the art in diagrams of OO system structure.

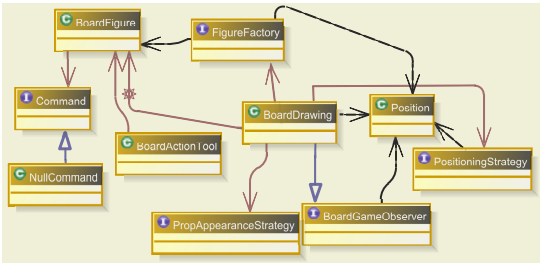
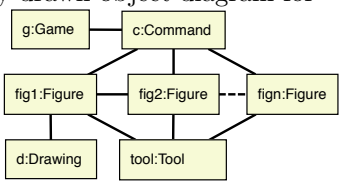
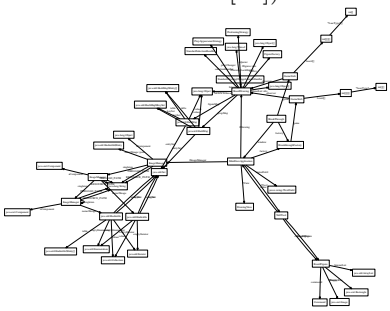
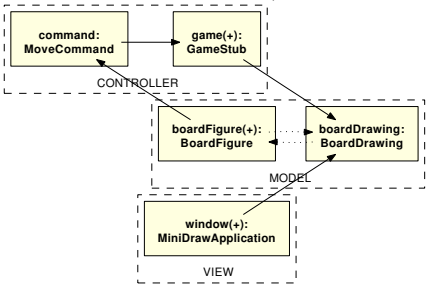
Partial automatically extracted class diagram [9]

Manually drawn object diagram for MiniDraw 
Flat object graph of MiniDraw using Womble [29]) 
Hierarchical object graph (Partial OOG for MiniDraw) 

Table 4: Participants' self-reported experience. Familiarity with Eclipse is on a Likert scale: 1 (beginner) to 5 (expert).

P	Prog. Exp.	Ind. Exp.	Yrs. Java	Yrs. C#	Yrs. C++	Eclipse
C1	4	0 (Ph.D.)	5	2	4	3
C2	20	6 (Ph.D.)	8	2	11	5
C3	9	4 (M.Sc.)	2	3	7	3
C4	4	0 (Ph.D.)	4	≤ 1	4	3
C5	6	0 (B.S.)	3	0	4	3
E1	3	2 (M.Sc.)	1	0	6	3
E2	8	0.5 (Ph.D.)	4	1	2	5
E3	25	20 (M.Sc.)	5	4	15	5
E4	24	20 (Ph.D.)	10	2	0	5
E5	10	2 (B.S.)	3	3	5	3

Table 5: 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 6: 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 7: 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	

Table 8: Experimental procedure (3-hour session).

Part I (25 min)	**I**ntroduction	2 min
	Eclipse brief tutorial	3 min
	OOG Tutorial	20 min
Part II (2.5 hours)	Performing tasks and answering questionnaire	90 min
Part III (5 min)	Exit Interview	5 min

Table 9: Recurring questionnaire between tasks. X refers to a task.

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

Table 10: Exit interview questions to capture subjective evaluation. I2 and I3 were asked to only the E group.

No.	Question
I1	Do you think the tasks were hard or easy?
I2	Do you think the OOG is useful?
I3	Is the OOG more useful than a class diagram or complementary?
I4	Do you think your change respected the design?
I5	Is there anything else you would like to add?

Table 11: Classification of questionnaires according to the level of detail.

No.	Level of detail
QX.1	High-level (conceptual)
QX.2	High-level and Detailed (technical)
QX.3	High-level and Detailed (technical)

Table 12: Strategies used by participants with and without OOGs.

Question	Strategy	Fact used
	Eclipse	
Which-Tier-Has-A	package explorer	Is-In-Layer
How-To-Get-A	Java Search (Type references, scope: project)	Has-A, References
	file search	Has-Label
How-To-Get-A-In-B	code assist	Visibilty
	find/replace (scope: file/class)	Has-A, Has-Label
Which-A-In-B	reading code	Has-Label
	Java doc	Has-Label
How-To-Get-A	Call hierarchy	Control flow
All	Type Hierarchy	Is-A
All	Java Search (Type declarations, scope: project)	Has-A, References
	Diagrams	
CD		
How-To-Get-A-In-B	association	Has-A
How-To-Get-A-In-B	dependency	Control flow
All	implements	Is-A
	extends	Is-A
How-To-Get-A	aggregation	Has-A
OOG		
Which-Tier-Has-A, How-To-Get-A	Explore objects in top-level domains	May-Alias
How-To-Get-A, How-To-Get-A-In-B, Which-A-In-B	explore/search ownership tree	Is-Owned, Is-part-of
How-To-Get-A-In-B	explore incoming/outgoing edges	Points-to
How-To-Get-A,How-To-Get-A-In-B,Which-A-In-B	expand/collapse objects (graph)	Is-Owned, Is-part-of
All	labeling type	Is-A
All	trace to code	N/A

Table 13: Strategies followed by participants to answer questions.

Strategy	How-To-Get-A	How-To-Get-A-In-B	Which-Tier-Has-A	Which-A-In-B	Fact used
Eclipse					
Package Explorer			X		Is-In-Layer
Java Search (Type References, scope: project)	X				Has-A
File Search	X				Has-Label
Java Search (Type Declarations, scope: project)					N/A
Code Assist (scope: file/class)		X			Visibilty
Find/Replace (scope: file/class)		X		X	N/A
Reading Code (scope: file/class)	X	X	X	X	N/A
Java Doc (scope: file/class)				X	N/A
Call Hierarchy		X			Control flow
Type Hierarchy	X	X		X	Is-A
Diagrams					
OOG					
Explore objects in top-level domains	X		X		N/A
expand/collapse objects (graph, tree)	X				Is-Owned, Is-part-of
explore incoming/outgoing edges (graph, tree)		X			Points-to
search ownership tree (objects, edges)	X	X			Points-to
trace to code	X			X	N/A
labeling type	X	X		X	Is-A
CD					
association		X			Has-A
aggregation	X				Has-A
dependency		X			Control flow
implements	X	X		X	Is-A
extends	X	X		X	Is-A

Table 14: Two navigation paths followed in answering a question without OOGs.

path	Navigation Target (Class/Method/field/ local variable)	Outcome

1	`BoardFigure`
	`PositioningStrategy.CalculateFigureCoordinates()`
	`AbstractFigure`
		Failure

2	`FigureFactory`
	`BreakThroughPieceFactory`
	`BreakThroughPieceFactory.generatePieceMultiMap()`
	`figureMap`
		Success

Table 15: Two attempts (navigation paths) followed in answering a comprehension question using OOGs.

path	Navigation Target (Class/Method/OOG node (triplet))	Outcome

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

2	≺`MoveCommand,CONTROLLER,BreakThrough`≻ ≺`BoardDrawing,MODEL,BreakThrough`≻ ≺`ArrayList,owned,BoardDrawing`≻ ≺`HashMap,MAPS,BoardDrawing`≻	Success

Table 16: Descriptive statistics of the code elements explored and results of wilcox test and non-parametric effect size.

Task	C			E			p-value	mean percent. difference	Cliff's delta
	Mean	Median	SD	Mean	Median	SD			
T1	29.40	30.00	8.71	13.80	13.00	6.46	**0.008**	53% reduction	-0.92
T2	11.60	13.00	2.97	10.40	8.00	6.54	0.264	10% reduction	-0.24
T3	14.20	7.00	15.06	5.80	6.00	2.59	0.068	60% reduction	-0.56

Table 17: Descriptive statistics of the time spent and results of wilcox test and non-parametric effect size.

Task	C			E			p-value	mean percent. difference	Cliff's delta
	Mean	Median	SD	Mean	Median	SD			
T1	38.20	29.00	21.28	24.60	25.00	4.83	0.147	36% reduction	-0.4
T2	19.40	18.00	5.18	15.20	10.00	10.43	0.232	22% reduction	-0.28
T3	27.40	24.00	15.98	11.80	9.00	7.98	**0.0476**	60% reduction	-0.68

Table 18: Code understanding activities attempted by all participants in each task. The code definitions are in Table 19. The participants did not attempt the activities in the same order, but for simplicity, we list them in the order specified. Some participants deferred T1.d and T1.e to the later tasks.

Task	Activity	Question	run-time structure related question.
T1	T1.a	In which class shall I implement the validation logic?	Which-Tier-Has-A
	T1.b	Where is the data structure (of type A) representing the game board?	Which-A-In-B
	T1.c	How can I get an instance of this data structure of type A inside class B?	How-To-Get-A-In-B
	T1.d	Where is the object that is responsible for showing the status message?	How-To-Get-A
	T1.e	How can I get that object of type A inside the class B that is responsible for validating the movement?	How-To-Get-A-In-B
T2	T2.a	In which class shall I implement the capture?	Which-Tier-Has-A
	T2.b	Which object represents a piece so I can compare it to an opponent piece?	How-To-Get-A
	T2.c	How can I get that object inside the class responsible for handling captures?	How-To-Get-A-In-B
	T2.d	How can I remove a piece from the game board? Which object shall I use?	How-To-Get-A-In-B
T3	T3.a	In which class B shall I add the menu bar?	Which-Tier-Has-A
	T3.b	In which class shall I implement the undo logic?	Which-Tier-Has-A
	T3.c	How can I get the objects that handle the movements and the captures inside those classes?	How-To-Get-A-In-B

6.3 Qualitative Analysis of Activities, Questions, and Strategies

We further broke down tasks into activities, following a hierarchical task decomposition [17]. At a high-level, our participants attempted three feature implementation tasks. All participants divided their tasks into smaller *activities*, which ranged from code understanding to GUI testing to code modification and reuse to refactoring to debugging. For the purposes of qualitative analysis, we focus on the thought process, so we analyzed activities that we thought are interesting, including activities related to answering the questionnaire (QX.1, QX.2, and QX.3, Table 9), code understanding (e.g. understanding how direct manipulation occurs), code modification (e.g. accessing a field in a class), and debugging (e.g., fixing a run-time exception). In each activity, participants required information, which they knew by experience or had learned from exploring the code or did not know and had to search for. Thus, all participants did not attempt the same activities, but they all engaged in similar understanding activities, which they either documented as comments in the code or expressed in their think-aloud. We refer to the activities of a task Tn as Tn.a,...,Tn.z (Table 18). Activities T1.a, T2.a,T3.a and T3.b correspond to QX.1 and activities T1.b, T1.d, and T2.b correspond to QX.3 (Table 9).

The participants expressed their need for information as *questions*. We ignored questions that were not specifically about the run-time structure, e.g., *Why do they have tool? What tool is for?* We identified four main questions about the run-time structure: How-To-Get-A, How-To-Get-A-In-B, Which-Tier-Has-A, and Which-A-In-B (Table 19). For each activity, we coded the questions about the run-time structure involved in it using our classification (Table 18). For example, T1.b. involved questions like *how can I get an object of type* `BoardFigure`*?* (How-To-Get-A), while T1.c involved a question like *How can I get a* `BoardDrawing` *object in* `GameStub` *so I can get the* `figureMap` *object?* (How-To-Get-A-In-B). The activities did not necessarily require the exact same questions. For example, in T1.b, some participants searched for

how can I get a `Position` object? while others searched for *How can I get `Map<Position,BoardFigure>`?* Some activities required information about types but the main activity was concerning objects. For example, T1.c involved a question like *How can I get `BoardDrawing` in `MoveCommand` so I can get `figureMap`?* Such a question required investigating further the type hierarchy of a field of type `BoardGameObserver` to investigate possible concrete types.

> *...Ok. `HashMap` is here [`BoardDrawing`] that's what we're looking for. I want to get the `figureMap`. Why isn't it in `Game`?...*(E2,T1.c)

Each question triggered asking more questions. (Fig. 8 and 9), so we also counted all the questions involved in answering the original question each under its category (Table 26). For example, to determine inside which class he should validate the movement, one participant wanted to answer a Which-Tier-Has-A question which then required him to answer a How-To-Get-A-In-B question:

> *...Ill put this `editor.showStatus()` line to make sure [that this is the right place]. The object is not available! So the `editor` object is there [in some class C] but I need to find it...*

For example, to answer a How-To-Get-A-In-B question, the participants first sought an answer to a How-To-Get-A question looking in different classes `C1,...,Cn` for an instance of type `A`. Then, they investigated whether they could access an instance of any of the types `C1,...,Cn` inside `B` so they could access the desired instance of type `A`:

> *... so i will call the create status field method to setup the message... it's a `MinidrawApplication` so ... I need to know where this object is instantiated*

> *...if I access the object of `MinidrawApplication` and call the function `showStatus()` it will get displayed. So, for that I need to find who is using this `MiniDrawApplication`. In which class?...(C3,T1.d)*

Also, to access an object of type `A`, some participants wanted to know in which tier instances of type `A` are created (Which-Tier-Has-A):

> *...what I'm trying to do is find the UI part of the code where I can add it [menu bar]. `MinidrawaApplciation` so its probably this guy okay...*

> *...I'm wondering if I can access it [figureMap] from here [GameStub]because this is going to call again later all this drawing stuff. I think this [BoardDrawing] is the graphical representation where this [GameStub] is more like the logic of it...(C2,T1.c)*

For some activities, e.g. T1.b, the participants struggled with distinguishing between different instances of the same type that are in the same class (Which-A-In-B):

> *...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...(C5,T1.b)*

Both groups switched between Eclipse and diagrams (Table 36), but the *strategies* that they used to answer questions about the run-time structure varied based on the source of information. Each strategy produced a navigation *path*, i.e, a sequence of code elements to which a participant navigated.

More formally, a path can be defined as a starting object of type Tsrc, a subpath, and a target object of type Tdst. Since we coded the participants questions using our coding model, we define a path using

Table 19: Classification of questions about the run-time structure.

General form of questions asked by a participant	**Related question about the run-time structure**
In which class of type `A` shall I implement the task?	Which-Tier-Has-A
I know the type `A` is related to this task, but I don't know where an instance of type `A` is created.	How-To-Get-A
How can I access an instance of type `A` in class `B`	How-To-Get-A-In-B
I'm in class `B` and it has many instances of type `A`, how can I distinguish between them.	Which-A-In-B

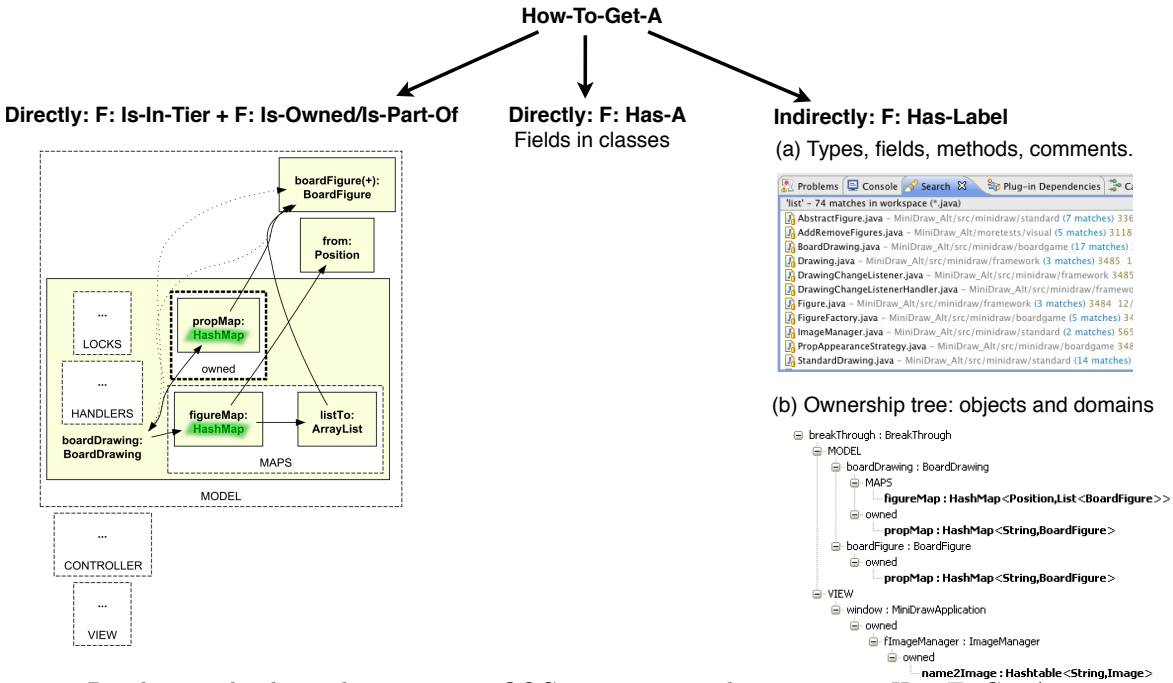


Figure 8: Developers who do not have access to OOGs approximate the answer to a How-To-Get-A question using time consuming strategies (understanding inheritance relations of type A of a field) or imprecise strategies (searching for a concept that has a label). On the OOG, the developer knows that each node corresponds to a run-time object. Developers can rely on the fact that object of type A could be created in a run-time tier or the fact that an object of type A could participate in building the substructure of an object of type B. Even if a developer relies on a concept that has a certain label, he searches a tree of only objects and domains on the OOG.

Table 20: Participants referred to OOGs and class diagrams frequently regardless of their experience level. E participants referred mainly to the OOG while C participants referred to class diagrams.

Task	Source	C1	C2	C3	C4	C5	E1	E2	E3	E4	E5
T1	Eclipse	2	7	5	4	13	2	4	7	6	3
	Diagrams	5	11	16	3	18	1	8	8	9	5
T2	Eclipse	3	2	1	3	3	3	5	4	2	4
	Diagrams	2	3	0	6	2	4	4	3	1	8
T3	Eclipse	2	4	2	2	2	3	3	4	1	3
	Diagrams	3	11	1	4	0	4	2	2	2	5
Total	Eclipse	7	13	8	9	18	8	12	15	9	10
	Diagrams	10	25	17	13	20	9	14	13	12	18

the types A and B to describe the path followed to answer each question (Table 23). Thus, a path starts with a type B and ends with the desired object of type A, i.e.,[B, sub path, A]. Each path is determined by the question that a participant wanted to answer (Table 23). For example, to perform the task T1, the participants had to answer the question *"How can I get the data structure representing the chess board?"* (**How-To-Get-A**, T1.b, Table 18). They often followed a path with an object of type `HashMap` or `ArrayList` as the target (A). On the OOG, the E participants either picked the root object `system:BreakThrough` as a starting point (B) which resulted in the path `[BreakThrough, sub path, Map]` , or picked a related object from the application domain such as `p:Position` and started from that object which resulted in a path `[Position,sub path,Map]`. Another question the participants needed to answer in task T1 was *"I'm in class `GameStub` (B) and it does not have the `figureMap` (A) directly.*(**How-To-Get-A-In-B**,T1.c, Table 18). On the OOG, the E participants explored all incoming edges of any object of type A then investigated one of the source objects of type B. For example, they knew that they could access `figureMap` from `boardDrawing`, but they needed to know if they can access `boardDrawing` in `GameStub`. They followed the path `[GameStub,`

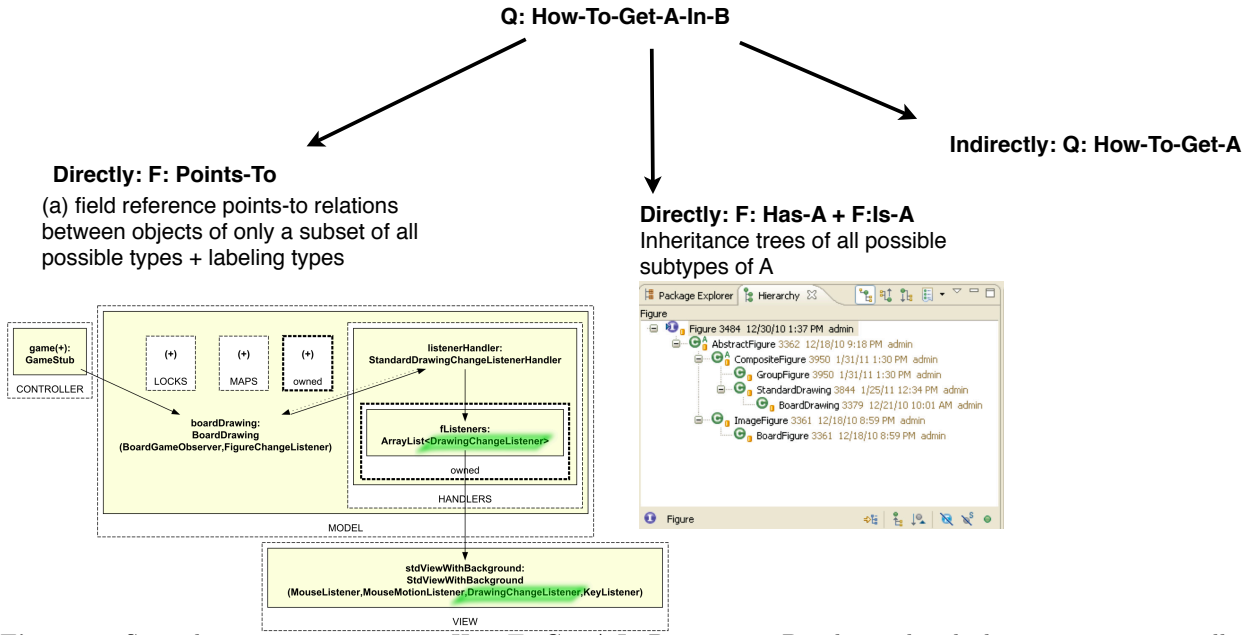


Figure 9: Several strategies to answer a How-To-Get-A-In-B question. Developers break this question into smaller questions to be able to answer it. For example, they first answer a How-To-Get-A looking for a class C that has an instance of type A. Then again, they try to figure out how they can get C in B which requires repeating the process until they are finally able to answer the original question.

`sub path,Map]`. In the code or on a class diagram, the C participants followed the same path, but the sub path involved investigating several concrete types of the super type `BoardGameObserver`. Some questions required the participants to try different strategies each of which produced a different path until they finally found a successful path that either answered their question or verified their assumption. As a result, the participants followed one or more paths to answer a single question. We measured the code explored and time spent in both successful and unsuccessful paths in each activity.

If a participant found the answer to a question, he provided a comment in the code to which he returned later to provide the implementation:

```
// get the figureMap object from BoardDrawing object which I can access from
BreakThrough class
```

Two E participants and one C participant implemented and demonstrated the three tasks. Even the participants who did not demonstrate their implementation for all three tasks, attempted most of the understanding activities in each task (Table 21). Some of them successfully completed those activities and tested their implementation, but they encountered bugs that prohibited them from proceeding. Other participants completed the activities and provided precise comments to indicate how they would have done the implementation had they had enough time:

```
// remove previous piece
//boardDrawing.getFigureMap.getKey(to).isEmpty(),
get list, remove(0);
```

There was some degree of variability in the code explored and time spent in the attempted activities (Table 21). In some cases, both variables were proportional, but in many cases, the difference in one variable was either greater or less than the other. To investigate possible causes of difference, we discuss the *questions* raised by participants in each activity, and the *strategies* used to answer those questions.

Table 21: Code explored and time spent in the attempted activities in each task. Some participants did not have enough time to attempt some of the activities. Some participants attempted all the activities, but did not succeed in running the application.

Activity	code explored										time spent (min)									
	C1	**C2**	**C3**	**C4**	**C5**	**E1**	**E2**	**E3**	**E4**	**E5**	**C1**	**C2**	**C3**	**C4**	**C5**	**E1**	**E2**	**E3**	**E4**	**E5**
T1.a	**11**	**11**	**16**	6	**19**	3	3	3	3	3	10	6	**11**	5	**27**	6	10	2	1	4
T1.b	10	4	2	6	15	18	1	3	3	2	**12**	2	2	7	**36**	**18**	4	4	7	6
T1.c	9	3	1	6	2	1	2	3	4	2	**19**	4	1	10	5	1	5	6	9	**20**
T1.d	-	2	2	-	5	2	6	2	-	-	-	2	4	-	5	2	10	1	-	-
T1.e	-	11	5	-	1	-	4	4	-	-	-	**15**	4	-	1	-	6	11	-	-
T2.a	8	5	1	1	1	1	**15**	1	1	1	8	5	1	1	1	1	**14**	1	1	1
T2.b	-	6	4	7	6	**11**	2	1	6	2	-	**11**	5	8	**11**	**22**	2	5	7	3
T2.c	1	1	2	4	1	1	2	1	1	1	1	1	6	9	1	1	4	1	1	1
T2.d	6	1	1	1	1	1	1	3	1	1	**19**	1	2	1	5	6	2	3	2	1
T3.a	4	1	2	7	3	2	5	3	3	5	4	1	2	7	3	**17**	7	1	2	**14**
T3.b	3	34	2	-	2	-	1	2	1	3	**20**	**35**	5	-	**12**	-	1	4	1	8
T3.c	3	6	3	-	1	-	1	1	1	1	**20**	7	6	-	1	-	1	1	1	1

Table 22: Participants followed navigation paths in which they always searched for an object of type A or B. We use an object of type **C** to refer to any intermediate object(s) that they needed before reaching the desired object of type `A`.

Question	**Desired object of type...**	**Path**	**Sub path using the OOG**	**Sub path using class diagrams and Eclipse**
How-To-Get-A	`A`	`[B, sub path, A]`	involves exploring top-level domains for an instance of type `A` or expanding/collapsing several instances of types `B1,...,Bn` looking for an instance of type `A`	involves searching through several classes `B1...Bn` for an instance of type `A`
How-To-Get-A-In-B	`A` through an object of type `C`	`[B, sub path, A]` involves two paths: `[A, sub path, C]` `[B, sub path, C]`	involves exploring incoming edges from objects of types `C1...Cn` to an object of type `A`, then exploring all outgoing edges of an object of type `B` to any of the objects of types `C1...Cn`	involves exploring call hierarchies of methods on several calling classes `C1...Cn` and type hierarchies of a field of type `A` if `A` is an interface type

6.4 Theory Revisited

In this section, we provide observations backed by evidence from the participants navigations and think-aloud to support our theory. To answer questions about the run-time structure, the E group used facts on the OOG that directly answered their questions (Table 24). The C group relied on facts about the code structure from class diagrams combined with facts obtained from Eclipse features, which answered their questions only partially (Table 25). We provide two observations supported by quantitative data:

Participants who did not use the OOG struggled more with questions about the run-time structure. For each question type, we counted the concrete questions that a participant asked in each task and totals in each group (Table 26). We observed that some questions arose more often than others. For example, How-To-Get-A and How-To-Get-A-In-B were asked most frequently, but more often in the C group (100 times and 58 times, respectively) than in the E group (48 times and 19, respectively). The C group benefited from class diagrams to identify the classes that they needed for a task, but they still had to search for where an instance of a class is created:

> *...I'll use this one [a class diagram] I'll go to the `Position` class first, so this class...I think an object will be created when you make a move...(C3,T1.a)*

The C group alternated strategies in Eclipse and refined their original questions until they found an answer. The E group wondered about these questions only when they worked for a while in Eclipse then struggled

Table 23: Paths followed using the OOG tool compared to paths using DCS tools to answer questions about the run-time structure.

Question	Path followed using the OOG involved...	Path followed using DCS tools involved...
How-To-Get-A	exploring top-level domains for an object of type `A` or expanding/-collapsing objects of types `B1,...,Bn` looking for an object of type `A`	searching through several classes `B1...Bn` for an instance of type `A`
How-To-Get-A-In-B	exploring incoming edges from objects of types `C1...Cn` to an object of type `A`, then exploring all outgoing edges of an object of type `B` to any of the objects of types `C1,...,Cn`	exploring call hierarchies of methods on several calling classes `C1...Cn` or exploring type hierarchies of a field of type `A` if `A` is an interface type
Which-A-In-B	searching in different domains in the ownership tree or expand an object of type `B` looking in different domains for different objects of type`A` that are strictly encapsulated or logically contained in that object	reading a class B to distinguish between different instances of type A.
Which-Tier-Has-A	looking for a top-level domain that corresponds to a run-time tier, then pick an object of type A in that domain and trace to `new A()`	exploring the package structure looking for an application specific layer or reading class methods or the JavaDoc trying to guess the tier to which a class belongs.

Table 24: Facts from the OOG used by the E group.

Question	Fact used to answer the question
Which-Tier-Has-A	Is-In-Tier: look inside the root instance for related run-time tier, then pick an instance of type A in that tier and trace to `new A()`
How-To-Get-A	pick an instance of type `A` in that tier and trace to `new A()`
How-To-Get-A-In-B	Points-To:explore all incoming points-to edges to an instance of type `A`
Which-A-In-B	Is-Owned/Is-Part-Of: expand an instance of type `B` looking for different instances of type `A` that are strictly encapsulated or logically contained in an instance of type `B`

with a question, which they directly answered by referring back to the OOG.

The E Participants successfully used the OOG to gain both high-level and detailed understanding. Our results indicate that the E group relied on different facts on the OOG based on the level of detail in each activity (Table 27). The C participants relied on class diagrams and package structure in Eclipse to gain high-level understanding. However, to gain detailed understanding, they had to dig into the code or use

Table 25: Facts from DCS tools used by the C group.

Question	Fact used to answer the question
Which-Tier-Has-A	Is-In-Layer: explore the package structure, but instances of the same type can be created in different packages, e.g., `java.util.ArrayList`. Also, one package can contain types from different tiers.)
How-To-Get-A	Has-Label: file search (many hits in comments) Has-A: Java search (time consuming)
How-To-Get-A-In-B	Is-Visible: code assist (wrong assumptions when declared type is an interface or field is private) Is-A: Type hierarchy (time consuming) Control flow: Call hierarchy (time consuming)
Which-A-In-B	Read the code (time consuming) Has-Label: Java doc (time consuming)

Table 26: Frequency of questions about the run-time structure.

Question	C1	C2	C3	C4	C5	E1	E2	E3	E4	E5	Total C	Total E
How-To-Get-A	15	54	31	9	26	8	9	7	18	10	135	48
How-To-Get-A-In-B	8	34	16	3	14	2	3	6	4	4	75	19
Which-A-In-B	2	1	1	1	1	1	1	1	2	1	6	6
Which-Tier-Has-A	3	8	2	2	2	2	2	2	2	2	17	10

Table 27: Classification of activities according to the level of detail and Facts used to answer each question.

Task	Activity	level of detail
T1	T1.a	High-level
	T1.b	Detail
	T1.c	Detail
	T1.d	Detail
	T1.e	Detail
T2	T2.a	High-level
	T2.b	Detail
	T2.c	Detail
	T2.d	Detail
T3	T3.a	High-level
	T3.b	Detail
	T3.c	Detail

Eclipse exploration features.

As for the questions involved in activities, we also grouped them into High-level (conceptual) and detailed (technical) questions (Table 27). Some activities triggered using diagrams more than others based on the level of detail (Table 36). In the E group, the OOG enabled the participants to answer both high-level and detailed questions easily.

Participants who did not use the OOG approximated answers to questions about the run-time structure using alternative strategies. The strategies that the participants used in the absence of OOGs made the process of retrieving information about objects and their relations either time consuming, or imprecise and misleading, or tedious and distracting. We classified strategies used by participants based on our coding model (Table 13) to compare between different strategies followed by participants in both groups.

Time consuming strategies: Since some activities were more time consuming to the participants than others (Table 21), we discuss some of the longest code *understanding* (Table 32) and *modification* activities (Table 33) to investigate possible causes of difference. In general, the C group relied heavily on Eclipse features and class diagrams, which provided them with information about only the code structure (Table 29). Some Eclipse features were not specialized enough to answer questions about the run-time structure immediately (e.g. file search) while others provided only information about the code structure (e.g. type hierarchy). The E group, on the other hand, relied on facts from the OOG (Table 28) which answered their questions more easily.

During *understanding* activities (Table 32), the participants had to decide where to implement the task, and for that, they needed to answer a Which-Tier-Has-A question. The C group relied on either Is-In-Layer facts, reading code, or Java doc comments (Table 30). One C participant used debugging and GUI testing to verify a simple assumption. For example, to locate where to implement the change, one participant added a print statement in a method body to check if this is the method being called when a piece is moved. He ran the application in order to verify his assumption, which required 11 minutes (Table 32). . During understanding activities, the most time consuming strategies in the C group were the different search

mechanisms in Eclipse (Has-Label, Has-A) and investigating type hierarchies (Is-A), and the C group relied on these facts more than the E group (Tables 29 and 28).

The participants who used Has-Label facts preferred to identify an object based on some string. The C group had to search the whole code base including comments. For example, in T1.b, they searched for either a container object, e.g., `list`, or a contained element, e.g., `piece`.

> *...I'll search for `list`. It's probably not the most efficient way; so many lists so let's search `piece`...(C1,T1.b)*

The E group did not rely much on Eclipse search to answer the question in T1.b, since the OOG displayed both the container and the contained elements as distinct objects each in its own domain. Also, they searched a tree of only objects and domains (F1, Fig. 2).

Searching for Is-A facts was necessary since the participants were asked to modify a specific application of MiniDraw, so they were interested in concrete types, especially if a field was declared in one class using an interface type or if it gets passed as a method parameter. The C group had to further explore inheritance relations in Eclipse of all possible concrete types of a field then filter out the desired type. For example, in T1, all participants wondered what the concrete type of the field `BoardGameObserver observer` in `GameStub` would be and used Is-A facts for that purpose. The E group benefited from the labeling types on objects while the C group used Eclipse type hierarchy. However, in some cases, exploring all possible concrete types of fields was unnecessary:

> *...now we have to find out where the other tools are being used and what resources they have...* **10 minutes later** *...so thats like a centralized location, which is what I thought about initially instead of wandering all that nonsense!...*(C2,T3)

> *...so Im in `BoardFigure` which extends `ImageFigure` which extends God knows what...*(E4,T2)

For example, in T2 and T3, the participants needed to identify the concrete type of a field of type `Figure` in `BoardActionTool`. The C group investigated 7 possible subtypes of `Figure` compared to the E group who identified a points-to relation from object `boardActionTool` to object `boardFigure` (Fig. 2).

The participants needed Is-A facts since they were asked to modify a specific application of MiniDraw, so they were interested in concrete types. Therefore, some of the run-time structure related questions required the participants to understand what a specific concrete type of a field would be:

> *...since `BreakThroughFactory` is a specific implementation of the `Factory` interface, it goes to a specific factory, whereas `GameStub` is not breakthrough specific; its just another very general game...*(C5,T1)

The E group explored objects of only a subset of the concrete types on the OOG.

Even after spending some time on code understanding activities, the participants struggled with questions about the run-time structure during implementation activities (Table 33), especially when they encountered a run-time exception which required answering the question *where an object of type A is created so I don't have to recreate it in this class?* or a compilation error due to a wrong assumption: *I thought I could access a field of type A in B, but the code assist does not show it!*

For example, to implement T1, C4 created a new instance of type `HashMap<ArrayList<BoardFigure,Position>` inside `MoveCommand`. While testing his modification, he got a run-time exception, so he used the debugger and found that he should search for a class in which a `HashMap` instance is initially created instead of recreating it. He searched for `new HashMap`, which produced several hits inside `BoardDrawing`, so he spent some time reading this class to distinguish between these instances. Then, he struggled with getting an instance of the class `BoardDrawing` in `MoveCommand` to be able to access the desired `HashMap` instance. He searched for `drawing` but founds so many hits and asked for help:

> *Is there a quick way to find out where there are people creating that `BoardDrawing` object?...*(C4,T1)

Finally, he resorted to refactoring techniques to be able to access the desired objects. This activity took him 20 minutes (not shown in Table 21). The E group, on the other hand, referred back to the OOG, which enabled them to identify where an object is created, e.g., `figureMap` inside `BoardDrawing`, how objects are connected through field reference points-to relations (Points-To), e.g., `moveCommand` points-to `boardDrawing`,

and what role each instance of the same type is playing (Is-Owned/Is-Part-Of), e.g., two `HashMap` instances in two different domains inside `boardDrawing` (Fig. 2).

Some E participants spent long time on some activities, so we explain why this was the case. For example, E1 spent around 30 minutes in Eclipse without referring to the OOG until he got stuck in activity T1.d, where he tried to answer a Which-Tier-Has-A question. The OOG helped him identify all objects created in the `VIEW` domain, and to trace to the corresponding instance creation expression. In later tasks, this participant referred to the OOG more often (Table 36) and benefited from facts on the OOG. Similarly, E2 thought that he can just dig in Eclipse then refer to the OOG whenever he struggled with a question about the run-time structure:

> *...okay so usually I start from the code I get a high level understanding of the program and what it does then I go back to a diagram like this [OOG] and it is more helpful...*

In T2.a, this participant wanted to understand how the Command design pattern is used since he thought he should implement the capture as another command. He spent 7 minutes trying to understand and implement the CaptureCommand. The code did not support his implementation since in MiniDraw, when each BoardFigure object is created, it is assigned a MoveCommand. He thought that he should understand how the tools work in MiniDraw, so he spent another 7 minutes on that activity.

Tedious and distracting strategies: Some participants expressed their frustration from Eclipse features, such as file search and JavaDoc comments, or from missing details in class diagrams, such as the type of a container object:

> *...at this point Im going to read what the guy put in front of me [Javadoc], so this tells me that this guy crossed lines and Im not going to be able to find stuff... Im very methodical and the right thing has to go in the right spot and if there is no right spot I'll consider creating a right spot... This is the boardgame [package] so Im assuming we implemented this so if we implemented this.. so at this point I just want to know what the person who implemented the class meant by it..Ok. so that does not do me any good sorry!...*(E4,T1)

> *...whos calling `editor`? [using file search for "editor]...very convoluted here!...*(C2,T1)

Imprecise and misleading strategies: Participants in both groups found Is-In-Tier facts to be most useful to determine where to implement the change. Even though tiers were not visible in the code, the C participants could still conclude in which tier instances of a class are created by reading the class description:

> *...So right now the bar is not there we need to check... I think `DrawingEditor` is related to that but it is the interface...*

The C participants investigated how classes are organized into modules (Is-In-Layer) to locate where to implement a change. For example, one participant wanted to rely on an Is-In-Layer fact:

> *I'm only making undo for the breakthrough game, so I dont want to add this toolbar to every board game type. I want to add it to only a breakthrough type. I would need that probably in the breakthrough package. It only has these four files `Breakthrough`, `BreakThroughPieceFactory`, `Constants`, `ChessBoardPositioningStrategy`...*(C5,T3)

Then he read through the class `MiniDrawApplication`, and decided to rely on an Is-In-Tier fact instead, but he was still confused:

> *...I feel like i would attach it to the DrawingEditor window. Actually Im going to go into MiniDrawApplication since it extends JFrame I think Im probably close to the right area. The only problem is that this is in the `minidraw.standard` package, so this would affect a whole lot more than just breakthrough...*

Table 28: Facts on the OOG used by the E group to answer questions about the run-time structure.Details on how frequently the participants used Eclipse and OOG features are in Tables 30 and 31, respectively.

Fact from the OOG	Used to answer ...	E1	E2	E3	E4	E5	Total
Is-A: Labeling type	How-To-Get-A-In-B	5	2	5	4	3	19
Has-A: Reading code	How-To-Get-A-In-B	2	2	9	5	1	19
Has-Label: Search ownership tree	How-To-Get-A	2	1	0	0	1	4
Is-In-Tier: explore top-level domains	Which-Tier-Has-A How-To-Get-A How-To-Get-A-In-B	5	2	4	9	6	26
Points-To:Explore incoming outgoing edges	How-To-Get-A-In-B	6	4	9	12	10	41
Is-Owned/Is-Part-Of: expand collapse objects	How-To-Get-A How-To-Get-A-In-B Which-A-In-B	1	2	4	4	7	18

Table 29: Facts From DCS tools used by the C group to answer questions about the run-time structure.Details on how frequently the participants used Eclipse and OOG features are in Tables 30 and 31, respectively.

Fact from DCS tools	Used to answer ...	C1	C2	C3	C4	C5	Total
Is-A:Type hierarchy or extends relation	How-To-Get-A-In-B	4	9	2	2	18	35
Has-A:Java search, reading code	How-To-Get-A-In-B	11	21	9	11	20	72
Has-Label:File search or JavaDoc	How-To-Get-A, Which-A-In-B	7	2	2	6	6	25
Is-In-Layer:Package explorer	Which-Tier-Has-A How-To-Get-A How-To-Get-A-In-B	3	2	0	2	11	18
Is-In-Tier:JavaDoc or Reading code	Which-Tier-Has-A	0	5	4	1	5	15

Table 30: The C group used Eclipse to answer questions about the run-time structure. Some of the facts in Table 29 are obtained using combinations of different features. Some are obtained from both Eclipse and class diagrams, e.g. Is-In-Layer. Also, sometimes using Package explorer was just to explore classes not to identify a class based on a layer.

Question	Strategy	C1	C2	C3	C4	C5	Total
Which-Tier-Has-A	Package explorer	3	2	0	4	6	15
How-To-Get-A	Java Search	0	8	3	2	0	13
	File search	5	2	0	5	3	15
How-To-Get-A-In-B	Code assist	0	5	1	0	5	11
	Find/replace	2	0	0	6	1	9
Which-A-In-B	Reading code	3	6	0	0	21	30
	Java doc	2	3	0	0	3	8
How-To-Get-A	Call hierarchy	4	3	1	1	1	10
How-To-Get-A-In-B	Type Hierarchy	4	9	0	1	3	17

Table 31: Strategies used by **E** group to answer questions run-time structure. Some of the facts in Table 28 are obtained using combinations of different features.

Question	OOG	E1	E2	E3	E4	E5	Total
Which-Tier-Has-A How-To-Get-A	Explore objects in top-level domains	5	2	4	9	6	26
How-To-Get-A How-To-Get-A-In-B	Explore/search ownership tree	1	3	2	5	10	21
Which-A-In-B	Expand/collapse objects	0	3	7	2	0	12
How-To-Get-A-In-B	Explore incoming/outgoing edges	1	4	12	7	1	25
	Read labeling types	0	0	1	2	0	3
How-To-Get-A How-To-Get-A-In-B	Trace to code	6	9	11	6	8	40

Table 32: Some of the longest code understanding activities related to questions about the RS.

Participant's original question	run-time structure related question	Path	Source	Strategy	Time (min)	Code
code understanding activities						
...just looking for the class that has a function that keeps track of all these positions of all the pieces	How-To-Get-A	1	Eclipse	package explorer reading code	11	10
it [CD] does not really help you a lot in a sense...its nice to see how it is connected, but I need to see a specific... I guess the `Position` of the piece is going to play a role, but it doesn't really seem		2	class diagrams	inheritance associations		
...so there are limited amount of things that are interfacing with `Position` here			Eclipse	type hierarchy call hierarchy reading code JavaDoc		
I'll go to the `Position` class first. I think...an object will be created when you make a move."		1	class diagram	associations	11	16
Now I want to find who is creating this object?	How-To-Get-A		Eclipse	Java search for `Position`		
since this function [`adjustFigurePosition()`] instructs to move, let me execute it to see...I want to make sure that this function [is] getting called when you make a move				GUI testing and debugging		
so now we have to find out where the other action tools are being used and what resources they have ...so thats like a centralized location, which is what I thought about initially instead of wandering all that nonsense!	How-To-Get-A-In-B		Eclipse	Type hierarchy of `Tool`	10	8
alright so Im gonna think that...so Im in `BoardFigure` which extends `ImageFigure` which extends God knows what	How-To-Get-A-In-B		Eclipse	Type hierarchy of `Figure`		7

Table 33: Some of the longest code modification activities related to questions about the RS.

Participant's original question	run-time structure related question	Path	Source	Strategy	Time (min)	code
code modification actitvities						
the `listTo` is not created its null! I guess I don't need to recreate it. I just need to refer to the one who created it somewhere else	How-To-Get-A-In-B	1	Eclipse	GUI testing and debugging	20	
Ok. Now I want to search	How-To-Get-A	2		File search for `new HashMap`		
it's easier for me later I want to access this `figureMap`."	How-to-get-A-In-B	3		added public static getter		
It will save me the hassle of finding where this `BoardDrawing` object is created. Is there a quick way to find out where there are people creating that `BoardDrawing` object?	How-To-Get-A					
oh `addObserver()` so this is added as observer! So how can I retrieve this object?	How-to-get-A-In-B			extract local variable refactoring		

Participants who did not use the OOG followed longer paths to answer their questions about the run-time structure. In general, the E participants, who first used the OOG then explored the code followed shorter paths than the C participants, who first used class diagrams then explored the code or who only explored the code in Eclipse. Even the E participants who did not use the OOG at the beginning of an activity and preferred to use Eclipse explored more code in that activity (e.g., E1,T1.b, Table 21) Both groups benefited from Eclipse navigation features, which enabled them to explore a hierarchy of classes and navigate to class declarations. Even though the C group navigated to only relevant packages and only concrete types in those packages, they still searched for where an instance of the desired type was created. The alternative strategies that the C group used to answer questions about the run-time structure often resulted in more navigations.

For example, one participant started from the root class `BreakThrough` (B), picked the class `Position` and searched for another class `C` that had an instance of type `Position`. He found an instance of type `Position` in the body of the method `generatePieceMultiMap()` inside the class `BreakThroughPieceFactory` (C). While reading that method, he found another object (`figuremap` of type `Map<Position,Figure>`) (A) (Table 34). The participant wanted an object of type `Map`, which resulted in him following the paths `[Position,sub path,BreakThroughPieceFactory]` and `[Position,sub path,Map]` Often times, the C participants found it easier to use a class diagram before navigating arbitrary classes in Eclipse. In those cases, a C participant picked a class, then often followed an association with or a dependency on this class. If the association was with an interface type, a participant further explored inheritance relations in Eclipse of all possible concrete types then filtered out the desired type.

The E group explored a hierarchy of objects or searched an ownership tree of objects and domains. To locate where an object is created, an E participant located that object on the OOG then traced to object creation expressions, which occurred within a field or a method body. To determine how to access an object from another object, an E participant explored a points-to edge with that object then traced to a field declaration in the code. If a desired object of type `A` did not show in top-level domains, a participant expanded different objects of types `B,...,Bn` and possible intermediate objects of types `C,...,Cn` within each parent object looking for the desired object. Similarly, if a participant noticed a points-to edge lifted to nearest visible ancestor of a target object of type `A`, he expanded the destination object until he found a solid edge to the target object.

On the OOG, a participant picked an object then investigated direct incoming or outgoing edges of that object. After finding the target object, the participants navigated directly from the target object to the corresponding line of code. Two things determined length of a path on the OOG: the depth of the target object in the ownership hierarchy (if the target object is hidden within a collapsed object), and the depth of the target object (if a lifted edge points to the target object and the target object is hidden within a collapsed object). The depth of an object in the ownership tree was determined by how many times a participant expanded the substructure of an object of type B and possible intermediate object of type C looking for the target object of type A. The depth of an object that caused a lifted edge to appear was determined by how many objects a participant expanded before he found a solid edge to the target object.

Table 34: Paths followed on OOGs are shorter than those followed using other sources of information.

Question	Paths on the OOG	length	Paths in the code	length
How-To-Get-A	Path1= `<Drawing,MODEL,BT>, <Map,MAPS,Drawing>`	2	Path1= `BreakThrough,BreakThroughPieceFactory, generatePieceMultiMap(),figureMap.` figureMap is a local variable	4
	Path2= `<MiniDrawApp,VIEW,BT>, <ImagManger,owned,MiniDrawApp>, <Map,owned,ImageManager>`	3	Path2=`BoardActionTool, draggedFigure, Figure, AbstractFigure, BoardFigure, BoardDrawing, BoardFigure, BoardActionTool, mouseUp(), BoardFigure.performAction(), MoveCommand.execute(), from, Position, execute(),...,`	14

6.5 Influence of Experience

To determine whether the differences in experience between participants influenced their performance, we performed Analysis Of Covariance (ANCOVA) [14]. We selected as covariates total programming experience, industry experience, and Eclipse experience. Also, since some of the E participants with industry experience did not necessarily have more Java experience than the C participants, we considered the Java experience

Table 35: Results of ANCOVA indicate no significant influence (α=0.05) of Java or industry experience on the independent variables.

Task	covariate	time spent	code explored
T1	Java experience	P=0.48	P=0.52
	Industry experience	P=0.45	P=0.78
	Programming experience	P= 0.46	P=0.83
	Eclipse experience	P= 0.47	P=0.71
T2	Java experience	P=0.39	P=0.92
	Industry experience	P=0.17	P=0.20
	Programming experience	P= 0.14	P=0.28
	Eclipse experience	P=0.64	P=0.59
T3	Java experience	P=0.80	P=0.15
	Industry experience	P=0.50	P=0.63
	Programming experience	P= 0.79	P=0.01
	Eclipse experience	P= 0.08	P=<<0.05

as a covariate. The time spent and code explored by participants are the dependent variables in our case and the group is the categorial factor with two levels "C" and "E".

To determine how the covariates relate to the dependent variables in both groups, we applied two regression models to each combination of the dependent variables and covariates. In some cases, applying ANOVA to both models indicated that removing the interaction between the group and the covariates does not significantly affect the power of the original model. Some examples of interaction plots are in Figures 10,11,and 12. Thus, in those cases, applying ANOVA to the simplified models was justified. Also, since Eclipse experience was rated on a 5-point likert scale, we centered the values around the mean before applying ANCOVA. The summary of resultsof applying ANOVA to the simplified models show that neither Java experience nor industry experience had a significant effect on performance, i.e., the code explored or time spent. For T3, the total programming and Eclipse experience had significant influence on the code explored but not the time spent (Table 35).

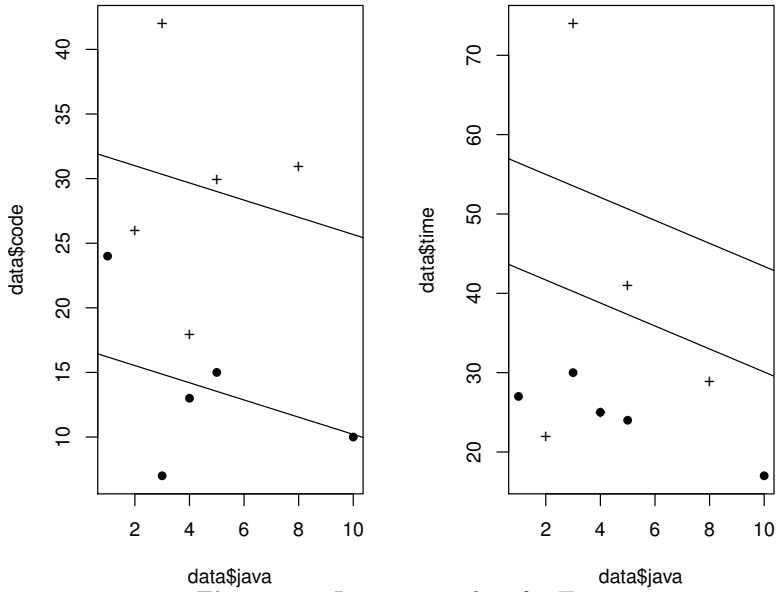


Figure 10: Interaction plots for T1

7 Discussion

7.1 Study Design Issues

Our quantitative analysis lacks statistical significance across all three tasks. Presumably, due to issues in our task design. We designed the tasks to have the participants add the missing game logic to BreakThrough. All of the participants reported that the tasks were not hard in general and were doable in 2.5 hours. The study revealed possible issues in our task design. In general, some code modification tasks may require less information about the RS than others. Admittedly, we could have obtained stronger results by selecting tasks that specifically trigger questions about only the RS or that are highly crafted to trigger specific navigation of the OOG. But we did not do so, to ensure that the tasks are plausible code modification tasks. Also, the participants reported that the tasks T1 and T2 were related since the capture involves a diagonal move, which is a special case of movement, so they had to only change the algorithm for T2 with the extra effort of removing a piece from the board. The results suggest that the time difference increased significantly with T3, presumably since T3 was different from the other two.

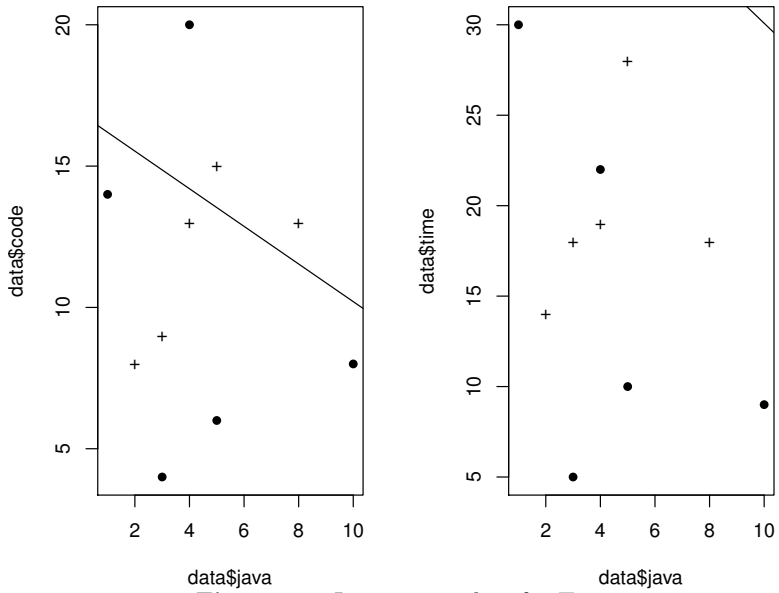


Figure 11: Interaction plots for T2

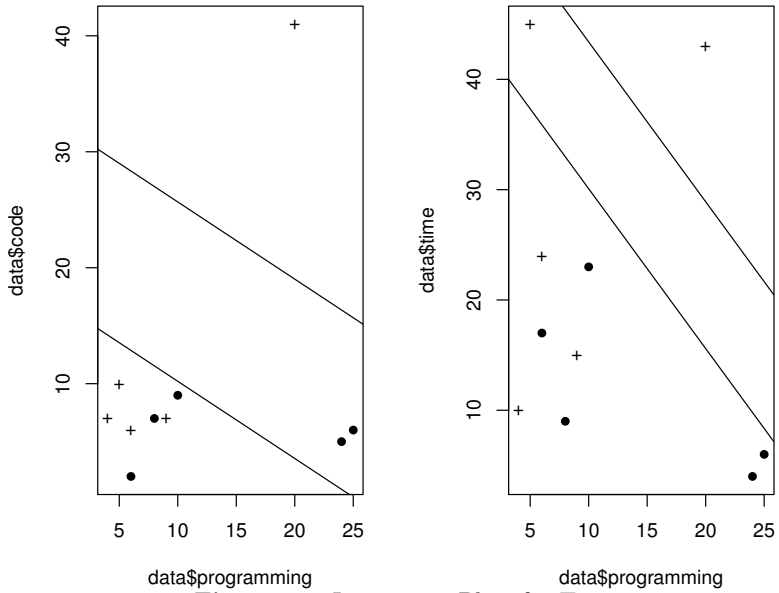


Figure 12: Interaction Plots for T3

7.2 Threats to Validity

Our study may have several threats to validity.

Construct Validity. The lack of statistical significance across all tasks may affect the validity of our conclusion. We attribute the lack of significance to the small sample size, as the small number of observations in these studies is likely to hamper statistical power [37], and study design issues. The OOG, like any points-to graph, provides developers with answers to How-To-Get-A and How-To-Get-A-In-B questions. A main contribution of the OOG is the ability to distinguish between multiple instances of the same type that are created in different domains, i.e., answers to Which-A-In-B questions. Even though the OOG helped in

answering these questions by providing Part-of/Is-Owned facts (Table 28), our tasks did not trigger many such questions (Table 26). We could obtain more significant results by designing tasks that trigger more Which-A-In-B questions or that are highly crafted to trigger specific navigation of the OOG. Instead, we chose plausible code modification tasks that a developer would encounter when completing a board game implementation. Moreover, there could have been a learning effect when the participants moved from T1 to T2, since the two tasks were related in that the capture involves a diagonal move, which is a special case of movement, so they had to only change the algorithm for T2 with the extra effort of removing a piece from the board. The results suggest that the time difference increased significantly with T3, presumably since T3 was different from the other two. Finally, measuring the code explored in addition to the time spent could have been unnecessary. We argue that diagrams can help a developer locate more quickly where the information needed for a task could be in the code. With OOGs, some information about objects and object relations that are scattered in several files in the code is localized and can be found with fewer navigations. OOGs collapse object nodes based on containment, ownership and type structures, not according to where objects are syntactically declared in the program, some naming convention or a graph clustering algorithm, so in our case, the code explored was a measure of needless work in the C group compared to the E group.

Internal Validity. Our study may have some threats to internal validity. First, the participants had varying experience. As indicated in section 6.5, the differences in Java and industry experience did not have a significant influence on developers performance. The total programming experience, which was in some cases more than the industry or Java experience, should not be an issue because MiniDraw is a Java framework that uses object-oriented best practices. We mitigated the uneven reported Eclipse experience by giving the tutorial on Eclipse navigation features and the beginning and throughout the experiment. Second, using examples from MiniDraw to tutor both the OOG and Eclipse could have had a learning effect on the participants, especially since the OOG tutorial was longer than the Eclipse tutorial. We tried to avoid the bias by also demonstrating the Eclipse navigation features using examples from MiniDraw. Also, we used random methods and fields in the class `BreakThrough` that were not specifically related to the tasks that the participants had to complete. However, if a learning effect had occurred, it would have occurred in both groups since both groups tried these examples, with the extra effort in the E group to learn how to navigate the OOG. Third, relying on an interactive OOG in the E group as opposed to relying on only images of class diagrams in the C group could have affected the time results. An interactive class diagram would enable a developer to browse a hierarchy of classes and trace to class declarations, methods, and fields, and both groups benefited from the full features of Eclipse to interactively navigate the code structure.

External Validity. Several factors may affect the generalizability of our findings. First, MiniDraw may not be representative of all code bases. Second, we designed our tasks to implement the logic of a board game, so our tasks may not be representative of real maintenance tasks such as bugs or feature requests submitted by framework developers. Still, our tasks are fairly broad software engineering tasks that were not specifically crafted to favor OOGs. Third, although four of our participants had professional experience, we recruited mostly graduate students. Finally, some participants in our study came from a C++ background. We could have obtained better results had we recruited only Java developers.

7.3 Lessons Learned

This paper dispels several myths. **Object graphs of a hierarchical nature are not over whelming to developers:** First, the prevalent thinking is that developers are overwhelmed by global object graphs. This is certainly true for flat graphs (Table 3). However, for global object graphs of hierarchical nature, hierarchy enables both high-level and detailed understanding. The participants' navigations suggest that they benefited from the abstraction on the OOG to gain both high-level and detailed understanding (Table 31). As explained in Section 4, we designed our questionnaire to measure high-level understanding and detail. We observed that to answer QX1, the participants used Is-In-Tier facts on the OOG to answer that question. To answer QX2, the participants used Points-To facts on the OOG. As for QX3, they used Is-Part-Of and Is-Owned facts (Table 31).

Statically extracted diagrams that show sound approximations are also useful: Second, any statically extracted diagram risks being too imprecise to be useful to developers. One form of imprecision in the OOG include displaying extra edges due to aliasing soundness. This form of imprecision is due to the inherent difficulty of being sound without having false positives in the extracted graph. The other form of imprecision is due to the excessive merging of objects due to possible aliasing. However, before we provide developers with semi-automatically extracted OOGs, we follow a type checker that warns us about any inconsistent annotations which leads us while we refine the extracted OOGs. When we refine the extracted OOGs, we make sure that we use both precise types (by refactoring to generic types) and precise annotations to ensure that the extracted OOG are relatively precise for code modification. But since we use annotations and since we use context-sensitive analysis where the context is the domain, the OOG has sufficient precision. Admittedly, there is additional imprecision [11, Chap.3], but our experiment revealed that OOGs are useful and that that imprecision did not significantly interfere with comprehension (Table 28 and 31).

Class diagrams are not enough for comprehension: The wide spread belief is that class diagrams are good enough for the status quo. Our study confirm that types are not enough for identifying objects and their relations. Since class diagrams show only information about the type structure, they provide developers with only a small portion of the information that they need to answer challenging questions about the run-time structure. In fact, even when they looked at class diagrams, our participants searched for object relations and were not totally satisfied:

> *...class diagrams don't usually provide us with data structures probably in aggregations but still it is not clear what the specific implementation would be...*(C5,T1)

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

> *...Ok. so somehow I need to access* `MiniDrawApplication` *MiniDrawApp is the class that has the* `showStatus()`*, so you know I need the object its not a static class...*(C5,T1.d,e)

OOGs provided information about the run-time structure that participants needed to answer the questions that deal with objects and their relations, thus improved comprehension by being complementary to type structure diagrams:

> *...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)

> *class diagrams dont usually provide us with data structures probably in aggregations but still it is not clear what the specific implementation would be...*(C5,T1)

> *...okay so somehow i need to access MinidrawApplication. MiniDrawApp is the class that has the showstatus so you know i need the object its not a static class...* (C5,T1.c)

Diagrams are of great value to developers: Third, it is commonly understood that expert developers tend to use diagrams less than novice ones, as the former could understand the system by just exploring code. Subjective answers to QX.5 suggest that the participants preferred reading code over using diagrams.:

> *...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...*

> *...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...*

> *...okay so usually I start from the code I get a high level understanding of the program and what it does then I go back to a diagram like this [OOG] and it is more helpful...*

> *...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...*

However, we not only relied on participants' perceived usefulness of different sources of information, but also on their navigations, i.e., the information sources that they actually used (Table 36). It is clear that

Table 36: Participants in both groups used diagrams frequently regardless of their experience level. E participants referred mainly to the OOG while the C participants referred to class diagrams.

Participant	Diagram usage	Java Exp.	Ind. Exp.
C1	10	5	0 (Ph.D.)
C2	25	8	6 (Ph.D.)
C3	17	2	4 (M.Sc.)
C4	13	4	0 (Ph.D.)
C5	20	3	0 (B.S.)
E1	9	1	2 (M.Sc.)
E2	14	4	0.5 (Ph.D.)
E3	13	5	20 (M.Sc.)
E4	11	10	20 (Ph.D.)
E5	18	3	2 (B.S.)

regardless of their level of experience, they used both Eclipse and diagrams. The numbers in the table show how many times they switched back and forth between Eclipse and diagrams, but do not provide details on how much time they spent using those diagrams or how long they spent using Eclipse or which features they used. The details of Eclipse feature usage is in Table 30.

The participants expressed their need for the diagram whenever the code did not provide them with enough information:

> *so we need to have the one [class diagram] that has BoardFigure on it and then figure out who is ... [looking at CD6,CD1]FigureFactory, BoardDawing, BoardFigure...*(C2,T2)

> *yeah so what I'm searching for now is a representation of the board itself..the container that holds all the pieces on the board and I don't see any containers in here [Eclipse], so I'm not looking at the right place. I need to back up. Let's go back to the diagram [OOG]. Let me see so that's [figure:BoardFigure] the individual piece, so boardDrawing maybe that's where the container is so lets take a look at there [expand boardDrawing], so property map... an ArrayList of figures that's a container this looks like what I'm looking for...*(E3,T1)

> *so lets look inside figures [CompositeFigure] ... all these methods are UI stuff so I'm in the wrong spot... ok I was in MoveCommand. Let me actually go back here [OOG] see this is helpful! ...so I'm gonna go inside [expand object] this dude [boardDrawing] ok so that's cool so now I'm here..*(E4,T1)

> *...so what references the position? Ah OK, so that actually helps.. looks like there are just two objects.. Im gonna go inside this one [boardDrawing:BoardDrawing]..even though i did not think it had anything to do with it because it's name doesn't make any sense to me...so AbstractFigure::owned I was there and it wasn't there AhA figureMap..dada..list to...thats gonna be structural because what I'm looking at are object references BoardDrawing is what I want...*(E4,T1)

They also referred to diagrams whenever they wanted to remember how classes or objects are related. In the case of class diagrams, the diagram provided only partial support:

> *...here Im not exactly sure Im doing some experiment. let me take a look at the diagram [CD6]. BoardFigure[CD1] Im looking for I forget ..how [CD2]. I think i should look at the boardfigures implementation. [I looked at the class diagrams] because I forgot how the image manager is connected to this boardfigure. Im not sure how to use it*

> *so let me go back to the diagrams and look for something related to the how the user moves the pieces and implement some sort of logic code to only allow valid moves let me take a look at the game interface [CD2,CD6] game stub actually implements it*

Understanding the run-time structure is as important as understanding the behavior. Fourth, it seems that understanding the run-time structure is as important as understanding the behavior. Our participants expressed their need for GUI testing to understand some behavior, such as how direct manipulation using mouse clicks occurs (e.g., *"how a figure is dragged?*) or how rendering occurs (e.g., *"how the view gets repainted?*) The GUI testing helped the participants understand the behavior, but they still needed to understand the run-time structure. All three tasks required the participants to understand relations between

instances of concrete types of `Tool`, `Figure`, `Drawing`, and `DrawingView`. The other intuitive strategy for participants was to investigate call graphs, but it was not enough for answering questions about the object structure. For example, MiniDraw uses the Observer pattern, which has two roles, the subject (container of data) and the observer (the object to notify upon data changes). Understanding "what" instances got notified by a change was as important to our participants as understanding "how" the notification occurs:

> *...OK. I guess I dont entirely have to understand exactly what it [a method] is doing. But Im not sure exactly which objects to use to control that behavior. It does give me the positions, so I guess I would like to know a little bit more about how the* `Position` *is stored...*
>
> *...Is there a quick way to find out where there are people creating that* `BoardDrawing` *object?Oh* `addObserver()` *so this is added as observer! So how can I retrieve this object?...*

An OOG is not specifically designed to highlight design patterns in object-oriented frameworks or to explain object behavior, but it expresses the design intent in the code. Since the OOG treated both the subject and the observer as objects, the E group benefited from points-to edges between those objects to answer their questions.

For example, in MiniDraw, a double observer chain is used, where a `Figure` object notifies a `Drawing` object which in turn notifies a `DrawingView` object. Therefore, the type `Drawing` is a `FigureChangeListener` and a `DrawingView` is a `DrawingChangeListener`. The E participants were able to identify a points-to relation between `game:GameStub` and `boardDrawing` then trace to a field of type `BoardGameObserver`. Also, On the OOG, the E participants were able to identify a relation between an `ArrayList` of `DrawingChangeListener` objects inside `boardDrawing` and `StdViewWithBackground` as well as a labeling type `DrawingChangeListener` on `StdViewWithBackground` (Fig. 6).

OOGs are not useful for every code modification Task: Finally, in general, some modification tasks require more information about the run-time than others. Even though our results suggest that even simple code modification tasks that are not specifically designed to promote OOGs, trigger many questions about the run-time structure, such as How-To-Get-A and How-To-Get-A-In-B questions, and that the OOG can successfully answer such questions. The OOG, like any points-to graph, provides developers with answers to these two questions. The novelty in the OOG lies in its ability to distinguish between multiple instances of the same type that are created in different domains, i.e., answers to Which-A-In-B questions. Therefore, we could obtain more interesting results by designing tasks that trigger more Which-A-In-B questions.

8 Related Work

Research in the area of program comprehension involved both developing theories of how developers understand software and tools that aid in software understanding. We discuss several areas of related work, including researchers effort to define theories of program comprehension, studies on analyzing developers questions during comprehension tasks, and studies on evaluating tools and diagrams for program comprehension. Several graphs have been proposed to visualize objects, but have not yet been evaluated for code modification, so we also survey some of these graphs by comparing their extraction approaches to our approach.

8.1 Theories of Program Comprehension

Storey et al. [57] discussed bottom-up and top-down comprehension. Pacione et al. [44] proposed a visualization model to support comprehension at multiple levels of abstraction and identified which diagrams can be used to satisfy the questions at each level.While other theories build on general aspects of comprehension, such as bottom-up and top-down comprehension [57] and comprehension at multiple levels of abstraction [44], our theory is based on identifying questions that developers ask about the run-time structure, and the facts that they rely on to answer those questions. So, our goal isnot to define a new theory of comprehension or to compare to other theories of comprehension, but to complement existing theories and fill gaps in current diagrams and tools. Our effort is to identify the specific questions that developers ask about the object

structure and the facts about the object structure that they mostly rely on to answer those questions. Once we identify the difficulties and the most useful facts on the OOG to overcome those difficulties, we will fill the gap in current tools and diagrams by designing a tool that is most useful to answer those specific questions to developers.

8.2 Studies on Analyzing Developers Questions

Classifying developers questions has been the focus of several researchers [55, 34, 22], but their analysis focused on the range of questions that developers ask about the code in general. Also, they mixed questions about objects with other types of questions. Based on two studies, Sillito et al. [55] defined a catalog of 44 types of questions under four top-level categories, and they listed questions about objects, control flow, and execution paths under the same category. In their model, we were able to identify three questions about the object structure: *How are instances of these types created and assembled? How are these types or objects related (whole part)?* and *What is the correct way to access this data structure?* LaToza et al. [34] conducted a survey, where they asked developers to report the hard-to-answer questions about code, and classified these questions under three categories. In their classification, we found a few questions that could be related to object structures. They classified these questions under "hard-to-answer questions about how to implement a code change", such as *which function or object should I pick?* or questions about architecture, such as *how is this functionality organized into layers?* Our method is different, in that we did not prompt the developers to explicitly report their questions, and they were not aware of our classification model. Our classification is based on observing questions about only the run-time structure that our participants actually struggled with and facts on the OOG that they actually used. The classification scheme reported in this paper is a result of what we observed in this study and a previous exploratory study [7], so it provides a more fine grained classification of possible questions and facts about only the run-time structure that are not included in their catalog. In that study, we defined a preliminary model of questions or facts about object structures by analyzing the participants think-aloud (Table 19). Since this study involved more participants that the previous study, we analyzed the participants questions and refined the initial coding model by having only four main questions about the object structure. For each of the main questions, we list the facts that the participants used to answer that question. We also expressed the facts that the participants used on the OOG in terms of our theory of comprehension using both types (of instance) and domains. Fritz et al. [22] also interviewed developers, but they focused on very high-level questions that are not directly related to changing the code. In their studies, these researchers also surveyed the tool support available to answer the questions in each category. Sillito et al. found that only 34% of all questions in all categories had full tool support. Of that 34%, 0% of questions under the category that covers questions about objects had full tool support and 13% of those questions had partial support. They found that answering precise questions about objects require both static and dynamic information and they did not identify any direct tool support to answer such questions, so they suggested general visualization frameworks, such as SHriMP. Still, SHriMP would require, as input, a statically extracted graph of the run-time structure and they did not identify any tool that directly produces object graphs. LaToza et al. identified the tools that could potentially help answer a question, but did not study to what extent a tool was useful.

8.3 Studies on Evaluating Diagrams for Program Comprehension

Several diagrams have been proposed 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 we discuss how widely these diagrams have been evaluated empirically, which aspects of these diagrams have been investigated in detail, and how they are different from our evaluation method described in this paper.

Evaluating object-based diagrams. Many studies identified the importance of object-based diagrams and proposed solutions to complement class-based diagrams.

UML static structural object-based diagrams. To our knowledge, only few researchers studied static object

Table 37: Survey of studies on program comprehension.

Diagram/Tool	Publication	Sample size	Tasks
Object graphs	Quante et al. [46]	25	Feature location
UML diagrams	Hadar et al. [26]	55	Questionnaires
Diagramming tools	Lee et al. [36]	19	Interviews
Object diagrams	Torchiano et al. [54]	24	Questionnaires
Call graphs	LaToza et al. [35]	12	Control flow questions
Code exploration tools	Robillard et al. [50] Alwis et al. [18]	5 18	Code modifications

diagrams. Tonella et al. [59] compared, in a case study, static object diagrams to dynamic object diagrams. Torchiano et al. conducted a controlled experiment followed by an external replication [54] to evaluate the usefulness of UML static object diagrams as compared to class diagrams. They found that object diagrams are significantly more useful when combined with class diagrams than using only class diagrams. They used manually created diagrams and their study was questionnaire-based.

UML dynamic behavioral object-based diagrams. Much of the research on object-based diagrams was done on dynamic, behavioral views such as sequence diagrams and collaboration diagrams, in comparison to class diagrams [26, 24, 8]. A study by Hadar et al. [26] on evaluating different types of UML diagrams identified that developers consult class diagrams to study static relations, but to understand the dynamic behavior, they need sequence diagrams. The diagrams used in those studies were manually crafted to describe specific scenarios and the studies were questionnaire-based. The OOG is a global diagram that is a sound approximation for all possible scenarios [4].

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

UML structural class-based diagrams. Ricca et al. [48] and De Lucia et al. [19] studied, in controlled experiments, the usefulness of class diagrams for comprehension. However, their studies were questionnaire-based and focused on specific aspects of diagrams, such as the effect of stereotypes on web application design comprehension [48] and comparing class diagram notation to Entity-Relationship diagram notation for data model comprehension [19]. The diagrams that they used were manually crafted and were not reverse-engineered from the code, so there was no direct correspondence between the elements on the diagrams and the code elements.

Enhancing class-based diagrams. A few researchers have enhanced class diagrams. such as integrating both dynamic and static views. Lallchandani et al. [32] worked on combining information extracted from sequence diagrams along with those from class and state-machine diagrams into a model dependency graph (MDG). Gogolla et al. [25] proposed the use of layered graphs to describe both type graphs and instance graphs. Riehle et al. [49] 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. Torchiano et al. [38] have extended the class-centered model by creating a Hierarchical Instance Model based on a schema taken from the class model to describe a system model like a business process. Our effort is not to particularly enhance class diagrams. Those approaches are manual. Since we believe developers should base their decisions on diagrams that are consistent with the code, we provide them with diagrams that are extracted from the code and we express design intent using annotations.

8.4 Approaches for Object Graph Extraction

Several researchers worked on producing object-based diagrams, which were either statically extracted or dynamically extracted from object-oriented Java code (Table 2).

Statically extracted object graphs. Some approaches statically extract flat object graphs from object-

oriented code either automatically, including Womble [29], Ajax [43], and Pangaea [56], or using annotations [33]. While these approaches can be useful for showing object interactions, they share a fundamental scalability limitation. 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. For example, Lam and Rinard [33] 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. The Scholia [4] approach extends Lam and Rinard's both to handle hierarchical architectures and to support object-oriented language constructs such as inheritance.

Dynamically extracted object graphs. Dynamically extracted object graphs, on the other hand, consider specific executions of the system. Quante proposed Dynamic Object Process Graphs (DOPGs) [46]. 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, in a controlled experiment, that the DOPG helped for concept location tasks on one codebase but not another. He presented quantitative numbers of success and time, but it is unclear how DOPGs were used or why they 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 [20]. Dynamic structures are more precise, but they are, by definition, partial and hold for specific scenarios. 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, developers should base their decisions on a sound diagram with sufficient precision.

8.5 Our previous studies on evaluating OOGs.

OOGs were developed on seven systems [2] and several aspects of OOGs have been evaluated before this study [3]. As part of our work on evaluating OOGs for comprehension, we previously conducted an exploratory study [7] and a case study [5]. In the exploratory study, we observed three participants perform code modifications while using OOGs, but our analysis remained qualitative since the tasks changed slightly between the participants and we did not have a control group. The case study provided qualitative data of how the OOG answered several questions the developer had about the run-time structure (in Table 19, their codes are Is-In-Tier, Is-Owned-By and Is-Part-Of). It also involved one participant, which limited the external validity of our findings. The controlled experiment reported in this paper is the first to evaluate, in the context of code modifications, global hierarchical object points-to graphs, that are statically extracted from the code, and that had been difficult to obtain using prior technology. The experiment in this paper overcomes limitations in our previous studies.

8.6 Tools and Diagrams of other program representations

Several researchers have proposed diagrams of other program representations and developed tools to visualize those diagrams (Table 37).

Points-to and shape analysis. The hierarchical object graph in a OOG is a kind of a points-to graph. Many static analyses extract points-to graphs [41], as well as shape graphs [53]. These analyses have the stated goal of aiding program comprehension but their results have not been evaluated empirically with developers asked to perform coding tasks. Moreover, unlike OOGs, shape graphs are neither hierarchical nor global. They rather illustrate some key interactions between a few objects. We believe the OOG can help developers understand the global run-time structure. Then, developers can launch a highly precise, intra-procedural shape analysis to study low-level details within a specific method.

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 [27, 35]. For instance, LaToza and Myers [35], in

their REACHER tool, statically extract path-sensitive call graphs. Indeed, they found that developers using REACHER were more effective compared to using Eclipse call hierarchies. However, REACHER does not track or display objects, to avoid an exponential blowup in the number of paths to display.

Ownership structures. Several tools [28, 45, 42] visualize ownership structures using dynamic analysis. Mitchell et. al [42] built the YETI tool for diagnosing storage inefficiencies and lifetime management bugs. These tools have not been evaluated for their usefulness in assisting developers with code modification tasks.

Dynamic code information. Other researchers worked on augmenting the currently available static code views in IDEs with extra information. Rothlisberger et al. [52] have recently developed the SENSEO tool as an extra support for developers exploring Java code in Eclipse. The purpose of the tool is to extract dynamic information from the code to enable developers to view dynamic information about the code while working in the IDE. Static and dynamic structures are complementary, in that the first is sound with respect to the objects and their relations, but it does not display the actual number of allocated objects. Dynamic structures are more precise, but they are, by definition, partial and hold for specific scenarios.

9 Conclusion

Comprehension is characterized by theories of how developers understand software and tools that assist in comprehension.In this paper, we posit that types are not enough for object-oriented code comprehension, and that instances matter. We provide a classification of challenging questions about the run-time structure asked by developers during code modifications, that can be answered using facts that only appear on a DRS. Our theory predicts that an OOG can answer questions about the run-time structure more easily because it provides the ability to distinguish the role that an instance plays not just by type, but by named groups (domains) or by position in the run-time structure (ownership). We designed and conducted a controlled experiment to investigate whether an OOG can answer developers questions about the run-time structure more easily than DCS tools, and thus reduce comprehension effort. We identified several questions about the run-time structure asked by developers, and we provided a preliminary classification of those questions. We found that the OOG helped developers answer those questions more easily than DCS tools. On average, the OOG had a positive effect of varying extents on comprehension that reduced the time spent by 22%-60% and the irrelevant code explored by 10%-60%. The participants who used information about the RS, in addition to information the CS always outperformed those who used only information about the CSThere were differences in statistical significance across different tasks, and the effect sizes reported lead us to think that the small sample is the most probable culprit for the lack of significance, which calls for external replication.

One question remains: is an observed positive effect of using OOGs worth the effort spent in extracting these diagrams? The effort is currently estimated at 1 hr/KLOC, and active research in the inference of ownership types promises to significantly reduce the annotation burden. Given the considerable costs of software maintenance and evolution, a measured improvement in developers' performance on code modification tasks justifies de-emphasizing DCS tools, which are reasonably mature, and instead, building useful DRS tools.We are enhancing the OOG to show, in addition to points-to relations, richer data-flow communication [60]. On our end, we are mining the usage data we gathered in this study to enhance our current tool.

References

[1] `www.cs.wayne.edu/~mabianto/oog_study2/`, 2012.

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

[3] M. Abi-Antoun and J. Aldrich. A Field Study in Static Extraction of Runtime Architectures. In *PASTE*, pages 22–28, 2008.

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

[5] M. Abi-Antoun and N. Ammar. A Case Study in Evaluating the Usefulness of the Run-time Structure during Coding Tasks. In *HAoSE*, 2010.

[6] M. Abi-Antoun, N. Ammar, and Z. Hailat. Extraction of Ownership Object Graphs from Object-Oriented Code: an Experience Report. In *QoSA*, 2012.

[7] M. Abi-Antoun, N. Ammar, and T. LaToza. Questions about Object Structure during Coding Activities. In *CHASE*, 2010.

[8] S. Abrahao, E. Insfran, C. Gravino, and G. Scanniello. On the Effectiveness of Dynamic Modeling in UML: Results from an External Replication. In *ESEM*, 2009.

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

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

[11] N. Ammar. Evaluation of the Usefulness of Diagrams of the Run-Time Structure for Coding Activities. Master's thesis, WSU, 2011. Chap.3 discusses OOG refinement.

[12] N. Ammar and M. Abi-Antoun. Adding Ownership Domain Annotations to and Extracting Ownership Object Graphs from MiniDraw. Technical report, WSU, 2011.

[13] K. H. Bennett, V. Rajlich, and N. Wilde. Software evolution and the staged model of the software lifecycle. *Advances in Computers*, 2002.

[14] J. M. Chambers, A. E. Freeny, and R. M. Heiberger. Analysis of Variance; Designed Experiments. In J. M. Chambers and T. J. Hastie, editors, *Statistical Models in S*. Wadsworth & Brooks/Cole, 1992.

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

[16] N. Cliff. Answering Ordinal Questions with Ordinal Data Using Ordinal Statistics. *Multivariate Behavioral Research*, 1996.

[17] A. Crystal and B. Ellington. Task Analysis and Human-Computer Interaction: Approaches, Techniques, and Levels of Analysis. In *AMCIS*, 2004.

[18] B. de Alwis, G. Murphy, and M. Robillard. A comparative study of three program exploration tools. In *ICPC*, 2007.

[19] 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.

[20] B. Demsky and M. Rinard. Role-Based Exploration of Object-Oriented Programs. In *ICSE*, 2002.

[21] C. Flanagan and S. N. Freund. Dynamic Architecture Extraction. In *Workshop on Formal Approaches to Testing and Runtime Verification*, August 2006.

[22] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. ICSE '10, 2010.

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

[24] 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 *MoDELS*, 2008.

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

[26] I. Hadar and O. Hazzan. On the Contribution of UML Diagrams to Software System Comprehension. *JOT*, 2004.

[27] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the Neighborhood with Dora to Expedite Software Maintenance. In *ASE*, 2007.

[28] T. Hill, J. Noble, and J. Potter. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *JVLC*, 2002.

[29] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *TSE*, 2001.

[30] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg. Systematic review: A systematic Review of Effect Size in Software Engineering Experiments. *Inf. Softw. Technol.*, 2007.

[31] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary Guidelines for Empirical Research in Software Engineering. *TSE*, 2002.

[32] J. Lallchandani and R. Mall. Integrated state-based dynamic slicing technique for uml models. *Software, IET*, 2010.

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

[34] T. D. LaToza and B. A. Myers. Hard-to-Answer Questions about Code. In *PLATEAU*, 2010.

[35] T. D. LaToza and B. A. Myers. Visualizing Call Graphs. In *VL/HCC*, 2011.

[36] S. Lee, G. Murphy, T. Fritz, and M. Allen. How Can Diagramming Tools Help Support Programming Activities? In *VL/HCC*, 2008.

[37] M. W. Lipsey. *Design Sensitivity Statistical Power for Experimental Research*. Sage, 1990.

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

[39] M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *PASTE*, pages 43–49, 2008.

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

[41] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-To Analysis for Java. *TOSEM*, 2005.

[42] N. Mitchell, E. Schonberg, and G. Sevitsky. Making Sense of Large Heaps. In *ECOOP*, 2009.

[43] R. W. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, CMU, 2001.

[44] M. J. Pacione, M. Roper, and M. Wood. A Novel Software Visualisation Model to Support Software Comprehension. In *WCRE*, 2004.

[45] A. Potanin, J. Noble, and R. Biddle. Checking Ownership and Confinement. *Concurrency and Computation: Practice and Experience*, 2004.

[46] J. Quante. Do Dynamic Object Process Graphs Support Program Understanding? - A Controlled Experiment. In *ICPC*, 2008.

[47] D. Rayside and L. Mendel. Object Ownership Profiling: a Technique for Finding and Fixing Memory Leaks. In *ASE*, 2007.

[48] 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.

[49] D. Riehle. JUnit 3.8 Documented Using Collaborations. *SIGSOFT Softw. Eng. Notes*, 2008.

[50] M. P. Robillard, W. Coelho, and G. Murphy. How Effective Developers Investigate Source Code: An Exploratory Study. *TSE*, 2004.

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

[52] D. Rothlisberger, M. Harry, W. Binder, P. Moret, D. Ansaloni, A. Villazon, and O. Nierstrasz. Exploiting Dynamic Information in IDEs Improves Speed and Correctness of Software Maintenance Tasks. *TSE*, 2011.

[53] M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In *POPL*, 1999.

[54] G. Scanniello, F. Ricca, and M. Torchiano. On the Effectiveness of the UML Object Diagrams: a Replicated Experiment. *IET Seminar Digests*, 2011.

[55] J. Sillito, G. Murphy, and K. D. Volder. Asking and Answering Questions during a Programming Change Task. *TSE*, 2008.

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

[57] M.-A. Storey. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In *IWPC*, 2005.

[58] 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.

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

[60] R. Vanciu and M. Abi-Antoun. Extracting Dataflow Communication from Object-Oriented Code, 2011.

APPENDIX

A Excerpts from the Transcripts

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

B Answers to Questionnaires

Table 38: 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-A
C2	"I mean somebody is keeping track of all these pieces so i want to figure out who is doing that"	How-To-Get-A
	"and then figure out how in this execute function access that . . . "	How-To-Get-A
	"I'm more concerned about how to get to [BoardDrawing]"	How-To-Get-A
	"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-A
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 39: 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 6.

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 40: 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 7.

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-A		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-A How-To-Get-A		Ec.F10	

Table 41: Responses of the C participants to QX.1(Table 9).

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 42: Responses of the E participants to QX.1 (Table 9).

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 43: Responses of the C participants to question QX.4 (Table 9).

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 44: Responses of the E participants to QX.4 (Table 9).

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`"