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

Nariman Ammar Marwan Abi-Antoun
Department of Computer Science, Wayne State University, Detroit, Michigan, USA
Email: {nammar, mabiantoun}@wayne.edu

***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 evaluate, in a controlled experiment, whether an OOG, as a diagram of the run-time structure, 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). We observed 10 participants, for 3 hours each, perform three feature implementation tasks on a framework application. 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.

***Keywords*-controlled experiment, program comprehension**

I. 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 [1]. To support comprehension, researchers have produced many tools to visualize the structure of a software system based on the widespread belief that diagrams are useful for comprehension.

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 [2]. 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.

One DRS recently proposed and formalized by Abi-Antoun and Aldrich [3] 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 based on *annotations*. In this paper, we deemphasize evaluating the effort of extracting and refining OOGs, which we have recently measured [4]. The interested reader can refer to [5] and [6, 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;
- A preliminary classification of questions that developers ask about the run-time structure;
- A controlled experiment to evaluate the effectiveness of the OOG, as a DRS, for code modification tasks as compared to DCS tools.

Outline. In the rest of this paper, we discuss related work (Section II) and motivate the need for OOGs (Section III). In Section IV, we explain our theory. Next, we describe our method (Section V), our analysis (Section VI), and both quantitative and qualitative results (Section VII). We then discuss threats to validity (Section VIII), and conclude.

II. Related Work

We discuss related work in the area of program comprehension, including theories of comprehension, developers questions about the code, and researchers effort to produce and evaluate diagrams for comprehension.

While other theories build on general aspects of comprehension, such as bottom-up and top-down comprehension [7] and comprehension at multiple levels of abstraction [8], 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 is to complement existing theories and fill gaps in current diagrams.

Classifying developers questions has been the focus of several researchers, but their analysis focused on 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. [9] defined a catalog of questions and they listed questions about objects, control flow, and execution paths under the same category. LaToza et al. [10] conducted a survey, where they asked developers to report the hard-to-answer questions about code. In their studies, these researchers also surveyed the tool support available to answer the questions in each category. Sillito et al. found that answering questions about objects require both static and dynamic information and they did not identify any direct tool support to answer such questions. LaToza et al. identified the tools that could potentially help answer a question, but did not study to what extent a tool was useful.

Several researchers evaluated diagrams for comprehension and some of them have proposed new diagrams and tools to aid in comprehension. We discuss how the evaluation methods in these studies are different from our method.

Many studies identified the importance of object-based diagrams and proposed solutions to complement class-based diagrams. Tonella et al. [11] compared, in a case study, static object diagrams to dynamic object diagrams. Torchiano et al. conducted a controlled experiment followed by an external replication [12] 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. Torchiano et al. have extended the class-centered model by manually creating a Hierarchical Instance Model based on a schema taken from the class model [13]. Developers can benefit more from diagrams that are consistent with the code, so we provide them with diagrams that are extracted from the code and we express design intent using annotations.

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 [14], [15]. 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 [3].

Several graphs have been extracted from object-oriented Java code. Some approaches statically extract flat object graphs either automatically, including WOMBLE [16], AJAX [17], and PANGAEA [18], or using annotations [19]. While these approaches can be useful for showing object interactions, they share a fundamental scalability limitation. For example, Lam and Rinard [19] also use annotations, but do not achieve the same level of domain-sensitivity, thus extract flat object graphs. Many static analyses extract points-to graphs [20], as well as shape graphs [21]. These analyses have the stated goal of aiding program comprehension but their results have not been evaluated 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.

Dynamically extracted graphs, on the other hand, consider specific executions of the system. Quante proposed Dynamic Object Process Graphs (DOPGs) [22]. 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, but it is unclear how DOPGs were used or why they helped only sometimes. Demsky and Rinard also used dynamic analysis to extract role-based object diagrams [23]. Several tools [24], [25] visualize ownership structures using dynamic analysis. These tools have not been evaluated for their usefulness for code modifications. Rothlisberger et al. [26] have recently developed the SENSEO tool to enable developers to view dynamic information about the code while working with static code views in the IDE. Static and dynamic structures are complementary, in that the first is sound with respect to 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. 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.

OOGs were developed on seven systems. As part of our work on evaluating OOGs for comprehension, we previously conducted an exploratory study [27] and a case study [28]. Those studies did not have enough participants and we did not have a control group, so our analysis remained qualitative. The controlled experiment reported in this paper is the first to evaluate global hierarchical object points-to graphs, that are statically extracted from the code, and that had been difficult to obtain using prior technology.

III. BACKGROUND

We motivate the need for the OOG, as a DRS, by comparing it to diagrams that visualize the structure of a software system (Table I). We use examples from our subject system. MiniDraw is a pedagogical object-oriented framework specifically designed for creating board game applications [30]. It consists of around 15,00 lines of Java

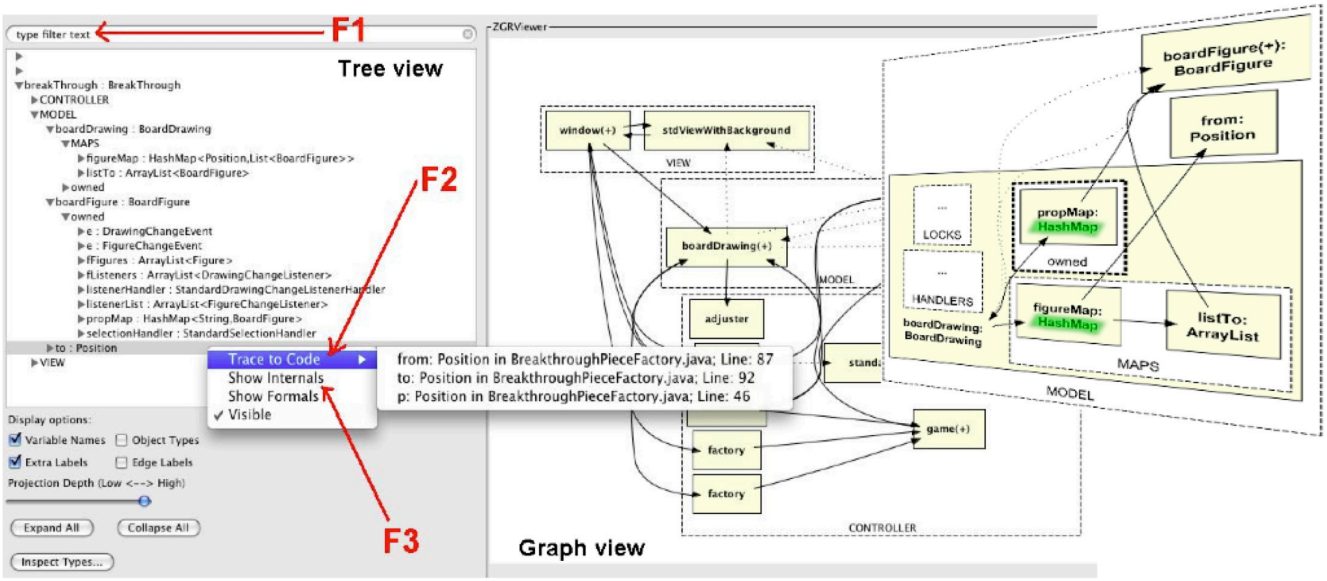


Figure 1. A viewer to interactively navigate the OOG. The graph shows the OOG with nested boxes indicating objects. The tree enables developers to search for an object (F1), trace from a selected object or edge to the code in Eclipse (F2), and collapse or expand an objects sub-structure (F3).

code, 31 classes and 17 interfaces and follows the Model-View-Controller architecture.

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 I) to illustrate specific scenarios. The naive approach for extracting a DRS produces a flat object graph (Table I), 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 I). 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. 1). 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. 1).

IV. A Theory of Comprehension

We define a theory in comprehension in terms of facts that developers can learn from an OOG (Table II).

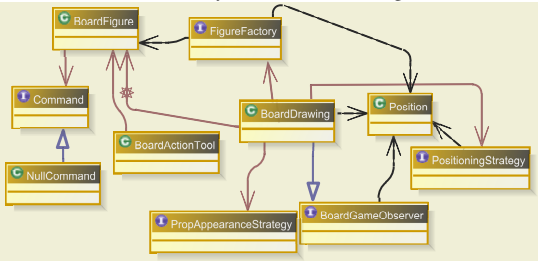
Instances matter in object-oriented code. In object-oriented design patterns, much of the functionality is determined by what instances point to what other instances. For example, in the Observer design pattern [2], understanding "what" gets notified during a change notification is crucial for the operation of the system, but "what" does not usually mean a type, "what" means an instance.

Do specific instances really matter? The OOG merges instances of the same type that are in the same domain, e.g. `Position` (Fig. 1). 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 II).

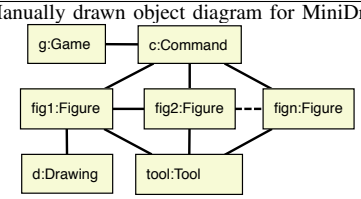
Does information about $types + ownership + domains$ **on the OOG answer key questions in program com-**

Table I
STATE OF THE ART IN DIAGRAMS OF OO SYSTEM STRUCTURE.

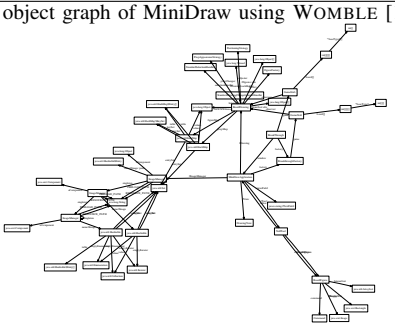
Partial automatically extracted class diagram [29]



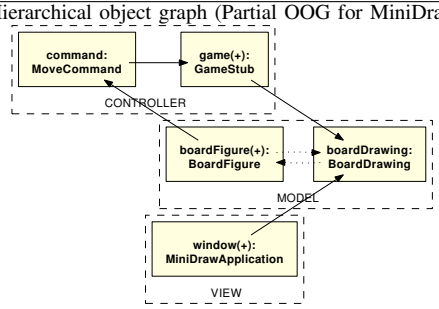
Manually drawn object diagram for MiniDraw



Flat object graph of MiniDraw using WOMBLE [16])



Hierarchical object graph (Partial OOG for MiniDraw)



prehension? 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 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 II).

V. METHOD

We followed the between-subjects design 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. The *independent variable* in our experiment was having access to the OOG. We used as *dependent variables* the number of code elements explored and the time spent by a participant on each task. 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.

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 while performing code modifications than developers who use only DCS tools.

H2: Developers who use a DRS spend less time while performing code modifications than developers who use only DCS tools.

Participants. We advertised the study around the Computer Science Department at Wayne State University. We had 14 respondents, which we pre-screened for 1 hour each. We selected 10 participants (Table III): 4 professional programmers, 3 Ph.D. students in their 4th year, 2 M.S. students, and 1 senior undergraduate. The median in programming experience was 8.5 years , while the median in Java experience was 4 years. All were familiar with Eclipse and UML, and all except one, with frameworks and design patterns.

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. Both groups also received 6 diagrams that we reverse-engineered using AgileJ [29]. Five of these diagrams explained class relations in the five packages of MiniDraw, while the last diagram described dependencies and associ-

Table II
FACTS ABOUT THE RUN-TIME STRUCTURE PROVIDED BY AN OOG.

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 (e.g. `from:Position`) in that domain (Fig. 1)
May-Alias	An object on the OOG may represent more than one run-time object. A developer can pick an object, e.g., `Position` and trace to all possible `new Position()` expressions (Fig. 1-F2)
Points-To	A developer can explore all incoming points-to edges (solid arrows, Fig. 1) to an object . If an edge is lifted (dotted arrows, Fig. 1), 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. 1-F1) or expand an object, e.g. `boardDrawing` (Fig. 1-F3) looking in different domains for different objects that are strictly encapsulated or logically contained in that object

Table III
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

ations with the main class `BreakThrough`. In addition, the E group received a printed OOG. Since the OOG is hierarchical, we installed an interactive viewer of the OOG in Eclipse (Fig. 1) to allow the E group to interactively expand objects (Fig. 1-F3) or search the ownership tree (Fig. 1-F1). We used Camtasia to record the participants' think-aloud as well as a screen capture of their navigations. The study materials are available on our online appendix [30].

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.
- **T3** Implement the undo move feature.

Procedure. Our experiment was in the form of a 3-hour session. The experimenter briefly introduced MiniDraw, then she tutored the participants on the basic navigation features in Eclipse. She gave the E group a 20-minute tutorial explaining the OOG notation and how to interactively navigate the OOG. 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 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, she used a recurring questionnaire between the tasks (Table IV) to measure the level of comprehension by a participant. At the end, exit interview questions captured the participants' subjective feedback.

Table IV
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?

VI. DATA ANALYSIS

We transcribed the video recordings and screen captures offline. We measured the code elements explored by counting the navigation targets to which a participant navigated in Eclipse, including classes, methods, fields, and local variables. In the E group, we also counted nodes on the OOG. Our analysis of 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 included the time a participant spent answering a question, thinking about a task, implementing a task, and testing the implementation. Also, since we are measuring time benefits, we considered the time spent in both successful and unsuccessful attempts. Answers to questionnaires are in [31, Appendix B].

VII. RESULTS

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. 2).

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 [32], 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[1]. We also used non-parametric effect size (Cliff's delta) along with the corresponding confidence intervals with a 95% confidence level. Delta is bound to be negative as, generally, lower values are found in the experimental groups [33].

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

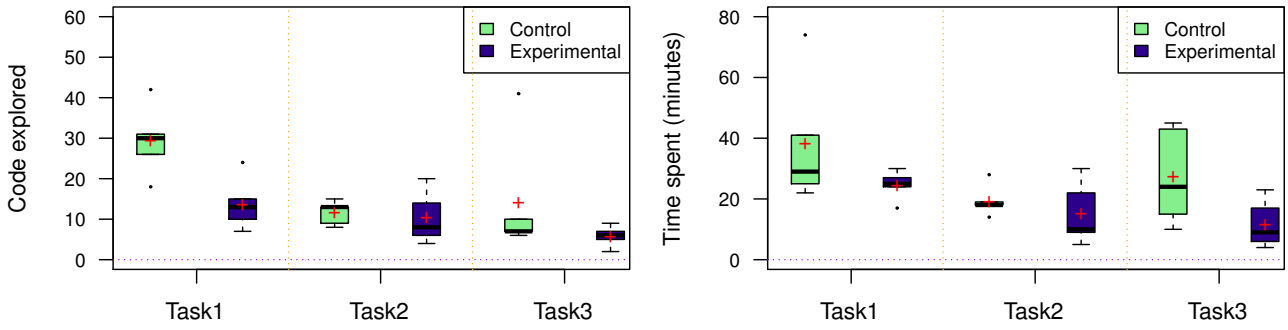


Figure 2. Box plots of the difference in time spent and code explored by both groups by task. The red (+) signs show the mean values.

Code explored. For T1, the mean reduction in code explored achieved using OOGs is 53% (Table VI). The median of the code explored using OOGs is 13, and without it, it is 30 (Table V). The difference is statistically significant with a p-value of 0.008. The median of the code explored in T2 is 8 using OOGs, and 13 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 6 using OOGs, and 7 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]).

Time spent. The mean reduction in time spent achieved using OOGs is 36% for T1, 22% for T2, and 60% for T3. In T1, the median time spent using OOGs is 25 min, and 29 min without. In T2, the median is 10 min using OOGs, and 18 min without. In T3, the median is 9 min using OOGs, and 24 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.

A. *Analysis of Activities, Questions, and Strategies*

We further broke down tasks into activities, following a hierarchical task decomposition [34]. 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 to debugging. 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 VII). 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 IV).

Table V
DESCRIPTIVE STATISTICS FOR CODE EXPLORED AND TIME SPENT.

code explored (classes, interfaces, methods, fields, OOG nodes)						
Task	**C**			**E**		
	Mean	**Median**	**SD**	**Mean**	**Median**	**SD**
T1	29.40	30.00	8.71	13.80	13.00	6.46
T2	11.60	13.00	2.97	10.40	8.00	6.54
T3	14.20	7.00	15.06	5.80	6.00	2.59
time spent (min)						
T1	38.20	29.00	21.28	24.60	25.00	4.83
T2	19.40	18.00	5.18	15.20	10.00	10.43
T3	27.40	24.00	15.98	11.80	9.00	7.98

Table VI
RESULTS OF WILCOXON TEST AND NON-PARAMETRIC EFFECT SIZES.

Task	**p-value**	**mean percent. difference**	**Cliff's delta**	**p-value**	**mean percent. difference**	**Cliff's delta**
	code explored			**time spent**		
T1	0.008	53% reduction	-0.92 large	0.147	36% reduction	-0.4 medium
T2	0.264	10% reduction	-0.24 small	0.232	22% reduction	-0.28 medium
T3	0.068	60% reduction	-0.56 medium	0.0476	60% reduction	-0.68 medium

The participants expressed their need for information as *questions*. 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 VIII). For each activity, we coded the questions involved in it (Table VII). For example, 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).

> *...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. For example,

Table VIII
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

Table VII
CODE UNDERSTANDING ACTIVITIES ATTEMPTED BY ALL PARTICIPANTS IN EACH TASK. 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

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`:

> *...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):

> *...I'm wondering if I can access it [figureMap] from here [GameStub]. 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, but the *strategies* that they used to answer questions about the run-time structure varied based on the source of information.

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 (Tables IX and X). 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);
```

Table IX
CODE EXPLORED IN THE ATTEMPTED ACTIVITIES IN EACH TASK.

Activity	C1	C2	C3	C4	C5	E1	E2	E3	E4	E5
T1.a	11	11	16	6	19	3	3	3	3	3
T1.b	10	4	2	6	15	18	1	3	3	2
T1.c	9	3	1	6	2	1	2	3	4	2
T1.d	-	2	2	-	5	2	6	2	-	-
T1.e	-	11	5	-	1	-	4	4	-	-
T2.a	8	5	1	1	1	1	15	1	1	1
T2.b	-	6	4	7	6	11	2	1	6	2
T2.c	1	1	2	4	1	1	2	1	1	1
T2.d	6	1	1	1	1	1	1	3	1	1
T3.a	4	1	2	7	3	2	5	3	3	5
T3.b	3	34	2	-	2	-	1	2	1	3
T3.c	3	6	3	-	1	-	1	1	1	1

There was some degree of variability in the code explored and time spent in the attempted activities (Tables IX and X). 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.

B. Theory Revisited

To answer questions about the run-time structure, the E group used facts on the OOG that directly answered their questions (Table XI). 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 XII). We provide two

Table X
TIME SPENT IN THE ATTEMPTED ACTIVITIES IN EACH TASK.

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

Table XI
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`

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 XIII). 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 than in the E group. 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 with a question, which they directly answered by referring back to the OOG.

Participants who did not use the OOG used more time consuming strategies to answer their questions about the run-time structure. The participants struggled with questions about the run-time structure during both code *understanding* and *implementation* activities [31, Sec.6.4]. 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 XV and XIV).

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`. 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. 1).

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. 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, 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!*

When the C participants got stuck, they resorted to debugging or refactoring techniques to be able to access the desired objects, which either required extra time or violated the design. 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. 1).

Table XII
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 XIII
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 XIV
FACTS ON THE OOG USED BY THE E GROUP

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

C. Influence of Experience

To determine whether the differences in experience between participants influenced their performance, we performed Analysis Of Covariance (ANCOVA) [35]. 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 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". The summary of results show that neither Java experience nor industry experience had a significant effect on performance. For T3, the total programming and Eclipse experience had significant influence on the code explored but not the time spent (Table XVI).

VIII. Discussion

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 and study design issues. A main contribution of the OOG is the ability to answer Which-A-In-B questions. Even though the OOG helped in answering these questions by providing Part-of/Is-Owned facts (Table XIV), our tasks did not trigger many such questions (Table XIII). 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 is a special case of movement. The results suggest that the time difference increased significantly with T3, presumably since T3 was different. 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.

Our study may have some threats to internal validity. First, the participants had varying experience. As indicated in section VII-C, 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. 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.

Several factors may affect the generalizability of our findings. First, MiniDraw may not be representative of all code bases. Second, our tasks may not be representative of real maintenance tasks such as bugs or feature requests submitted by framework developers. 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.

Table XV
Facts From DCS tools used by the C group

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 XVI
Results of ANCOVA (α=0.05).

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

IX. Conclusion

In this paper, we define a theory in comprehension in terms of facts about the run-time structure that an OOG provides. 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. 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%. There 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.

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. On our end, we are mining the usage data we gathered in this study to enhance our current tool.

References

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

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

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

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

[5] N. Ammar and M. Abi-Antoun, "Adding Ownership Domain Annotations to and Extracting Ownership Object Graphs from MiniDraw," WSU, Tech. Rep., 2011.

[6] 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.

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

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

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

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

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

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

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

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

[15] 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.

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

[17] R. W. O'Callahan, "Generalized Aliasing as a Basis for Program Analysis Tools," Ph.D. dissertation, CMU, 2001.

[18] A. Spiegel, "Automatic Distribution of Object-Oriented Programs," Ph.D. dissertation, FU Berlin, 2002.

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

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

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

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

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

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

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

[26] 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.

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

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

[29] AgileJ, "StructureViews," www.agilej.com, 2008.

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

[31] M. Abi-Antoun and N. Ammar, "Empirical Evaluation of Diagrams of the Run-time Structure for Coding Tasks: a Controlled Experiment," WSU, Tech. Rep., 2012.

[32] 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.

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

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

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