ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/228958468

A case study in evaluating the usefulness of the run-time structure during coding tasks

Article · January 2010

DOI: 10.1145/1938595.1938597

CITATIONS
4

READS
24

2 authors, including:

Nariman Ammar
Wayne State University

14 PUBLICATIONS **34** CITATIONS

SEE PROFILE

All content following this page was uploaded by Nariman Ammar on 05 February 2014.

The user has requested enhancement of the downloaded file. All in-text references underlined in blue are added to the original document and are linked to publications on ResearchGate, letting you access and read them immediately.

A Case Study in Evaluating the Usefulness of the Run-time Structure during Coding Tasks

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

ABSTRACT

Diagrams can help with program understanding and code modification tasks. Today, many tools extract diagrams of packages, classes, associations and dependencies. However, during coding activities, developers often ask questions about objects and relations between objects, i.e., the run-time structure. Most tools that display the run-time structure show only partial views based on running the system. In previous work, we proposed extracting diagrams of the run-time structure using static analysis. In this paper, we investigate whether developers who have access to such diagrams of the run-time structure can perform a code modification task more effectively than developers who have access to diagrams of only the code structure.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*

General Terms

Experimentation, Documentation

1. INTRODUCTION

During coding tasks, developers often utilize diagrams to gain a high-level understanding of the system. Many tools extract class diagrams, however, reverse-engineered class diagrams often fail to explain relations between objects. For example, in framework programming, developers think in terms of the responsibilities of objects [12], interactions that a diagram of the run-time structure can depict. Some tools display the run-time structure as partial views of the system, based on running and monitoring the system [18]. We provide developers with diagrams of the run-time structure, extracted using static analysis without running the system.

The contribution of this paper is two-fold. First, we provide further empirical evidence that developers ask questions about object relations. Second, we present a preliminary case study on a developer doing code modification using diagrams of the run-time structure in addition to diagrams of the code structure. We observed that the developer benefited from the diagram of the run-time structure, and performed the task more effectively than another developer who did the same task using only diagrams of the code structure. To our knowledge, this is the first study that uses both types of diagrams in the context of a code modification task.

Outline. This paper is organized as follows. In Section 2, we give some background on the approach. Next, in Section 3, we describe the study's method. In Section 4, we describe our results. In Section 5, we discuss validity and future work. Finally, we discuss related work in Section 6 and conclude.

2. BACKGROUND

Before we discuss the study, we give some background on our approach and compare it to other approaches that rely on diagrams of the code structure. Reverse-engineering tools extract diagrams of packages, classes, associations, and dependencies. Packages are layers in the code structure, and do not necessarily reflect the run-time structure of the system, which is often partitioned into run-time tiers, e.g., "User Interface", "Logic", and "Data".

Previous work studied developers while they perform code modifications based on the code structure, such as class diagrams [14, 11]. In our approach, we provide developers with diagrams of the run-time structure and ask them to do code modifications based on those diagrams in addition to diagrams of the code structure. We use the SCHOLIA approach [1] to extract the diagram of the run-time structure statically, where architectural extractors add annotations to the original object-oriented Java code. These annotations specify within the code: object encapsulation, logical containment, and architectural tiers, which are not explicit constructs in general-purpose programming languages. A static analysis then scans the annotated program's abstract syntax tree and produces a hierarchical object graph, the Ownership Object Graph (OOG).

Figure 1 shows an OOG. The OOG, is composed of nodes and edges that show the interactions between these nodes. A node can be either an object, represented by a solid box, or a domain within an object, represented by a box with dashed border. Each object node has a parent domain, and each domain node has a parent object. Edges between objects, represented by solid arrows, correspond to points-to relations that show how these objects are related. Edges on the OOG are linked to field declarations in the source

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HAoSE '10 Reno, NV, USA
Copyright 2010 ACM 978-1-4503-0543-3/10/10 ...$10.00.

code. Hierarchical representation and the associated ability to expand or collapse elements have been shown to be effective for software architecture [16]. We provide developers with this capability by providing them with a viewer tool to interactively navigate the OOG where they can expand some of the collapsed objects. The plus symbol in Figure 1 indicates that an object has sub-structure.

SCHOLIA often requires the architectural extractor to refine the annotations until the extracted OOG reflects the architectural intent. However, the ownership type system keeps the architectural extractor honest that he cannot tweak the OOG in such a way as to make it more useful than it really is. He can only push an object underneath another object or merge objects that have related types, i.e., use abstraction by ownership hierarchy, and optionally, abstraction by types. For this study, the architectural extractor added annotations to DrawLets, and extracted OOGs for the benefit of a developer performing code modification tasks. Due to space limits, the details of the annotation process and OOG refinement are in a technical report [3].

3. METHOD

We conducted a month-long case study during which a developer attempted a code modification task using both class diagrams and OOGs. In the rest of this paper, the "participant" is the developer who was performing the code modification, and this paper's second co-author. The "architectural extractor" is the person adding annotations and extracting OOGs, and this paper's first co-author.

Study Design. We used the DrawLets subject system [8], an open source framework (version 2.0, 115 classes, 23 interfaces, 12 packages, 8,000 lines of code). The code was still being annotated during the study and the participant was provided with several extracted OOGs each showing different pieces of information reflecting the participant's mental model of the system as she was exploring the code. Also, we wanted to evaluate the effects of the OOG, rather than the effects of the annotations in the code, so we provided the participant with the same copy of the DrawLets code as the architectural extractor, but with the annotations suppressed. Also, developers operating under strict deadlines often make the most expedient changes, even if they violate the architecture. To avoid this problem, we gave the participant ample time to read and understand the task description and the system, then perform the code modification.

The Subject System. DrawLets supports a drawing canvas that holds figures and lets users interact with them. The figures include lines, rectangles, polygons, etc. The architectural extractor organized the core types in DrawLets into two top-level tiers: the `MODEL` tier and the `UI` tier each containing instances of the core types as follows:

- `MODEL`: has instances of `Drawing` and `Figure` objects (Fig. 1). A `Drawing` is composed of `Figure`s that know their containing `Drawing`. Tools are `InputEventHandler`s that act on drawing canvases and modify the figure attributes, such as size and location. Tools implement the `CanvasTool` interface. A `SimpleDrawingCanvas` adds `Figure`s to a `Drawing` and implements the `DrawingCanvas` interface.
- `UI`: has an instance of `SimpleModelPanel`. The `SimpleModelPanel` class implements the AWT `Panel` interface. In DrawLets, a `DrawingCanvas` can be part of a larger application, and needs a GUI-specific placeholder in order to be able to reside within the application's GUI. `DrawingCanvasComponent` allows a `DrawingCanvas` to reside within an AWT application.

Participant. The participant was a graduate student in computer science. She had good knowledge of frameworks, design patterns, and UML. She had good Java programming skills, and was familiar with the Eclipse navigation features. The participant had previously helped analyze data from a study on the OOG of JHotDraw [4]. JHotDraw and DrawLets are closely related as they both descend from the HotDraw Smalltalk framework for implementing drawing applications [14]. She had also received classroom instruction on the annotations and the static analysis for architectural extraction, but was not involved with the process of adding the annotations or extracting OOGs.

Architectural Extractor. The architectural extractor added annotations to the code, ran the static analysis to extract OOGs, and fine-tuned the extracted OOGs to reflect the participant's mental model. He provided the participant with the diagrams both as XML files (to be loaded into the viewer) and PDF files (to be viewed or printed). He was one of the developers of SCHOLIA and the tools to extract runtime views from a system. However, he did not contribute to the code modification.

Tools and Instrumentation. The participant used the Eclipse IDE (Version 3.5), with the OOG viewer plugin installed. The participant used the viewer only to view the OOGs, and could not edit them directly. The OOG viewer has a modeless dialog, which enabled her to display the OOG, while she was concurrently editing the code in Eclipse. For example, she was able to trace from an element on the OOG to the corresponding line of code in Eclipse text editor. The tool also displays a partial class diagram showing the inheritance hierarchy of a selected group of objects. A snapshot of the tool navigation features appear in the technical report [2], and include the following: collapse/expand sub-structures, search the ownership tree, find a label in the OOG, and other standard operations such as zoom in/out, pan and scroll. We also provided the participant with a description of the common patterns used in DrawLets [8]. Finally, the participant had access to two manually generated UML class diagrams from the previous case study [14]. One diagram displayed the core interfaces in DrawLets and their relations, the other showed the top-level classes and their dependencies. Due to space limits, the class diagrams are relegated to the technical report [2].

Task. The code modification task was to *implement an "owner" for each figure*: "An owner is a user who put that figure onto the canvas, and only the owner is allowed to move and modify it. At the beginning, each session declares a session owner, and this session owner will own all new figures created in that session. No other user will be allowed to manipulate them. At the beginning of a session, user inputs ID and password. Any function that attempts to modify a figure must check that the figure owner and the current session owner are same..." [14].

Procedure. We gave the participant ample time to understand the task description and implement the necessary changes, and did not impose any timing constraints. We asked her to capture a time log of the different activities in which she was engaged as well as the thought process, to

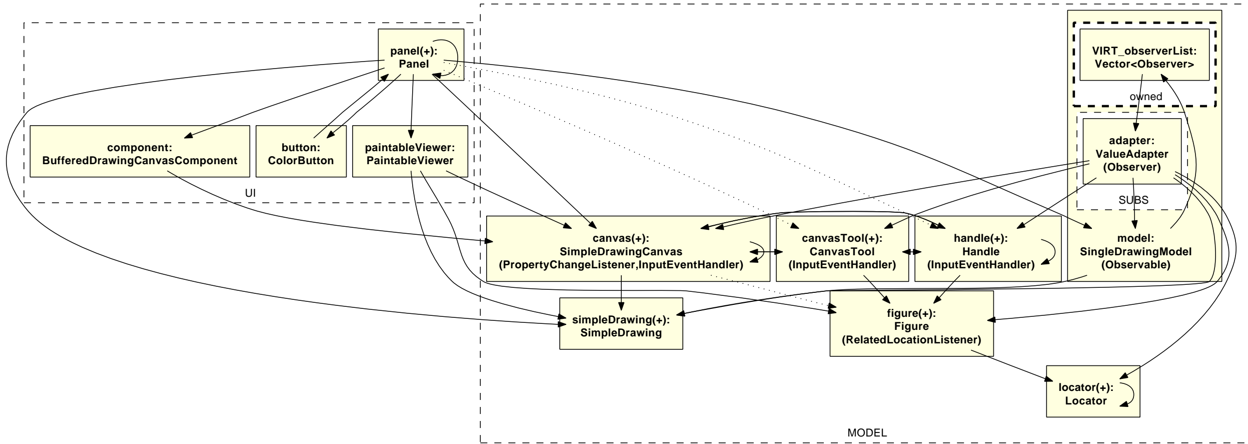

Figure 1: Ownership Object Graph (OOG) of DrawLets.

simulate the think-aloud protocol.

The participant spent around 20 hours brainstorming, navigating and exploring the DrawLets source code to determine where to modify the code, recording her thought process, studying the extracted diagrams, coding, and testing the modifications. The participant recorded the thought process in the form of a transcript consisting of the questions that she had, the classes that she visited, and which feature in Eclipse or in the OOG viewer tool she was using. The participant recorded the transcripts manually and did not use any screen capture or video recording because of the long running study.

Analysis. We analyzed the participant's thought process from the transcripts using a qualitative protocol analysis [17]. For the protocol analysis, we reused a model that we had defined previously to code the types of questions about object structures that developers ask during coding activities [4]. Among the things that we analyzed was the the different classes that the participant visited, the number of visits per class, the number of times she used Eclipse features, which diagrams she used, and how many times.

4. RESULTS

In this section, we discuss the changes that the participant implemented. We then report our observations supported with evidence from the transcripts on how the developer used the OOG and the tool to answer some of her questions about object structure.

4.1 Code Changes

In performing the task, the participant added two classes: a `LoginDialog` class to enable users to login to the system and a `SessionOwner` class which saves information about the current session. She also added a "New Session" button to the command panel inside `SimpleModelPanel` (or `SimpleApplet`) to enable the user to launch the login dialog and added the corresponding `launchLoginDialog()` method inside `SimpleDrawingCanvas` class. She also added the list of owners to both `SimpleDrawingCanvas` and `AbstractFigure`. To check the owner of a figure, she added `isOwner()`and `isExisting()` methods to the `SimpleDrawingCanvas` class. Finally, the participant added a call to the `isOwner()` method inside any method that attempts to change a figure attribute including `SelectionTool` and `SimpleDrawingCanvas` classes.

Several actions can modify a figure in DrawLets, including changing the figure's position, size, text or color, or linking it to another figure. The participant tested only three use cases: a user who tries to remove a figure which he does not own, a user who tries to move a figure which he does not own, and a user who tries to resize a figure which he does not own. In all three cases, she prompted the user to enter the correct ID and password. The three cases above required her to add the `isOwner()` method to each of the following methods: `SimpleDrawingCanvas.removeFigure()`, `SelectionTool.mouseDragged()`, and she was still looking for the method responsible for resizing the figure through its handles, but she thought that would be related to either `BoundsHandle.resize()` or `SimpleDrawingCanvas.mouseDragged()`.

The participant believed she fulfilled the task, even though she could have checked more actions such as changing a figure's color or text, or linking two figures. The modification done in the previous study [14] was also incomplete for the same reason. Still, the participant's modification was better in two respects: she covered more cases than in the previous case study, and she did not add any listener classes or interfaces, `SimpleListener`, like the developer in the previous study. We believe these code modifications were sufficient for the purpose of the study, and we explain how the OOG helped the participant stay within the design.

4.2 Observations

Developers need High-level views to understand the system. When the participant started modifying the code, she relied mainly on Eclipse's navigation features, since she was working in areas of the code where she did need a diagram. Later on, she wanted to get a high-level understanding of the system, and the class diagrams helped her see at a glance the main classes and interfaces and how they inherit from or implement each other. According to the transcripts, the participant looked at the class diagram showing the core interfaces in DrawLets, but she did not use it since she could get the same information using Eclipse call graphs. She preferred to use the other diagram showing the dependencies between different classes in DrawLets and she referred to that diagram twice. Then she refered to the extracted OOG and kept requesting updates on it. She also used the OOG viewer and found certain features to be more useful than others. We list the tool features that she used and the fre-

quency of usage in the technical report [2].

Developers use the traceability links effectively in relevant tasks The participant found useful the feature of tracing from an element on the OOG to the code. Instead of seeing a relationship on a diagram and then using the "Open Type" command in Eclipse, the OOG Viewer tool helped the participant trace directly from the OOG to the code. For example, the participant highlighted the edge between `panel:SimpleModelPanel` and `model:SingleDrawingModel` which linked her to the corresponding field declaration in the Eclipse text editor using the tool. The participant also able to see a tree representation of the OOG, where she could search for an object in the ownership tree by type or field name. She was also able to search for all the incoming or outgoing edges of a selected object and choose the relation that she was looking for and go to the specific line of code associated with this relation. The participant relied heavily on this feature during the study. For example, she could search for all the possible relations associated with the `figure:Figure` object.

The participant also learned useful information by exploring the edges between two objects. For example, she was searching for the method that was responsible for changing the figure's position. The edge between `figure:Figure` and `canvasTool:CanvasTool` took her to `SelectionTool.mouseDoubleClicked()` using the trace to code feature. She found that this method instantiates a `figure` object which gets its value from `canvas.figureAt(x,y)` and dug deeper into this method. This eventually led her to `getTool().mouseClicked()` inside `SimpleDrawingCanvas` which gave her an indication that she could do the modification either inside `SimpleDrawingCanvas` or `SelectionTool` especially since the diagram shows that both implement the `InputEventHandler` interface.

Developers can use the OOG to understand and respect the architectural intent. The participant was aware that she should add new classes, methods, and fields in a way that fit the design of the system. The instances that appeared in the top-level domains helped her understand better how different objects are related in DrawLets (Fig. 1). Because she understood the process of extracting OOGs, she knew where on the extracted diagram the objects that she added might appear.

Once the participant started thinking in terms of architectural tiers, she refactored the code that she had already added. For example, she was confused whether she should add the session check to `SimpleApplet` or `SimpleDrawingCanvas`. Presumably, calling the method to check a figure's owner should be done inside the `UI` tier, but the implementation of the checking logic itself should be inside the `MODEL` tier. That is why she eventually moved the `isOwner()` method to the `SimpleDrawingCanvas` class.

During the study, the participant was looking for the best way and location to modify the code. After several attempts and after visiting several classes, she added a button to the `canvas`. After she started using the diagram, she realized that the diagram could have saved her from visiting many classes. She noticed that she could have highlighted the edge between `panel:Panel` object and its nested object `commandPanel:CommandPanel` and trace to code to get to `SimpleModelPanel` class, where she could clearly see a list of other commands. Having realized this fact, she knew exactly how the different commands are handled in DrawLets. She noticed a certain pattern for dealing with actions in the `actionPerformed()` method. However, the method she added did not initially follow that pattern. As a result, she moved the `launchLoginDialog()` method from the `SimpleDrawingCanvas` class to the `SingleDrawingModel` class instead, to be consistent with the design.

This led us to wonder how the participant was able to violate the pattern described in the previous section or if DrawLets really respected the two-tiered style. The participant said that she was able to get hold of the `canvas` object inside `SimpleModelPanel` or `SimpleApplet` and did not need to go through `SingleDrawingModel`. This means that DrawLets does not strictly follow the two tier architectural style where objects in the `VIEW` tier should not have direct references to objects in the `MODEL` tier.

Developers can use the OOG to understand how the program implements some design patterns. The participant was able to understand the observer design pattern by looking at the OOG in Figure 1. She was able to see the `(Observer)` and `(Observable)` labeling types on `adapter:ValueAdapter` and `model:SingleDrawingModel` respectively. So, she understood that the `adapter` listens to updates passed through the `model`. She also noticed the `VIRT_observerList` which is a vector of observers inside the public domain `SUBS` inside `model`. When she highlighted the edge between `model` and `adapter` instances and traced to the code, she found that `SingleDrawingModel` class instantiates an `adapter` object and passes it the `model` and `canvas` objects as parameters. The diagram helped her understand these relations only partially, so she browsed the code to understand the implementation details.

The participant found the above information useful, and since she was looking for the method that was responsible for moving a figure, she wanted to see if the figure was being notified using this pattern. She found that `valueAdapter:ValueAdapter` points to `figure:Figure`, and tracing to the code, she navigated to the `target` object which was of type `Object`. This result led her to another observation that DrawLets could be treating the drawing, the figure, and the canvas as observers. She browsed the code and found that DrawLets uses the observer design pattern to handle actions on `Drawing` objects. This finding helped her understand that figures are not handled using this pattern but using listener interfaces. The `RelatedLocationListener` labeling type on the `figure:Figure` (Fig. 1) object that appeared on the diagram confirmed her understanding.

5. DISCUSSION

Our study, like other empirical studies, might be exposed to threats of validity. However, the qualitative nature of our analysis makes these threats more manageable.

To avoid a strawman argument in assessing the usefulness of an OOG, one might argue that we should have provided developers with more helpful class diagrams, that were refined just as the OOGs were evolved to reflect the participant's mental model. To mitigate this threat, we did not provide developers with automatically reverse-engineered class diagrams, which are often neither abstract nor precise representations of source code, and are of little interest to software engineers [10]. Instead, we provided the par-

ticipant with two carefully crafted class diagrams that were used in the previous case study on DrawLets. Still, a class diagram alone often could not answer some of the questions that the participant asked, such as which object instances are distinct and how to get to certain objects from a given object. In contrast, the OOG do show such relations, and the object labels often reference the types that appear on the class diagram. In other words, the OOG contained sufficient information about the static code structure, but the reverse is often not the case.

Given that our control group was a developer who performed the same task nearly a decade ago, the development environment used then was probably not identical to ours. Still, the technologies of call graphs and UML class diagrams were already mature then. As a result, our participant did not use more advanced features than those available at the time, with the exception of the OOG and the viewer tool.

In the real world, code modification tasks on real software systems are performed under strict deadlines. For this study, we did not impose any timing constraints, so this may reduce the external validity of the result. However, considering that the participant was using a new type of tool, and relied on the architectural extractor to refine the OOG, the lax timing could be justified. Also, the participant recorded her thought process manually, which could have interfered with the natural way of programming. This is a common issue in lab studies, in which developers risk losing some of their natural behavior.

An obvious threat is that we conducted the study with a single participant, a graduate student working in a lab, which may not be representative of experienced professional developers working on real systems. This is a case study, and case studies often rely on analytic generalization rather than statistical generalization [19, p. 43]. We are also planning to replicate our findings in another case study. Also, we might have been biased by choosing a participant who had previous knowledge of the tool and the approach. However, since the purpose of the study is to investigate the usefulness of the OOG, we chose a participant who did not require extensive training to avoid much of the learning curve involved in adopting a new tool and diagram, as is often desired for high-quality case studies [19, p. 68].

Other factors could have helped the participant understand the system. She had many sources of information including online help, textual documentation, the results of a previous DrawLets case study, familiarity with a similar system (JHotDraw), some domain knowledge, and Eclipse debugging skills. Although the participant was familiar with JHotDraw, she never modified that code. We also believe that a general knowledge of design patterns and frameworks contributed more to performing the modification than did the familiarity with the code itself. Also, the purpose of the study is to demonstrate that the information gained from the run-time structure is complementary to the information from other sources.

To achieve construct validity, we tried to avoid any potential bias by using a code modification task designed by others, rather than one we specifically designed to make the extracted diagrams appear more useful than they really are. Also, to prove the claims stated in this study, we provided multiple sources of evidence. We previously conducted a study to investigate whether developers ask questions about object relations during coding activities [4] which involved three developers using the run-time structure to do multiple code modification tasks. Our results were promising, so we conducted this study to investigate whether the run-time structure can help answer some of the questions asked by developers about object relations. We are also planning to seek a more compelling evidence by conducting more case studies according to the replication approach for multiple-case study design [19].

There could be some drawbacks due to the approach itself. For the diagram to reflect the entire system, the entire code must have annotations that typecheck. Adding annotations is currently a manual step. As a result, we did not annotate the code that the participant added or modified, to shield her from the annotation process. Admittedly, the architectural extractor could have added or updated the annotations, but this would have required the participant and the extractor to work in tandem. As a result, the participant kept working with an OOG that did not reflect the additional code and was working on a copy of the source code without annotations. Also, the current approach has the flexibility to respond to the developers requirements such as moving objects up or down in the hierarchy by refining the annotations and the extracted OOGs. However, the participant was not involved in the extraction process, and was not able to move objects using the tool. Perhaps, the ability to interactively refine an OOG during a code modification task might make the diagram more useful. This is a capability for which we are currently developing tool support [5].

Finally, we mentioned several times in this paper that we asked the developer to do the same task performed previously using the code structure [14]. Rajlich and Gosavi proposed a technique for unanticipated incrementql change that used programming concepts, and required knowledge of class dependencies. They applied this technique to DrawLets and observed that the design of DrawLets did not help localize the change that they made, so they used techniques such as refactoring to limit the length of change propagation. In this study, we observed that a developer who uses the OOG may modify fewer classes. However, we do not claim that using OOGs for code modification limits the length of change propagation, since modifiability seems to be a quality attribute of the software itself. Moreover, DrawLets seems to have been designed by professional object-oriented programmers. Still, we found a few places where the DrawLets code did not follow the best practice of using type safe declarations. The code also includes a few hacks when dealing with reflection. As a result, several casts may fail with runtime exceptions. Moreover, the use of reflective code poses challenges for the ownership annotations and the static analysis (the entities that the analysis may not understand must be manually summarized using virtual fields in order to preserve the soundness of the extracted diagrams). Therefore, through the process of annotating the subject system the architectural extractor had to refactor the code to fix some problematic code patterns and be able to add annotations easily. We discuss some of the architectural extractor's efforts to refactor the code in a technical report [3].

6. RELATED WORK

Our previous evaluation of the run-time structure. We previously conducted a field study to help us understand how developers understand object relations, and what tool

features they need to convey their mental model of the system [6]. In that study we provided a professional developer with an initial run-time architecture and refined it to convey his intent, but we did not use the refined diagram to do a code modification task. Prior to this study, we conducted an exploratory study to identify whether developers ask questions about object relations during coding tasks [4]. Our findings confirmed that developers do ask questions about object relations such as: points-to, is-owned, and may alias questions. In this study, we provided the developer with several extracted diagrams based on her evolving mental model and we did not consider any specific diagram to be the authoritative one. Since the focus of this study is on the usefulness of the run-time structure, we tried to get a diagram that reflects more the developers mental model to be able to answer their questions about object relations.

Previous studies on the run-time structure. Walker et al. [18] developed an approach for visualizing the operation of an object-oriented system at the architectural level. Their approach builds on the Reflexion Models technique, but uses the running summary model rather than the complete summary model. They allow developers to flexibly define the structure of interest, and to navigate to the resulting abstracted views of the system's execution. Approaches that rely on static information can often rely on the iterative mapping approach, and their approach relied on dynamic information which limits iteratively updating the mapping. Richner et al.[15] proposed a complementary approach to Walker's work that uses both static and dynamic information to answer developers questions about object oriented code. Their study focused on reverse engineering HotDraw and trying to understand it, but did not involve any code modification task.

Previous studies on diagramming tools. Several studies have been conducted about the contribution of diagramming tools in program comprehension [7, 9, 13]. Hadar et al. [11] conducted a study on developers comprehension of UML diagrams. Their study focused on how developers use several types of UML diagrams for program comprehension. They found that developers often need all types of UML diagrams and integrate the information they get from each one to understand and analyze the program. They also found that developers even sort diagrams by the type of information they can get from them such as using sequence diagrams to understand the dynamic behavior and class diagrams to study static relations. These studies focused on UML diagrams, and none of them used the runtime structure by statically analyzing the code to do a code modification task even though previous work [11, 7] used partial runtime views such as sequence diagrams.

7. CONCLUSION

We conducted a case study which serves as further empirical evidence that developers do benefit from having access to diagrams of the run-time structure, to answer some of their questions about object structure. We believe these results are promising even though the study involved only one developer. This study confirmed some usability challenges in the current tool, which may lower the usefulness of the diagram. Once we address the issues both in the approach and in the tool, we are planning to seek stronger empirical evidence by conducting more controlled experiments.

8. REFERENCES

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

[2] M. Abi-Antoun and N. Ammar. Usefulness of the run-time structure during coding tasks. Technical report, WSU, 2010.

[3] M. Abi-Antoun, N. Ammar, and F. Khazalah. A Case Study in Adding Ownership Domain Annotations. Technical report, WSU, 2010.

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

[5] M. Abi-Antoun and T. Selitsky. Interactive Refinement of Runtime Structure. In *FlexiTools*, 2010.

[6] M. Abi-Antoun, T. Selitsky, and T. LaToza. Developer Refinement of Runtime Architectural Structure. In *SHARK*, 2010.

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

[8] DrawLets Case Study: Online Appendix. `www.cs.wayne.edu/~mabianto/drawlets/`, 2010.

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

[10] Y.-G. Guéhéneuc. A systematic study of UML class diagram constituents for their abstract and precise recovery. In *APSEC*, 2004.

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

[12] D. Lange and Y. Nakamura. Interactive Visualization of Design Patterns Can Help in Framework Understanding. In *OOPSLA*, 1995.

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

[14] V. Rajlich and P. Gosavi. A Case Study of Unanticipated Incremental Change. In *ICSM*, 2002.

[15] T. Richner and S. Ducasse. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. In *ICSM*, 1999.

[16] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration. *J. Systems & Software*, 44(3), 1999.

[17] A. Strauss and J. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 1998.

[18] R. J. Walker, G. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing Dynamic Software System Information through High-Level Models. In *OOPSLA*, 1998.

[19] R. K. Yin. *Case Study Research: Design and Methods*. SAGE Publications, 4th edition, 2009.

View publication stats