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.

1. INTRODUCTION

During coding tasks, developers often utilize diagrams to gain a high-level understanding of the system. However, reverse-engineered class diagrams often fail to explain many of the system’s implementation details such as relations between objects and design patterns. For example, in framework programming, developers think in terms of the responsibilities of the objects or what they are supposed to do [17], 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 [25].

The contribution of this paper is two-fold. First, we provide further empirical evidence that developers ask questions about object relations. Second, we demonstrate how a diagram showing the run-time structure of the code can help developers answer some of these questions. In our approach (Fig. 1), we provide developers with diagrams of the run-time structure of the Java code, extracted using static analysis without running the system. We conducted a preliminary case study in a lab setting, where we asked a developer to work on a code modification task, adopted from a previous a case study by Rajlich and Gosavi [20]. We provided the developer with diagrams of the runtime 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 code modification task more effectively than another developer who had access to only diagrams of the code structure. To our knowledge, this is the first study of a code modification task, that uses both types of diagrams.

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. Current reverse-engineering tools help developers 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 (Fig. 2).

Figure 2 shows the extracted Ownership Object Graph (OOG) of the DrawLets system [2]. The OOG is a graph that is composed of nodes and edges that show the interaction between these nodes. A node can be a domain or an object. Domains are represented by white boxes with dashed borders, and objects are represented by solid boxes. Edges between objects correspond to field references and appear as solid arrows (Fig. 2).

Previous work studied developers while they perform code modifications based on diagrams such as class diagrams or Eclipse call graphs [20, 22, 18]. In our approach, we provide developers with diagrams of the run-time structure...
and ask them to do code modifications based on those diagrams (Fig. 1). In order to extract the diagram statically, architectural extractors add annotations to the original object-oriented Java code. The annotations implement an ownership type system, and 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 which abstracts objects by both ownership hierarchy and by types, the Ownership Object Graph (OOG).

For instance, a hierarchical representation and the associated ability to expand or collapse elements has been shown to be effective for software architecture [23]. The process often requires refining the annotations until the extracted diagram reflects the architectural intent. The dashed border in Figure 1 illustrates the refinement of the extracted OOG based on the developer's evolving mental model of the system [8].

For this study, the architectural extractors added annotations to DrawLets, and extracted diagrams of the run-time structure for the benefit of a developer performing code modification tasks. Due to space limits, the details of the annotation process are in a technical report [4].

3. METHOD

We conducted a month-long case study during which a developer attempted a code modification task using both a class diagram and a diagram of the run-time structure. In the rest of this paper, we refer to the “participant” as the developer who was performing the code modification task, and this paper’s second co-author. We use the term “architectural extractor” to refer to the person who was adding the annotations and extracting the diagrams of the runtime structure, and this paper’s first co-author.

Study Design. For the study we used the Draw Lets subject system [9], 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 diagrams each showing different pieces of information reflecting the participant’s mental model of the system as she was exploring the code base. We wanted to evaluate the effects of the diagram, rather than the effects of the annotations in the code. To avoid introducing an additional confound to the study, the participant used the same version of the Draw Lets 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 Draw Lets system, then to perform a code modification task designed by Rajlich and Gosavi in a previous case study [20].

The Subject System. Draw Lets supports a drawing canvas that holds figures and lets users interact with them. The figures include lines, rectangles, polygons, ellipses, etc. Many applications can be organized into three tiers: User Interface, Logic and Data tiers. For simplicity, we organized the core types in Draw Lets 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. 2). A Drawing is composed of Figures that know their containing Drawing. The class SimpleDrawing implements the Drawing interface. Tools are InputEventHandlers that act on drawing canvases and modify the figure attributes, such as size and location. They implement the CanvasTool interface. A SimpleDrawingCanvas adds Figures to a Drawing and implements the DrawingCanvas interface.
- UI: has an instance of SimpleModelPanel. The SimpleModelPanel class implements the AWT Panel interface. In Draw Lets, 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 and design patterns, good Java programming skills, familiarity with the Eclipse navigation features, and good knowledge of UML. She was also familiar with similar applications such as JHotDraw [16] since she helped analyze data from a previous study on the run-time structure of JHotDraw [5]. JHotDraw and Draw Lets are closely related since they both descend from the HotDraw Smalltalk framework for implementing drawing applications [13].

The participant was not involved with the process of adding the annotations to Draw Lets or extracting OOGs. She was provided with a tool to view the extracted OOGs, and could not edit them directly. She had also received classroom instruction on the annotations and the static analysis for architectural extraction.

Architectural Extractor. The architectural extractor added annotations to the Draw Lets code, ran the static analysis to extract OOGs, and fine-tuned the extracted OOGs. 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 the approach and the tools to extract run-time views from a system. However, he did not contribute to the modifications to the Draw Lets code.

Tools and Instrumentation. The participant used the Eclipse IDE (Version 3.5), with the OOG viewer plugin installed. The OOG viewer has a modeless dialog (Fig. 3), which enabled her to display the run-time structure of Draw Lets, while she was concurrently editing the code in Eclipse. For example, she was able to trace from an element in the diagram to the corresponding line of code in Eclipse. The tool also displays a partial class diagram showing the inheritance hierarchy of a selected group of objects. The navigation features of the OOG viewer appear on the right hand side and left hand side of the viewer and include the following features: collapse/expand sub-structures, search the ownership tree, find a label in the diagram, and other standard operations such as zoom in/out, pan and scroll. The run-time structure appears in the middle. We also provided the participant with a description of the common patterns used in Draw Lets [10]. The participant had access to two manually generated UML class diagrams from the previous case study [20]. One diagram displayed the core interfaces in Draw Lets and their relations, the other showed the top-level classes and their dependencies. Due to space limits,
Figure 2: Ownership Object Graph (OOG) of DrawLets.

Figure 3: The OOG viewer tool used during the study.
the class diagrams are relegated to the technical report [3].

**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. This change will allow the DrawLets framework to be used as a document collaboration tool” [20].

**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 the participant to capture a log of all the different activities in which she was engaged. We also asked her to carefully record a log of her thought process, to simulate the think-aloud protocol while sharing a lab with others.

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 itself she was using. The participant recorded the transcripts manually and did not use any screen capture or video because of the long running study. The study material are available online [11].

**Analysis.** We analyzed the participant’s thought process from the transcripts using a qualitative protocol analysis [24]. 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 [5].

We tracked the classes that the participant visited, and the number of visits per class. We also noted the features in the OOG viewer that she used and the frequency of usage (Table 1). We also counted the number of times she used Eclipse features including debugging, call graphs, and grep search. According to the transcripts, the participant referred to the class diagram twice during the study and found one version of the diagram more useful than the other.

4. RESULTS

In this section, we discuss the changes that the participant implemented. We then report our observations, and support them with evidence from the transcripts on how the developer used the run-time structure and the tool to answer some of the questions that she had about object relations.

**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 made the necessary changes to do the task, even though she could have checked more actions such as changing a figure’s color or text, or linking two figures. The participant assumed that the modification done in the previous study [20] was also incomplete for the same reason. Still, she believed her modification was better in two respects: she covered more cases than in the previous case study, and she did not add a class, `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 diagram helped the participant stay within the design. So next, we list our observations and support them with evidence from the transcripts.

**4.2 Observations**

Developers can impose a conceptual hierarchy on objects, to make the run-time structure match their mental model of 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. Then, to see more interesting relations, she referred

---

<table>
<thead>
<tr>
<th>Ref. no.</th>
<th>OOG viewer feature</th>
<th>Frequency of usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Search Ownership Hierarchy</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Trace To Code</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>Find Object Label</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Object Types</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>Show/Hide Internals</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>Highlight Edges</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>Edge Labels</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Inspect Types</td>
<td>1</td>
</tr>
</tbody>
</table>
to the diagram of the run-time structure and kept requesting updates on the diagram. We provide snapshots of the class diagrams and the details of run-time ownership object graph (OOG) refinement process in the technical report [3].

The participant found certain versions of the diagram of the run-time structure to be more useful than others. The initial OOG had many architecturally irrelevant objects at the top-level tiers, which made it more cluttered and thus less useful. For example, the initial diagram displayed the object toolbar:ToolBar which she believed was not worth showing. Whereas the canvasTool:CanvasTool object, which seemed architecturally important, was missing from the extracted OOG. Similarly, she wondered why the polygon:Polygon object appeared at the top level by itself or even as a nested object when there are other shapes such as rectangles and ellipses. Also, the participant compared the button:ColorButton object to the canvasTool:CanvasTool object which, as the name suggested, represented all types of tools, whereas button:ColorButton did not represent all buttons. Based on this observation, she considered the button:ColorButton object to be too low-level to appear in a top-level domain and that it should be either nested or replaced by a more abstract one. The architectural extractor updated the diagram to match the participant’s mental model [3].

When the participant expanded some of the objects to display their substructures, using the feature to collapse or expand object substructures, she found that the substructures were not showing any interesting relations other than field declarations. From her previous experience with the run-time structure of JHotDraw [5], she knew that the diagram could display interesting object relations such as listeners and design patterns. So she wanted to see if the diagram could help her understand some of the design patterns in DrawLets. The architectural extractor provided the participant with a newer version of the diagram that highlighted better how the Observer design pattern was implemented (Fig. 2). We discuss how the diagram helped interpret this design pattern below.

Finally, the participant was aware of the important interfaces and wanted to see those interfaces on certain objects. So she asked the architectural extractor to set additional labeling types. For example, the SimpleDrawingCanvas implements the PropertyChangeListener interface and the CanvasTool is an InputEventHandler (Fig. 2). With those additional labels, she no longer needed to go, as often, to the Eclipse Type Hierarchy to determine which classes implement a given interface.

The run-time structure helps developers link the graphical representation of the program to textual views of the source code. The participant found useful the feature of tracing from an element in the diagram 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 diagram to the code. For example, the participant highlighted the edge between panel1:SimpleModelPanel and model:SimpleDrawingModel which linked her to the corresponding field declaration in the Eclipse text editor (Fig. 3).

The tool supports searching for an object in the ownership tree by type or field name. Developers can also search for all the incoming or outgoing edges of a selected object and choose the relation that they are looking for and go to the specific line of code associated with this field declaration. 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 getTool1().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.

The run-time structure may help a developer 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. 2). 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 debating where to save the session information to avoid prompting users for their username and password after they logged in to the system. She then realized that she should move the saveSessionInfo() method call to the SimpleDrawingCanvas class. She was also 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.

The run-time structure helps a developer understand how the program implements some design patterns. DrawLets uses the Observer design pattern where SimpleDrawingModel is the subject being observed and SimpleDrawingCanvas is the observer notified of the change (Fig. 2). The diagram showed that the ValueAdapter (observer) listens to updates passed through the subject (SimpleDrawingModel) which has a list of observers VIRT_observerList. When she highlighted the edge between model:SimpleDrawingModel and adapter:ValueAdapter instances and traced to the code, she found that SimpleDrawingModel 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. 2) object that appeared on the diagram confirmed her understanding.

The run-time structure may help prevent introducing bugs in the existing code. In the beginning of 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 so many classes. She noticed (accidentally, and while she was looking for something else) that she can highlight 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:

```java
if (commandString.equals("Save"))
    getModel().saveDrawing(TempFileName);
else if (commandString.equals("New Session"))
    ((SimpleDrawingCanvas) this.canvas).launchLoginDialog();
```

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 SimpleDrawingModel 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 SimpleDrawingModel. 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.

4.3 Limitations

The experiment identified a few cases where the diagram of the run-time structure was not as useful as it could be.

Object labels can be misleading. When the architectural extractor generated the diagram, he used the feature to control the object labels to choose the object labels. Even then, some labels remained unclear to the participant.

Abstraction by types may cause confusion. At one point, the participant believed that the feature of further merging objects by their related types (abstraction by types) was hiding some useful information. For example, the diagram was showing the panel:Panel object, and when the participant was trying to navigate to the code from this object using the trace to code feature, the viewer took her to a different class each time. For example, the edge between panel:Panel object and the nested commandPanel object in its owned domain took her to SimpleModelPanel, whereas the edge between panel:Panel and button:colorButton took her to stylePalette:StylePalette. The participant considered showing Panel alone instead of instances of all different subtypes of Panel to be not so useful, especially since Panel is considered part of Java AWT. To address this concern, the architectural extractor fine-tuned the abstraction by types to reduce the amount of object merging, and have the diagram display those objects as separate boxes [3].

Other usability issues. The participant highlighted some usability issues in the tool that lowered the usefulness of the diagram. In some cases, the participant had to print hard copies of the extracted diagram to be able to use it concurrently with the interactive viewer. Also, the scrolling, zooming, etc. were not as helpful to the participant who avoided using them by relying on other features. Instead of highlighting an edge, for example, the participant used the features to search for objects in the ownership tree, or to search a diagram for a given label. For scrolling, the participant preferred to use the overview screen (Fig. 3).

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 diagrams of the run-time structure, one might argue that we should have provided developers with more helpful class diagrams, that were refined just as the diagrams of the run-time structure 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 [14]. Instead, we provided the participant 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, diagrams of the run-time structure do show such relations, and the object labels often reference the types that appear on the class diagram. In other words, the diagrams of the runtime structure 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 at the same 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 diagrams of the runtime structure.

In the real world, code modification tasks on real software systems are performed under strict deadlines. For this case 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
diagram, 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.

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 diagrams of run-time structure is complementary to the information from other sources.

In terms of the task, we used 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. In terms of the subject system itself, DrawLets may not be representative of other, more complex systems. On one hand, 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. In particular, 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. Please refer to the report discussing the annotation process for details.

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. 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 run-time structure, we chose a participant who did not require extensive training to avoid much of the learning curve involved in adopting a new tool and approach, as is often desired for high-quality case studies.

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 add annotations to 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 a diagram that did not reflect the additional code. Also, the study identified that there was initially a problem with the level of abstraction which required refining the annotations to make the diagram reflect the developer’s mental model. The participant required certain objects to be moved up or down in the hierarchy, but she was not involved in the extraction process, and was not allowed to move objects using the tool. Perhaps, the ability to interactively refine a diagram during a code modification task might make the diagram more useful. This is a capability for which we are currently developing tool support.

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. Our results were promising, so we conducted this study to investigate whether the run-time structure can help answer some of these questions. We are also planning to conduct more studies once we fix the issues identified both in the tool and the approach.

6. RELATED WORK

Our previous evaluation of the run-time structure. We conducted three empirical studies on the run-time structure. In one study, we asked the developer to sketch the intended architecture of the system, then we compared this desired diagram to the one we extracted from the code by adding annotations. We iterated this process to refine the architecture to reflect the design intent, however, we did not ask the developer to use this extracted architecture to do any code modification. 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 make the diagram as useful as possible to the developer doing the code modification.

Previous studies on the run-time structure. Walker et al. developed an approach for visualizing the operation of an object-oriented system at the architectural level. Their approach builds on the Reflexion Model 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. 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 case studies on DrawLets based on the code structure. Rajlich and Gosavi found that incremental change in software is sometimes unanticipated and impacts many classes. They proposed a technique for unanticipated change that used programming concepts, and contained steps of concept location, actualization, incorporation, and change propagation which required knowledge of class dependencies. They applied this technique to DrawLets using class diagrams to perform the change request. They observed that the design of DrawLets did not help localize the change that they made and thus used techniques such as refactoring to limit the length of change propagation. In our study, we used a different approach where we fine-tuned the run-time structure to make it more helpful, and we found
that a developer who uses the run-time structure may modify fewer classes. However, we do not claim that using diagrams of the run-time structure for code modification limits the length of change propagation since modifiability seems to be a quality attribute of the software itself.

**Previous studies on diagramming tools.** Several studies have been conducted about the contribution of diagramming tools in program comprehension [7, 12, 19]. Hadar et al. [15] conducted a study on developers comprehension of UML diagrams. Their study focused on how developers use several 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 [15, 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 of a system, to help answer their questions about objects and relations between objects. We believe these results are promising even though the study involved only one developer. This study in addition to previous studies revealed some usability challenges in the current tool, which may lower its usefulness to developers even though previous work [15, 7] used partial runtime views such as sequence diagrams.

### 8. REFERENCES


