Usefulness of the Run-time Structure during Coding Tasks

Marwan Abi-Antoun **Nariman Ammar**

September 2010

Department of Computer Science
Wayne State University
Detroit, MI 48202

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.

Keywords: runtime structure, case study, software evolution

Contents

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 [15], 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 [24].

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 case study by Rajlich and Gosavi [19]. 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.

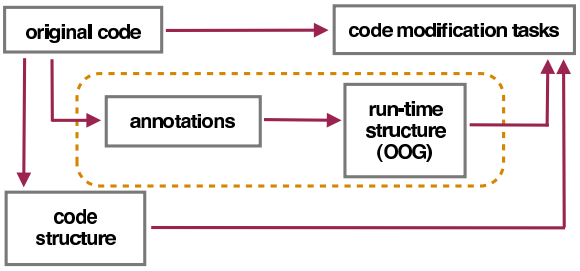

Figure 1: Approach for code modification based on both the run-time structure and the code structure.

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, e.g., `MODEL` and `UI` in figure 2.

Previous work studied developers while they perform code modifications based on the code structure, i.e., class diagrams or Eclipse call graphs [19, 21, 16]. 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 the code structure (Fig. 1). In order to extract the diagram statically, we previously proposed an ownership type system which enabled architectural extractors to 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, Ownership Object Graph (OOG). Figure 2 shows the extracted OOG of the DrawLets system [2]. The OOG is a hierarchical object graph that is composed of nodes that represent objects and edges that correspond to relations between these objects. Edges between objects correspond to field declarations in the code.

The hierarchical representation and the associated ability to expand or collapse elements has been shown to be effective for software architecture [22]. The OOG that we provide developers with provides them with the necessary architectural abstraction which abstracts objects by ownership hierarchy by nesting objects underneath other objects. Nested objects often reside in ownership domains which could be either public or private. An ownership domain is represented by a white box with dashed border, e.g. `owned` in figure 2. The other form of abstraction is by types, where the architectural extractor can use the notion of subtyping to specify the architecturally relevant types. We believe that these two forms of abstraction that the OOG provides make it scale to large code bases and thus more useful to developers.

The process that we follow often requires the architectural extractor to refine the annotations until the extracted diagram reflects the architectural intent. However, the ownership type system keeps the architectural extractor honest, and he cannot tweak the diagram in such a way as to make the diagram more useful than it really is. The only forms of abstraction that he can perform to refine the diagram are: abstraction by ownership hierarchy; and abstraction by types. 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 DrawLets 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 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 DrawLets system, then to perform a code modification task designed by Rajlich and Gosavi in a previous case study [19].

The Subject System. DrawLets 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 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. 2). 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

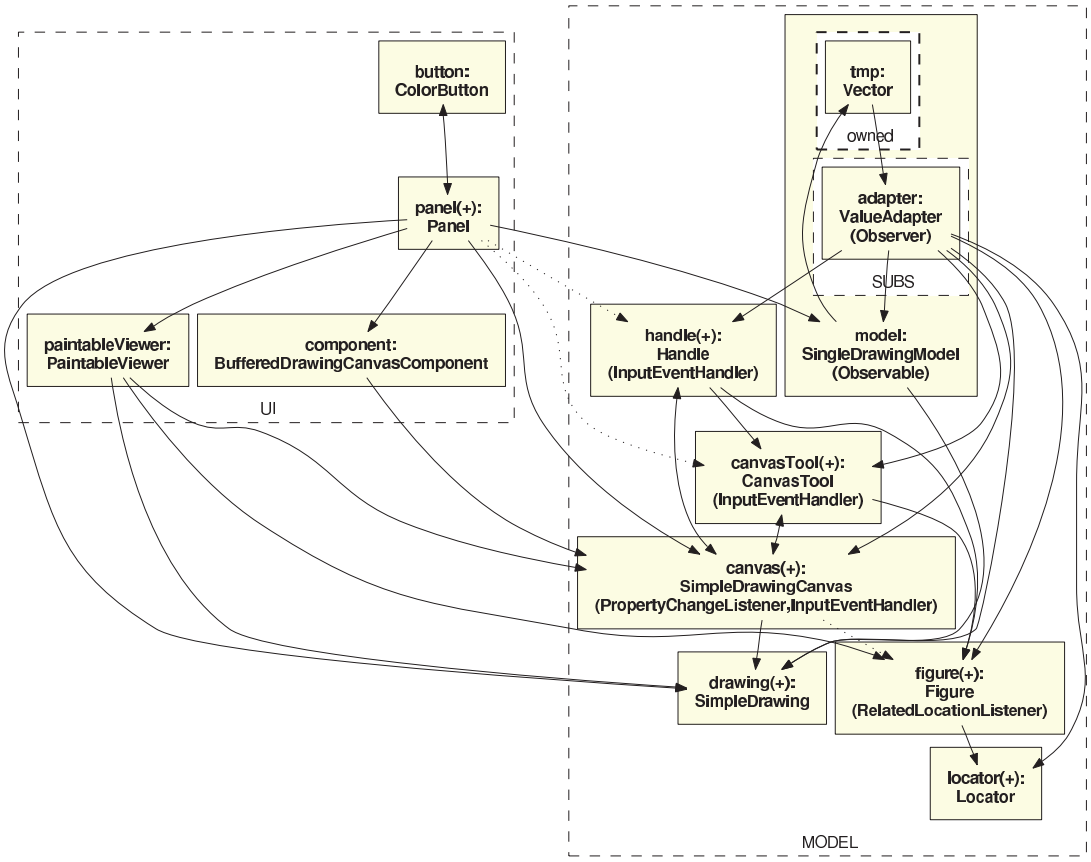

Figure 2: Ownership Object Graph (OOG) of DrawLets.

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 [14] since she helped analyze data from a previous study on the run-time structure of JHotDraw [5]. JHotDraw and DrawLets are closely related since they both descend from the HotDraw Smalltalk framework for implementing drawing applications [19].

The participant was not involved with the process of adding the annotations to DrawLets 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 DrawLets 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 DrawLets 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 DrawLets, while she was concurrently editing the code in Eclipse. For example, she was able to trace from an element on the diagram 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. 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 DrawLets [10]. Finally, the participant had access to two manually generated UML class diagrams from the previous case study [19]. 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 [3].

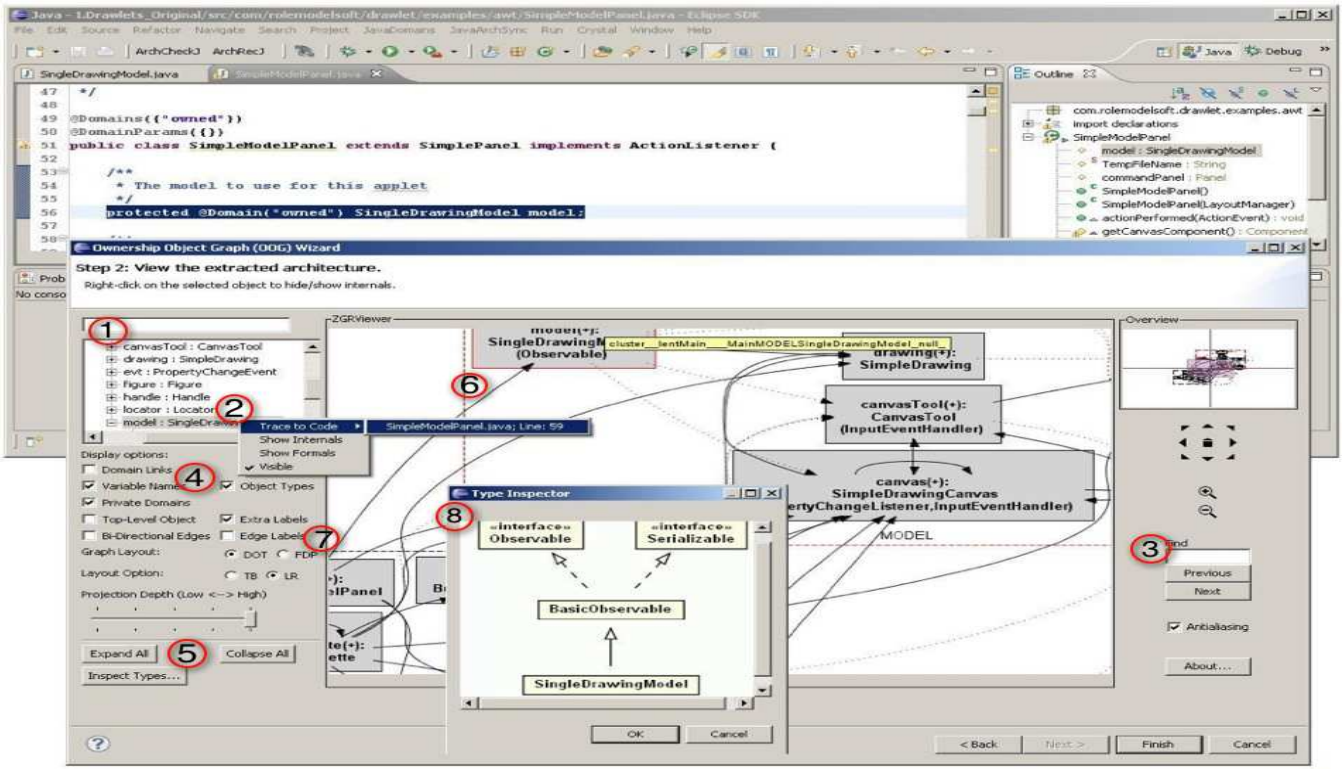

Figure 3: The OOG viewer tool used during the study.

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" [19].

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 time log of 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 recording mechanism because of the long running study. The study material are available online [10].

Analysis. We analyzed the participant's thought process from the transcripts using a qualitative protocol analysis [23]. 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, which is an interactive version of the diagram, 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. We assume that the modification done in the previous study [19] was also incomplete for the same reason. The developer in the previous study added a class `OwnerIdentity` which holds basic owner data, such as ID and a password. It also displays a dialog box, allows users to change current session owner. It also has a listener to listen when the user clicks on a button in this box. He also added a listener class `SimpleListener`. He modified two classes to incorporate the newly created classes into the old code: `AbstractFigure` and `SimpleDrawingCanvas`. In class `AbstractFigure`, he added four methods: `setFigureOwner()`, `getFigureOwnerID()`, `getFigureOwnerName()`, `validateOwner()`. He also modified the following methods to make them more secure by passing the session owner id: `move()`, `translate()`, `requestConnection()`, `setBounds()`, `setSize()`, and `setStyle()`. In class `SimpleDrawingCanvas`, all methods that called the above listed methods of the Figure interface were modified to handle the extra parameter. Method `cutSelections()` was modified to prevent figures not belonging to the current session owner from being cut. He also added `getNewID()`, `getCurrentSessionOwnerName()`, and `getCurrentSessionOwnerID()` methods to allow access to the current session owner information stored in `SimpleDrawingCanvas`.

Still, we believe the participant's modification was better in two respects: she covered more cases than in the previous case study, and she did not need to add any listener classes or interfaces, `SimpleListener`, like the developer in the previous study. We also believe these code modifications were sufficient for the purpose

Table 1: The OOG viewer tool features used by the participant during the study. The first column refers to the numbered parts in Fig. 3.

Ref. no.	OOG viewer feature	Frequency of usage
1	Search Ownership Hierarchy	8
2	Trace To Code	14
3	Find Object Label	2
4	Object Types	6
5	Show/Hide Internals	11
6	Highlight Edges	11
7	Edge Labels	1
8	Inspect Types	1

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 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 can 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.

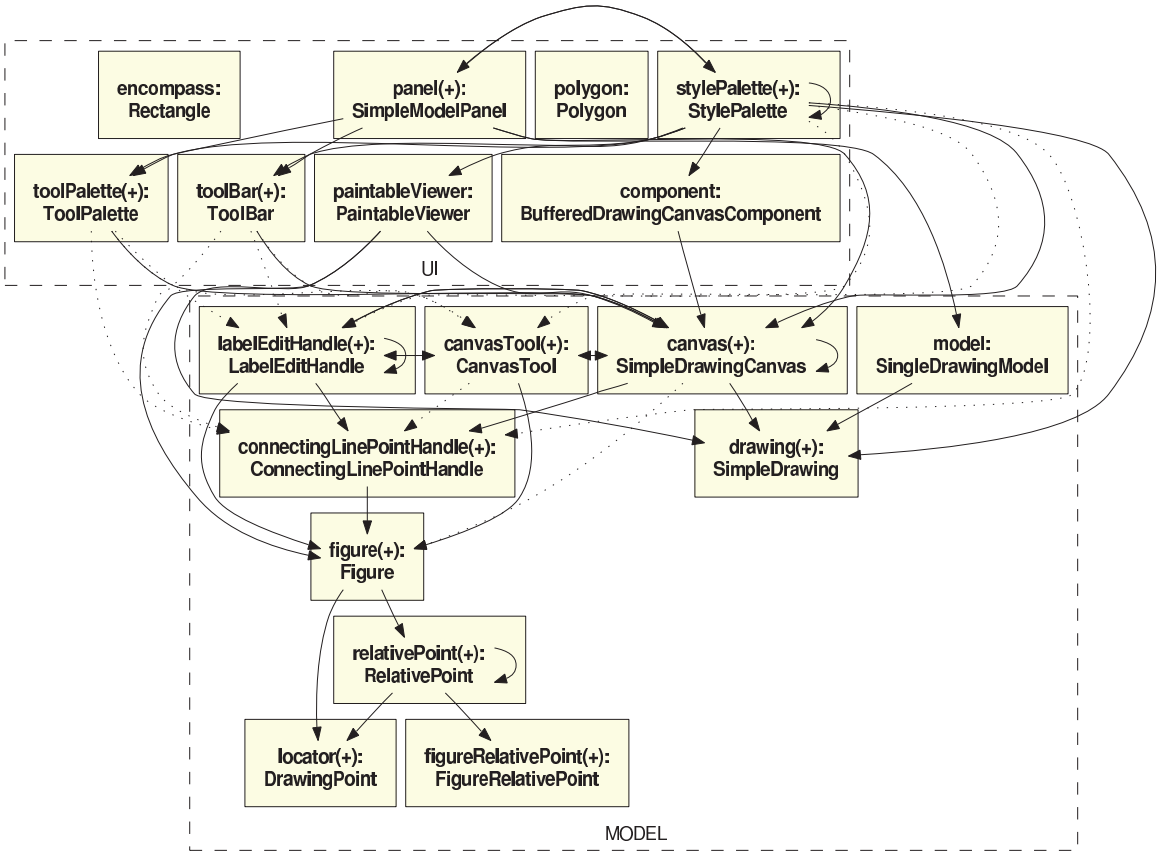

Figure 4: Initial version of the extracted OOG.

The interactive version of 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 `panel:SimpleModelPanel` and `model:SingleDrawingModel` which linked her to the corresponding field declaration in the Eclipse text editor (Fig. 3). The participant could not only navigate to code from the graphical representation of the diagram, she was also able to use a tree representation 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(Fig. 7).

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.

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

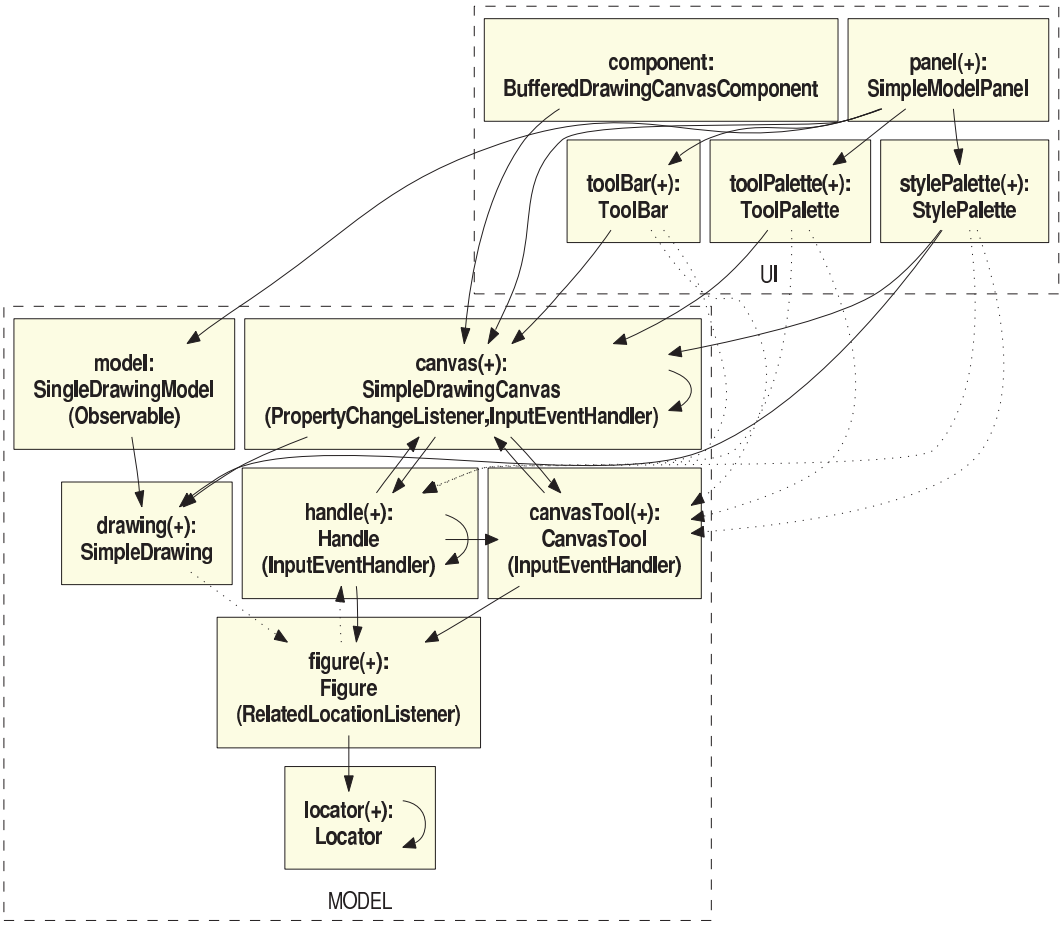

Figure 5: Second version of the extracted OOG.

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 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:

```
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

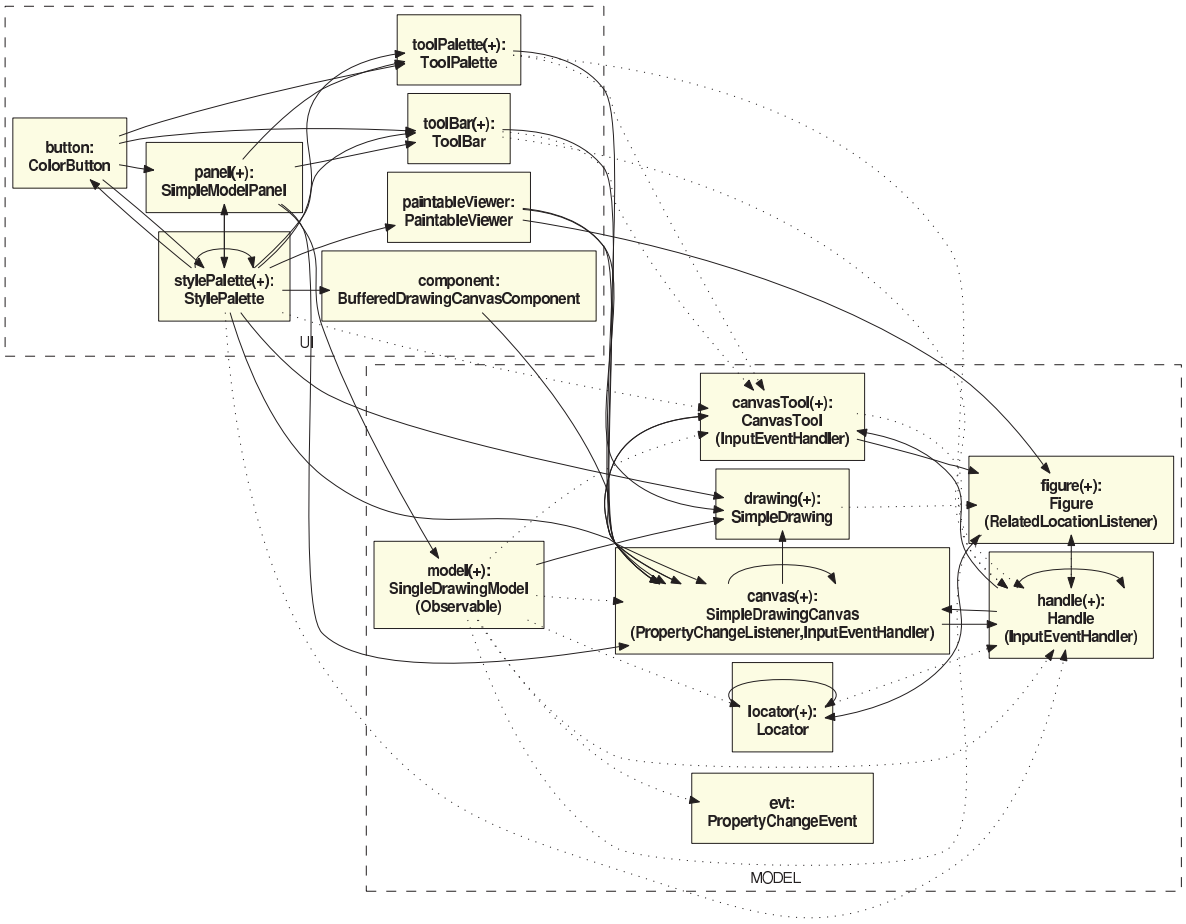

Figure 6: Third version of the extracted OOG.

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.

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 (`SingleDrawingModel`) 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 `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 nav-

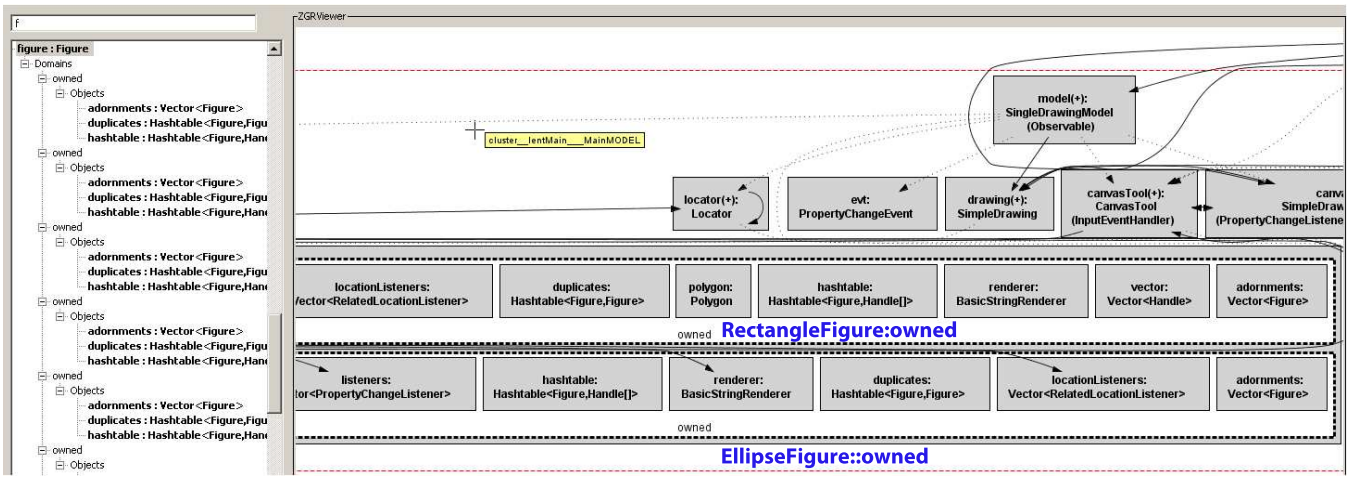

Figure 7: Search ownership tree feature shows all possible relations with `figure:Figure` object.

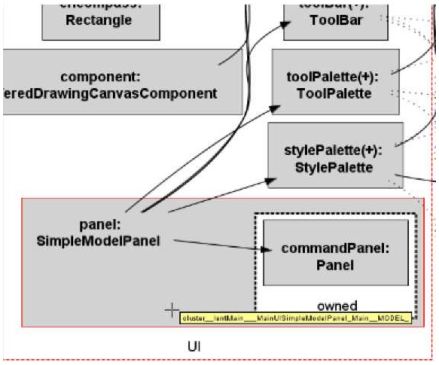

Figure 8: Substructure of `panel:SimpleModelPanel`.

igated 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.

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.

Domain names may need to be fully qualified when merging objects. At one point during the study, the participant thought that the diagram was showing redundant information. For example, when she expanded the substructure of the `figure:Figure` object, she noticed that it has several nested domains, obviously because DrawLets supports different types of figures. In the future, we will enhance the tool by showing the domain names qualified by their declaring classes, e.g., `EllipseFigure::owned` or `RectangleFigure::owned`, instead of `owned`. This could make the diagram more explicitly state that is it merging related objects and their domains.

Virtual fields. The participant noticed that `model:SingleDrawingModel` has a list of observers `VIRT_observerList`. However, when she highlighted the edge between these two objects and used the trace to code feature, she did not get any interesting code. This is a current limitation in the tool where the

VIRT_XXX notation corresponds to a "virtual field" annotation in the code that is manually added since the static analysis does not understand some implementation details in the code.

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.

For example, she wanted the `BufferedDrawingCanvasComponent` object to be labeled `DrawingCanvasComponent` instead, especially since the feature of trace to code took her to this line in the code:

```
DrawingCanvasComponent component =
    new BufferedDrawingCanvasComponent(canvas);
```

The participant had this observation because DrawLets supports different `DrawingCanvasComponent` instances depending on the application, and `BufferedDrawingCanvasComponent` is intended only for AWT applications. We could perhaps make the object labels more intuitive by allowing the developer using the OOG viewer to interactively rename selected objects.

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`. 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

Empirical studies are often exposed to threats of validity, and our study is no exception. However, the qualitative nature of our analysis makes these threats a bit more manageable. In Section 5.1, we discuss how we tried to limit those threats. Plans to remedy these threats in future studies are discussed in Section 5.3.

5.1 Threats to validity

External validity. 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 [12]. 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,

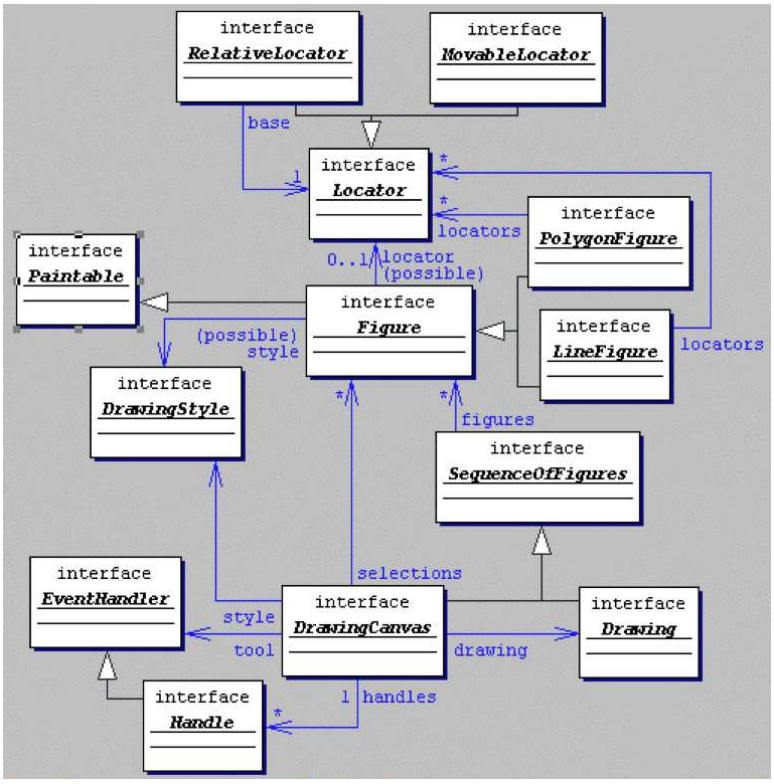

Figure 9: Class diagram of the core interfaces.

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 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.

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 [25, p. 43]. We are also planning to replicate our findings in another case study.

Internal validity. 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.

Construct validity. 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

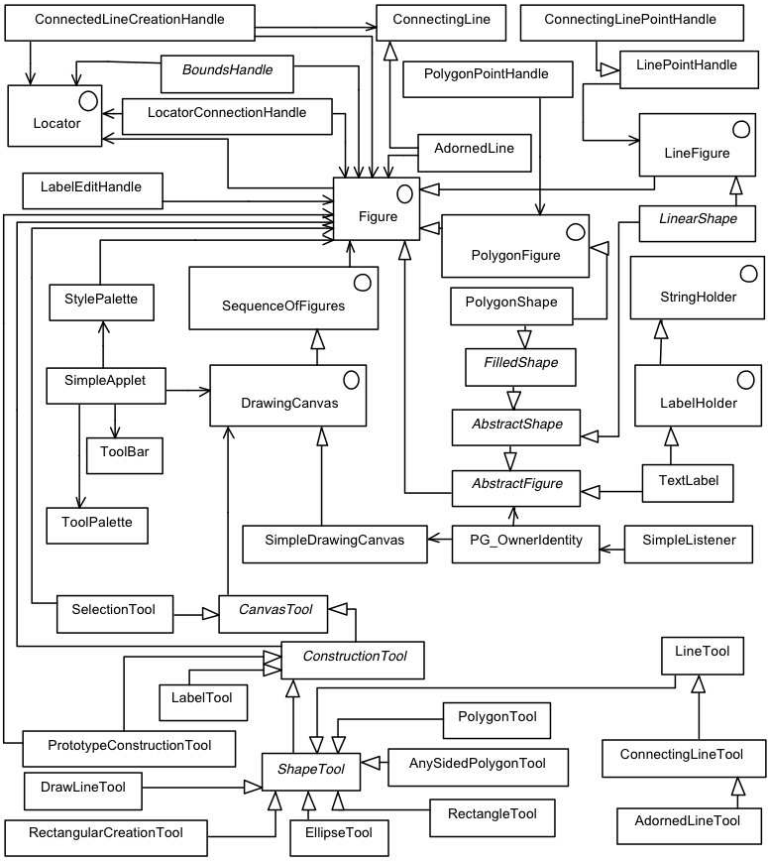

Figure 10: Class diagram of the top-level classes.

coding activities [5]. 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 follow the replication approach by conducting more case studies. 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 [25, p. 68].

5.2 Other limitations

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 [3] for details.

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.

5.3 Summary and Future Work

Usability and usefulness of software visualizations go hand in hand. In this study we found that for the run-time structure to be more useful, it should also be usable. For a software visualization tool to be considered useful, it should provide graphical representations of software structures linked to textual views of the program source code [22]. We provide developers with a tool that helps them visualize the object relations on the diagram and navigate to the code to see the corresponding field declarations. We also use an approach for code modification that is responsive to the developers changing mental model of the system. These features in the diagram and in the approach achieve program comprehension which is necessary for code modification.

The study identified some usability issues in the tool that may have lowered the usefulness of the diagram. The issues can be summarized into two broader categories: physical layout of the graph and the visual syntax and semantics which can play an important role in program comprehension [18]. Future work includes enhancements on both the diagram and on the approach.

For future studies, we are working on enhancing the visualization of the diagram to be more intuitive and user-friendly. We are also planning on incorporating the modifications that the developers make in the code being annotated. This way we can get the developers more involved in the refinement of the OOG by letting them see the new object relations on the refined diagram. To make the annotation process iterative and continuous, we need to make it fully automated instead of the current semi-automated approach.

6 Related Work

Our previous evaluation of the run-time structure. We conducted three empirical studies on the run-time structure [1, 5, 6]. In one study [1], 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 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. [24] 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.[20] 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 [19] 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 provided the developer with the run-time structure in addition to the code structure, 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, 11, 17]. Hadar et al. [13] 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 [13, 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 the usefulness of the diagram to developers. Once we address some of the issues both in the approach and in the tool, we are planning to seek stronger empirical evidence by conducting more case studies and, if possible, field studies of outside developers working on real systems under timed conditions.

References

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

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

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

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

[5] M. Abi-Antoun, N. Ammar, and T. LaToza. Questions about Object Structure during Coding Activities. In *CHASE*, 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] R. Brooks. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 1983.

[9] DrawLets. `www.rolemodelsoft.com/drawlets/`, 2002. Version 2.0.

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

[11] 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.

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

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

[14] JHotDraw. `www.jhotdraw.org`, 1996. Version 5.3.

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

[16] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program Comprehension as Fact Finding. In *ESEC/FSE*, 2007.

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

[18] D. L. Moody. The Physics of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *TSE*, 2009.

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

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

[21] M. P. Robillard, W. Coelho, and G. Murphy. How Effective Developers Investigate Source Code: An Exploratory Study. *TSE*, 30(12), 2004.

[22] 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.

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

[24] 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.

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