1-1-2010

A Front-End For An Ownership Object Graph Interactive Editor

Talia Frances Selitsky
Wayne State University

A FRONT-END FOR AN OWNERSHIP OBJECT GRAPH INTERACTIVE EDITOR

by

TALIA SELITSKY

THESIS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

2010

MAJOR: COMPUTER SCIENCE

Approved By:

Advisor	Date

DEDICATION

To my family

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Abi-Antoun, for all of his hard work in helping me to realize this thesis. He assigned me a project that I loved, which made it easy to be motivated. I learned from him so many things that I apply every day, and that I feel will help me tremendously throughout my career. He is a great addition to the Wayne State University Computer Science department.

I was very fortunate to have a wonderful and distinguished thesis committee. I would like to thank Professor Rajlich who got me interested in software engineering as a field, and involved me early in my master's degree in software engineering research. I would also like to thank him for helping me get my first computer science job. Thank you also Professor Reynolds for being part of my committee and giving me so much helpful advice.

Thanks also to the members of the SEVERE group for being so supportive and committed to research. I would especially like to thank Laurentiu Vanciu for being such a great mentor, and always being available when I needed help or advice of any type. Nariman Ammar got me through the program with all of her help, kindness, and humor. Sonia Haiduc and Grace Metri made coming to the lab something that I looked forward to, and something that I will miss.

I was fortunate to be a member of the ACM-Women chapter at Wayne State University, which is a great group full of fun and inspiring members who I thank for their enthusiasm, and teaching me how to play a role in the computer science community. I would especially like to thank Monika Witoslawski, who started the group, for all of her dedication and vision.

Thanks to my family, especially to my wonderful parents and sisters for putting up with me during this time. I would also like to thank my grandmother for all of her support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1: Introduction

One of the greatest challenges faced by practicing software engineers is understanding the structure of large software systems. This understanding is necessary for many software evolution tasks, such as isolating and fixing defects, adding new functionality, optimizing performance, or identifying and addressing security vulnerabilities.

Over the last 20 years, the discipline of software architecture has emerged to capture the high-level structure of a software system [39, 51]. A software architecture abstracts the system's organization into components and shows how those components interact.

The different ways of looking at a system's structure are called architectural views [11]. Different views serve different purposes. A code view shows modules as groups of source code functions, files, classes, and packages. Code views are useful for reasoning about dependencies between source code modules. Runtime views, the focus of this work, show components and connections between them. A component represents a set of one or more objects in the running system. Runtime views are useful for tasks related to performance, reliability, and security [8]. Such views are increasingly important in object-oriented code, which makes heavy use of design patterns [19].

Instead of reading the code to understand it, it would be ideal to have a tool that could assist the engineer in extracting a runtime view. Recent work has shown that sound runtime structure extraction from object-oriented systems is technically feasible [4]. Soundness means two things. First, each runtime object has exactly one representative in the object graph. Second, the object graph has edges that correspond to all possible runtime points-to relations between those objects.

The extraction tool produces a default decomposition [4]. But many decompositions are possible. So, ideally, one must provide efficient and interactive mechanisms

to let a developer refine the default decomposition to reflect their design intent. We consider that the object graph reflects design intent when it is at an abstraction level that is comparable to a conceptual architecture, i.e., they both have similar tiers (a tier is a way of grouping components with the same functionality to create conceptual partitions), similar hierarchical decomposition (a component can have sub-structure), and similar number of components and tiers at each hierarchy level.

The goal of this work is to support the above refinement. We propose a tool with a graphical user interface to manipulate an extracted object graph. We focused on the front-end of the tool and on addressing some of the usability limitations of the previous work on extracting object graphs.

1.1 Architectural Abstraction

Because a system's runtime structure often has many objects, the resulting graph is often large and complex. Hierarchy is often used to mitigate this problem. This is because hierarchy provides architectural abstraction, allowing two or more nodes to collapse into one, and allows collapsing or expanding selected elements [54] to allow both high-level and detailed understanding [54]. But architectural hierarchy is not directly expressible in a general purpose programming language.

In order to impose hierarchy on a flat object graph, we use *ownership domains*. An ownership domain is a runtime abstraction that groups together objects. An ownership domain has a name which indicates design intent, and policies that govern how it can reference objects in other domains. An ownership domain is either private or public. A private domain provides strict encapsulation. A public domain provides logical containment and its objects are accessible to all objects that can access the outer object. Each object can support one or more domains to hold its internal objects. In particular, public domains enable a developer to impose a conceptual hierarchy on objects. Thus, ownership domains support the conversion of a flat object

graph into a hierarchical object graph, which we refer to as an *ownership object graph* (OOG) [4], by allowing objects to contain other objects.

An OOG provides *abstraction by ownership hierarchy* when it shows architecturally significant objects near the top of the hierarchy and data structures further down. Moreover, an OOG can provide *abstraction by types* and allow objects to be collapsed further according to their declared types.

1.2 Object Diagrams

It is important for developers to understand the type structure of a program when they are developing a software system. The type structure is often represented as class diagrams. There are many tools that can extract class diagrams [28]. The runtime structure is depicted as ownership object graphs, which are a type of object diagram. An object diagram is important because it helps the developer understand the instance structure of the program which is important for object-oriented code [19]. There are not many tools that can effectively extract object diagrams from the code. The tools that do extract object graphs, extract flat object graphs [25, 57], which do not scale to programs of more than a few classes since the graph becomes quickly overly complex for people to use effectively. For this reason, it is important for object diagrams to include architectural abstraction.

1.3 Background on Scholia

In this section, we summarize previous work, Scholia [4, 3], on which this approach builds, and which is the state of the art in the sound, static extraction of runtime architectures.

Scholia reasons about runtime architecture through the use of ownership type annotations, which must be added to the code before analysis begins. In order for

the extracted architecture to be sound, the annotations must be consistent with each other and with the code, so the developer must run an ownership type checking algorithm on the annotated code, and fix any high-priority warnings.

SCHOLIA adopts the accepted *extract-abstract-present* strategy [31] for architectural extraction. A static analysis extracts a sound object graph from the annotated code, using ownership to generate a containment hierarchy of objects. Next, SCHOLIA *abstracts* the extracted object graph into a runtime architecture showing components and connectors. Then, SCHOLIA *presents* the built runtime architecture in an architecture description language (ADL). But this thesis is concerned with the manipulation of extracted object graphs, and does not abstract object graphs into *component-and-connector* views represented in an ADL, as in SCHOLIA [4].

1.4 Limitations of Scholia

An important issue in SCHOLIA is that the process of refining the extracted architecture is somewhat awkward. When the extracted architecture does not match the conceptual model, the architect must identify the cases where the cause of the discrepancy is an incorrect ownership relationship, change the ownership annotations in the code consistently to reflect the corrected ownership relationship, then regenerate the architecture. This issue makes using SCHOLIA tedious and time-consuming.

The proposed tool addresses this issues by allowing the developer to directly and interactively manipulate an extracted object graph. This way, developers can interactively refine the extracted view to bring its abstraction closer to their design intent—without, of course, making the diagram unsound in the process.

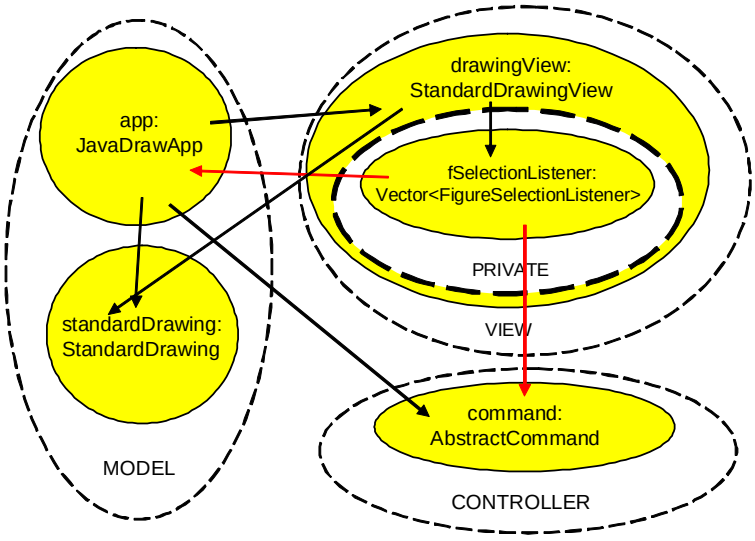

Figure 1.1: MicroDraw represented as an OOG.

1.5 OOG Structure by Example

We illustrate by example the SCHOLIA approach on MicroDraw (Fig. 1.1), which illustrates the design of JHotDraw [1], an open source framework full of design patterns. MicroDraw is a good example of the Model-View-Controller architecture [19]. The MicroDraw architects would like to indicate that MicroDraw follows the Model-View-Controller design pattern [19]. So, they represent the runtime architecture as having three top-level ownership domains, `MODEL`, `VIEW` and `CONTROLLER`, which contain instances of the core types as follows:

- `MODEL`: the `MODEL` domain has instances of types `Drawing` and `Figure` (a `Drawing` consists of `Figures`). In the diagram, the `standardDrawing` object is labeled by the type `StandardDrawing` (the class `StandardDrawing` implements the `Drawing` interface).

- `VIEW`: the `VIEW` domain has instances of type `DrawingEditor` and the `DrawingView`. In the diagram, the `drawingView` object is labeled by the type `StandardDrawingView` (the `StandardDrawingView` class implements the `DrawingView` interface). Also, the `app` object is labeled by the type

`JavaDrawApp` (the class `JavaDrawApp` implements the `DrawingEditor` interface).

- `CONTROLLER`: the `CONTROLLER` domain has instances of type `Command`. In the diagram, the `command` object is labeled by the type `AbstractCommand` (the class `AbstractCommand` implements the `Command` interface).

The interface `DrawingEditor` extends from the `FigureSelectionListener` interface. The class `AbstractCommand` implements the `Command` interface, as well as the interface `FigureSelectionListener`. Thus, a reference of type `FigureSelectionListener` could point to either `DrawingEditor` objects in the `VIEW` domain, or to `Command` objects in the `CONTROLLER` domain (the two thicker edges in Fig. 1.1). This subtyping illustrates one of the features of object-oriented languages that makes object-oriented code challenging to analyze.

1.6 Graphical Conventions

In this document, our visualization (Fig. 1.1) uses circle nesting to indicate containment of objects inside domains, and domains inside objects. Dashed-border white-filled circles represent domains. Solid-filled circles represent objects. Solid edges represent field references. An object labeled `obj:T` indicates an object reference `obj` of type `T`, which we then refer to either as "object `obj`" or as "`T` object", meaning "an instance of the `T` class".

1.7 Contributions

I break up the overall contribution into the following:

- **Front-end for the iterative refinement of an extracted object graph:** We designed the user interface for an interactive editor that allows developers to

refine an initial extracted flat object graph to better match their mental model of the runtime structure.

- **Novel techniques for refining a flat object graph into an ownership object graph with collapsing and expanding sub-structures:** OOGIE allows developers to interactively manipulate flat object graphs into ownership object graphs. The developer can create nested sub-structures, and developers can manipulate the hierarchy by adding domains which can be used to nest objects. OOGIE's contribution is that it supports these activities with easy-to-use features such as drag-and-drop and context menus. These activities are represented graphically so that developers can visualize how they are manipulating the runtime structure.

- **Evaluation of OOGIE:** We evaluated OOGIE to test whether outside developers can use OOGIE to refine an initial object graph and make it match their mental model of the runtime structure.

1.8 Thesis Statement

The thesis is:

> *Developers can use the OOGIE tool to iteratively and interactively refine an object graph to make it reflect their mental model or design intent through direct manipulation.*

I created several corresponding hypotheses, subordinate to the main thesis. Since each hypothesis is smaller than the main thesis, each can be directly supported by evidence. Taken together, these hypotheses solve the problem of interactive refinement of an object graph.

> *H1: A developer can use OOGIE to manipulate the ownership hierarchy.*

Success Criteria. The success criteria to objectively measure or falsify this hypothesis include:

- Developers can use abstraction by ownership hierarchy, such as adding or removing domains, and moving objects between domains to refine the hierarchy of an ownership object graph to better fit their mental model of a system's runtime structure.

Evidence. We support this hypothesis with the following evidence:

- We evaluate the tool on several real object-oriented systems.

H2: A developer can use OOGIE to control the level of detail of a runtime structure that is displayed in an OOG.

Success Criteria. The success criteria to objectively measure or falsify this hypothesis include:

- A developer can expand or collapse the sub-structures of selected objects in order to control the level of detail of a runtime structure that is displayed in an OOG.
- A developer can use abstraction by types to collapse objects further according to their declared types, or to reduce the amount of object merging.

Evidence. We support this hypothesis with the following evidence:

- We evaluate the tool on several real object-oriented systems.

H3: Developers can accomplish H1 and H2 through direct manipulation

Success Criteria. The success criteria to objectively measure or falsify this hypothesis include:

- Developers can accomplish H1 and H2 using user-friendly manipulation features such as selecting items from context menus, and using drag-and-drop.

Evidence. We support this hypothesis with the following evidence:

- We evaluate the tool using a self-evaluation methodology called cognitive dimensions.
- We let experts in Human-Computer Interaction review the tool's user interface design.

- We evaluate the tool's usability using outside developers.

  H4: OOGIE is user-friendly to developers

Success Criteria. The success criteria to objectively measure or falsify this hypothesis include:

- OOGIE has easy-to-use navigation.
- Developers can understand the graphical representation.
- Developers can refine the object graph to better match their mental model within a reasonable amount of time and steps.

Evidence. We support this hypothesis with the following evidence:

- We evaluate the tool using a self-evaluation methodology called cognitive dimensions.
- We evaluate the tool using real object-oriented systems and outside participants. During the exit interview, we ask the participants about the perceived user-friendliness of the tool.

1.9 Outline

The rest of this report is organized as follows: Chapter 2 discusses previous work. Chapter 3 discusses requirements. Chapter 4 discusses the tool implementation. Chapter 5 discusses evaluations that were conducted on the tool. Chapter 6 discusses how the requirements were satisfied, the limitations of this work, and concludes.

Chapter 2: Previous Work

2.1 Runtime Structure

Many dynamic analyses focus on visualizing the object structures of a running system [12, 14, 32, 49, 29, 26, 58, 45, 15, 20, 53, 52, 37, 13, 46, 38, 44].

These dynamic analyses handle programs for which source code is not available, do not require source code annotations, and allow more fine-grained user interaction in producing a visualization.

These task-focused views explain detailed interactions, help developers understand a program, or find low-level defects, such as memory leaks [15, 42]. The extracted views have the granularity of individual objects and classes.

Many of these approaches extract one or more collaboration diagrams [22, 29, 14, 45, 58], rather than a global object diagram for the entire system. A collaboration diagram that contains all objects and all invocations between them may be unusable, for anything but the smallest of systems. Most approaches allow the developer using the tool to focus the interaction diagram to include only specific method invocations, issued from a starting method of interest. In some cases, the recovered views highlight design patterns [30, 48], but often, they are not architectural, because they are neither abstract nor global.

2.2 Ownership Structure

More closely related are dynamic analyses that infer the ownership structure of a running program based on its heap structure [23, 36, 43, 18, 33]. In general, dynamic analyses have the advantages of being more scalable and more precise than their static counterparts. In addition, dynamic ownership analyses do not require a programmer to annotate their code with ownership type annotations. However, previous such

analyses assume a strict owner-as-dominator model which cannot represent many design idioms. In such a model, a higher-level object cannot collapse underneath it not many low-level objects, so they end up cluttering the top-level diagram.

Hill, Noble and Potter [23, 36] and Potanin et al. [40] used dynamic analyses and showed both matrix and graph views of ownership structures and demonstrated that ownership is effective at organizing runtime objects. Several others followed suit [33, 43, 18]. Other work [43] characterizes sharing and ownership and produces a matrix display of the ownership structure. They later used the results of this analysis to investigate memory leaks [42]. Similarly, another work [33] uses lightweight ownership inference to examine a single heap snapshot rather than the entire program execution, and scales the approach to large programs through extensive graph transformation and summarization.

This body of work showed that ownership does provide abstraction, and is effective at organizing large object graphs. SCHOLIA uses the same key insight but in a static analysis which must address several additional challenges. Most of the previous tools extracted abstractions that are hard-coded in the tool. OOGIE allows a developer to refine the abstraction, thanks to the expressiveness of ownership domains.

2.3 Hierarchical Views

Several tools produce hierarchical views of the code architecture that can expose or collapse sub-architecture such as the RIGI visualization system [35] and its follow-up SHRIMP VIEWS [56]. These tools also allow for code exploration by allowing developers to examine different parts of the code. RIGI also allows developers to refine the abstraction. But these systems show the code structure, and OOGIE shows the runtime structure.

2.4 Iterative Refinement

DA-TU [24] is an application that manages the complexity of large graphs in two ways, clustering and navigation. The clustering action allows developers to manipulate the level of abstraction of a graphical representation of a software base. This is done through the grouping of nodes together to be represented by a super-node. The navigation action allows the developers to only look at a sub-set of the graphical representation at a time. We incorporated clustering and navigation functionalities into OOGIE. But DA-TU groups nodes together based on their location in the graphical representation. OOGIE is designed to allow developers to group nodes together based on their view of the runtime structure, not on the location in the graph.

Another class of tools that often include iterative design features are UML tools, such as QuickUML [10], which allows developers to design UML class diagrams. But OOGIE follows strict guidelines as to how the developer can modify the visualization. The requirements come from experimental results [8], and they must be operations that preserve diagram soundness (Sec. 1).

Chapter 3: Requirements[1]

In this chapter, I discuss the requirements for the tool, which come from the discussion in the introduction (Chapter 1), our previous research [8] and from the results of the evaluations we conducted (Chapter 5).

3.1 General Requirements

The following is the list of the general requirements:

3.1.1 RQ 01-Support iterative refinement

The extracted object graph may not necessarily be the same as what the developer would have drawn [8]. Our tool must allow developers to iteratively refine the graph to make it match their intent, while still preserving soundness.

3.1.2 RQ 02-Must be an Eclipse plug-in

The tool must be well integrated with the Eclipse IDE.

3.1.3 RQ 03-Must load and save states

A developer must be able to save object graph configurations.

3.1.4 RQ 04-Must be easy to use

The tool must be easy to use because the developer needs to focus on manipulating the object graph, and not on trying to operate the tool.

[1] Portions of this chapter appeared in [7, 50].

3.1.5 RQ 05-Must invalidate incorrect manipulations of the OOG

The tool must prevent developers from manipulating the OOG incorrectly. This includes deleting non-empty domains, giving domains names that are already used by other domains, moving objects into a private domain that are not strictly encapsulated by the parent object, and adding domains to other domains.

3.1.6 RQ 06-Support the data model of ownership domains

OOGIE must support the data model of ownership domains which we explain in more detail later (Sec. 4.5).

3.1.7 RQ 07-Maintain diagram soundness

OOGIE must preserve diagram soundness (Sec. 1). In terms of tool implementation, this means that no relationships can be added or deleted directly, and no objects can be added or deleted directly. This is because the tool allows developers to edit an abstraction of the runtime structure, and not to modify the runtime structure directly.

3.2 Human-Computer Interaction Requirements

In order for the tool to be user-friendly and to take into account good practices from human-computer interaction research, we imposed the following requirements.

3.2.1 RQ HCI1-Create visual distance between semantic ideas

It is important that semantic concepts should be differentiated from each other with more than one variable. This helps the brain to quickly process a visual representation without much cognitive overhead so that developers can focus on more difficult tasks [34]. This includes using different shapes and different colors to distinguish between semantic concepts.

3.2.2 RQ HCI2-Make use of hierarchies

Previous work showed that organizing complex systems into hierarchies is very effective, and also helps in top-down understanding of software engineering diagrams [34].

3.2.3 RQ HCI3-Limit number of on-screen components

The number of components that appear on the screen at a time should not exceed what the working memory can handle. Crossing this boundary leads to cognitive overloading, and a decrease in comprehension [34].

3.2.4 RQ HCI4-Implement easy navigation

The tool should allow the developer to go from one place to another in a simple manner [55]. There should be navigation support for pan, zoom, and scroll.

3.2.5 RQ HCI5-Implement orientation cues

The tool must implement orientation cues to indicate to the developer where they are in the software and how to visit other areas [55]. This is done by highlighting selected objects in both the treeviewer and the graph.

3.2.6 RQ HCI6-Implement an undo feature

The tool must implement a feature that allows a developer to easily undo operations that were performed on the representation.

3.3 Tool Support

From the features in the stand-alone viewer in the previous study that we conducted [8], and from the results from our pilot evaluation (Sect. 5.3), the tool must provide the following: display inheritance hierarchy, collapse/expand sub-substructures, control node labels, trace to code, search for an object, distinguish between private and public domains, include object type in the label, and show all of an object's outgoing and incoming edges. We will describe each one in more detail.

3.3.1 RQ TS1-Display inheritance hierarchy

The tool must display the inheritance hierarchy of the types of the field declarations that an object merges.

3.3.2 RQ TS2-Collapse/expand sub-structures

The tool must produce a object graph that the developer can collapse or expand the sub-structures of selected objects. This allows a large graph to be manageable on a normal size screen.

3.3.3 RQ TS3-Control node labels

The tool must support renaming the labels of domains and objects in the diagram.

3.3.4 RQ TS4-Trace to code

The tool must allow the developer to select an element (object or edge) in the diagram and trace to the corresponding lines of code.

3.3.5 RQ TS5-Search for an object

The developer must be able to search for an object in the visualization by name or by type.

3.3.6 RQ TS6-Distinguish between private and public domains

The tool must differentiate between public and private domains throughout the visualization.

3.3.7 RQ TS7-Include object type in the label

The tool must include the type of the object along with the object's name in the label of an object.

3.3.8 RQ TS8-Show all of an object's outgoing and incoming edges

The tool should include a feature that allows the developer to see all of the relationships that an object has. This will help a developer to decide where to move an object.

3.3.9 RQ TS9-Collapse to domains

The developer must be able to collapse a graph to just reveal the top-level domains. This feature would allow the developer to get a general overview of the system's architecture.

3.4 Iterative Refinement

To allow for iterative refinement, our tool must support the following features; manipulate the ownership hierarchy, manipulate domains, abstract objects by type, summarize objects as connectors, lift edges, and support making an object be "shared". We will discuss each one in more detail.

3.4.1 RQ IR1-Manipulate the ownership hierarchy

The developer must be able to change the way that objects are grouped into components. This requires functionality to allow the developer to move an object from one domain to another (Fig. 3.2). If an object is in a higher level tier, but the developer thinks that the object is less relevant and should be in a lower-level tier, the developer must be able to move the object into a lower level tier, and vice versa (Fig. 3.1).

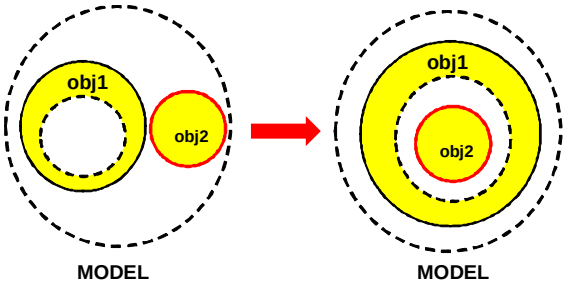

Figure 3.1: The developer moves the object `obj2` from a top-level domain to a lower-level domain nested inside an object.

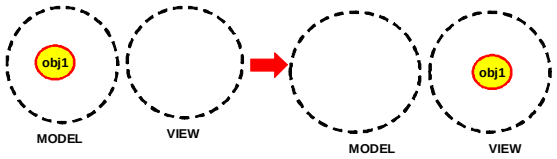

Figure 3.2: The developer moves the object `obj1` from one domain to another.

3.4.2 RQ IR2-Manipulate domains

The developer must be able to create new public and private domains in any object that the developer considers appropriate (Fig. 3.3), combine domains (Fig. 3.4), and split domains (Fig. 3.5).

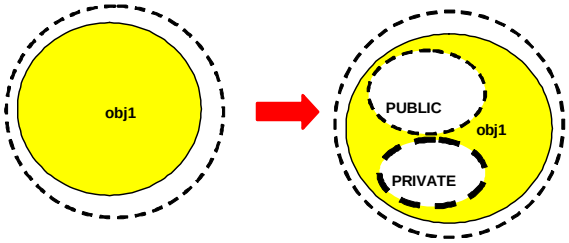

Figure 3.3: The developer adds the domains `PRIVATE` and `PUBLIC` to the object `obj1`.

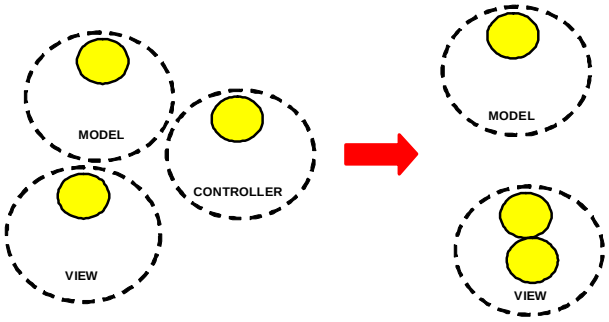

Figure 3.4: The developer merges the domains `VIEW` and `CONTROLLER` into `VIEW`, while keeping `MODEL` unchanged.

3.4.3 RQ IR3-Abstract objects by type

The initial extracted object graph creates a component for every type of object at every level in which it is created. However, often times, many types play the same architectural role. So one of the requirements for iterative refinement is that the tool

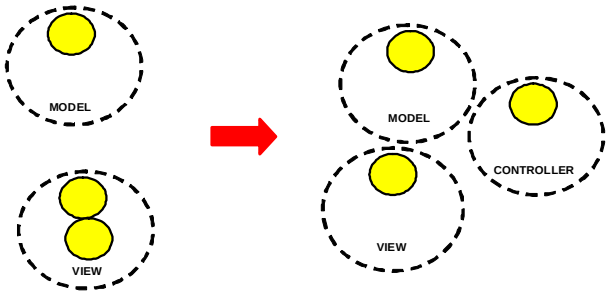

Figure 3.5: The developer splits the `VIEW` domain into `VIEW` and `CONTROLLER`, while keeping `MODEL` unchanged.

supports operations that merge components into one (Fig. 3.6), and split a conceptual component into several components, each of which includes different types of objects (Fig. 3.7).

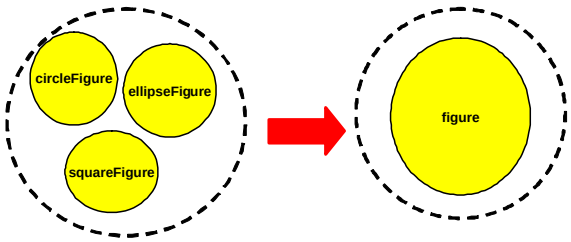

Figure 3.6: The developer merges the objects `circleFigure`, `ellipseFigure` and `squareFigure` into `figure` object.

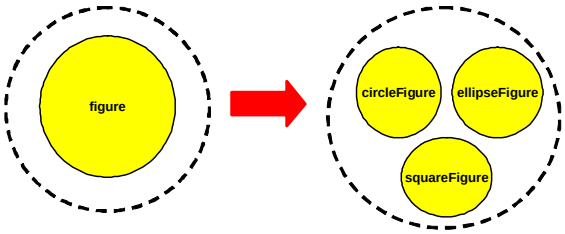

Figure 3.7: The developer splits the object `figure` into `circleFigure`, `ellipseFigure` and `squareFigure` objects.

3.4.4 RQ IR4-Summarize objects as connectors

We often treat connectors as mere references from one object to the next. However, often higher-level connectors are really implemented by some objects in the program:

Examples include buffers or streams. Thus, it may be necessary to provide an operation for treating one or more objects as a connector.

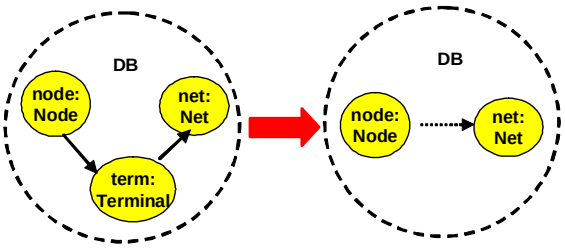

Figure 3.8: The developer elides the `term` object, which leads to a summary edge between the `node` and `net` objects.

3.4.5 RQ IR5-Support lifting edges

Because in our graph some objects can be hidden while others are shown, there could be relationships between hidden and exposed objects (Fig. 3.9). In such cases, the edge is lifted to the parent of the hidden node (Fig. 3.10). The tool does this automatically whenever the user collapses or expands objects. Edge lifting is a visualization feature commonly employed in hierarchical representations [17].

The definition of edge lifting is, if node x has an edge to node y, and x is a descendant of PX and y is a descendant of PY, then we lift the edge (x, y) to (PX, PY) only if PX and PY are distinct nodes and PX is not a descendant or ancestor of PY [17].

3.4.6 RQ IR6-Make an object shared

Some objects, such as Strings, are treated as shared, globally aliased references. So, the tool must allow a developer to mark an object as "shared" by moving it into the `shared` domain. The `shared` domain must be a top-level domain, and one which developers cannot add or delete.

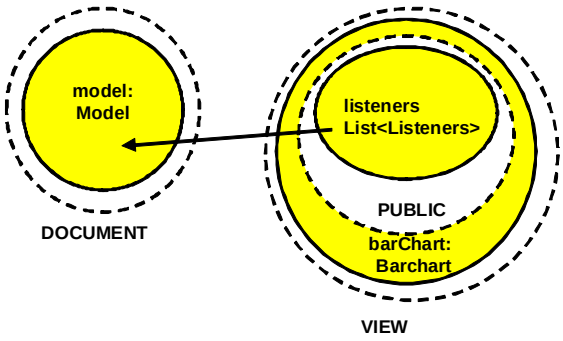

Figure 3.9: There is an edge between `listeners` which is nested in `barchart`, and `model`.

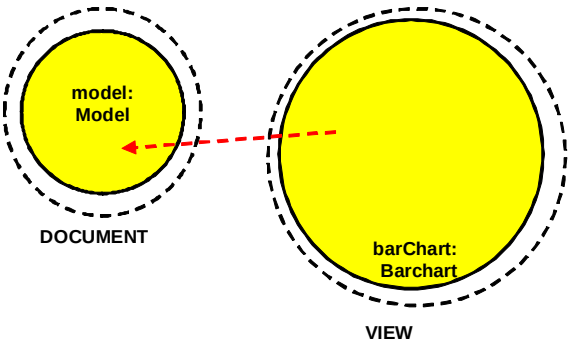

Figure 3.10: `listeners` is hidden, so the edge is lifted to the parent `barchart` which is exposed.

Chapter 4: Implementation

In this chapter, we will discuss how we implemented the tool based on the requirements (Chapter 3)

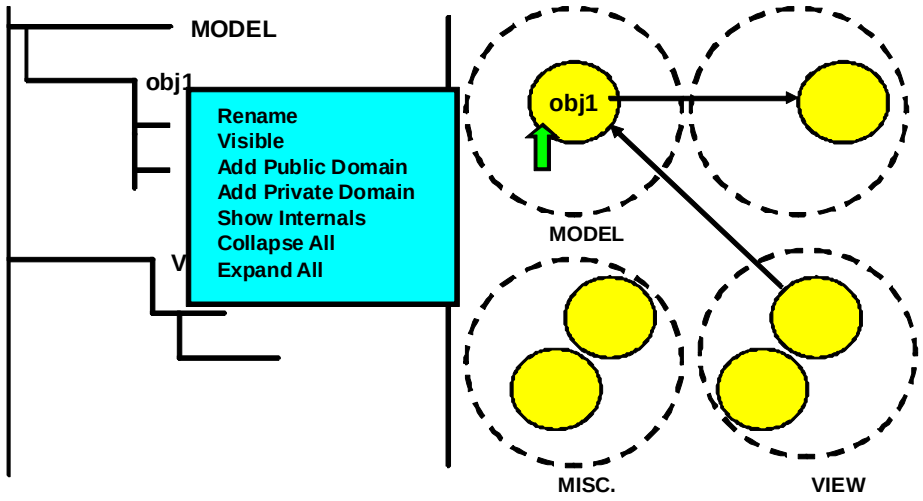

Figure 4.1: Prototype for OOGIE

4.1 Tool Implementation Overview

We implemented the interactive editor as a split-panel user interface. The left-hand side is a tree visualization of the ownership structure. The right-hand side is a graph visualization with nested circles to indicate containment (Fig. 4.1).

4.1.1 Treeviewer

For the treeviewer, we used the JFace UI toolkit [27], which is designed to simplify tasks related to building user interfaces, such as populating and updating widgets. It separates the data model from the user interface implementation. This is what allows the graph and the treeviewer to work in sync, since they are using the same data model. In the treeviewer, right-hand clicking on an object or domain will produce a

context menu with a list of options on ways to manipulate the visualization, such as renaming domains, or expanding/hiding object sub-structures (Fig. 4.1).

4.1.2 Graph

To represent the graph, we used the Prefuse framework [2] which is an extensible software visualization framework. It can create interactive information visualization applications. Right-clicking on objects can be used to select multiple objects or multiple domains but not both. Right-clicking on an object or domain will produce a context menu that gives the developer a list of options on how to manipulate the object graph, similar to the context menu in the treeviewer. Selecting an object or domain in the graph also highlights the object or domain in the treeviewer.

4.1.3 Motivation

Our motivation for implementing the interactive editor with both a treeviewer and a corresponding graph is that it allows for developers with different preferences to interact with the graph in different ways. In some ways, the treeviewer is independent from the graph. If a node is expanded/collapsed in the graph, it is not automatically expanded/collapsed in the treeviewer. Having a treeviewer be independent in this way also allows for information hiding. This is because it allows for developers to examine sub-structures without exposing them in the graph, and keeping the graph formation as is. In the future, we plan on allowing developers to hide an object in the graph while still displaying it in the treeviewer. In this way, soundness is preserved. In this way, soundness is preserved. Our motivation for highlighting objects in the tree that have been selected in the graph, and vice versa, is that it helps deal with the problem of scalability. Large graphs can be difficult to navigate. If a developer finds an object or domain in the graph to be examined in more detail, they do not have to search the tree in order to manipulate it, and vice-versa. Our motivation for allowing

the developer to manipulate the graph from the treeviewer or the graph itself, is that it improves the developer's speed and performance, since they do not have to go back in forth between the two views. Also, the treeviewer scales more than the Prefuse graph, in the sense, that screen real estate allows expanding only a limited number of nodes in Prefuse, whereas the entire tree can be expanded and still fit within the same space, due to the use of vertical and horizontal scroll bars.

4.2 Tool Features

- **Select a component:** Selecting an object, domain, or edge in the tree selects the corresponding component in the graph, and vice versa. This feature allows the developer to use the tool more efficiently, since it will allow them to quickly go back and forth between the tree and the graph, and choose which view they prefer to manipulate. The tool also supports multiple selection of objects and domains (not both).

- **Rename domains:** Since domains are conceptual groups of objects that are just runtime abstractions, they can be renamed if the developer sees fit. The domains are initially added as annotations in the code. But the developer can rename the domains after the object graph has been extracted. Developers can also rename domains that they created during the manipulation of the object graph.

- **Drag-and-drop objects:** The developer can move objects between domains by dragging them from one domain in the tree to another (Fig. 4.2). This allows for developers to choose whether the objects are more important than their location in the initial graph indicates, and should therefore be moved into a higher-level domain, or vice-versa. This change in the tree is reflected in the graph so that developers can better visualize the changes that they have made.

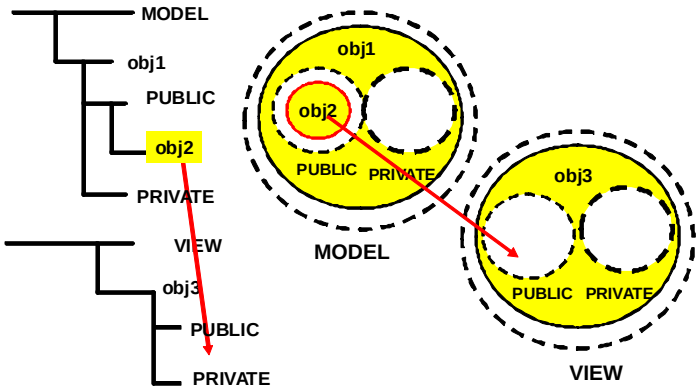

Figure 4.2: Moving objects in the tree from one domain to the other is reflected in the graph

- **Collapse/expand domains:** The developer can choose to collapse the graph to just show the top-level domains

- **Collapse/expand objects:** The developer can choose an object to examine in more detail. The sub-structure of objects are initially hidden, but through a context menu in the tree or graph, the developer can choose to view or collapse the sub-structure of an object.

- **Collapse all/expand all:** The developer can choose to show all of the graph up until a certain level. This feature will display all of the objects and all of their sub-structures up until the pre-determined level. The reason that the developer can see only a certain number of levels at the time is that large graphs are unusable and cannot always fit on a normal size screen. The developer can also choose to collapse all of the objects so that only the top-level domains are displayed.

4.3 Future tool features

We plan to implement in the future a tabbed window that will allow the developer to examine the different layers of an object. For example, the first tab might show the top level domains with top level objects (Fig. 4.3), and then the next tab might show the sub-structure of one of the objects (Fig. 4.4). This allows the developer to incrementally explore the different layers of the abstraction.

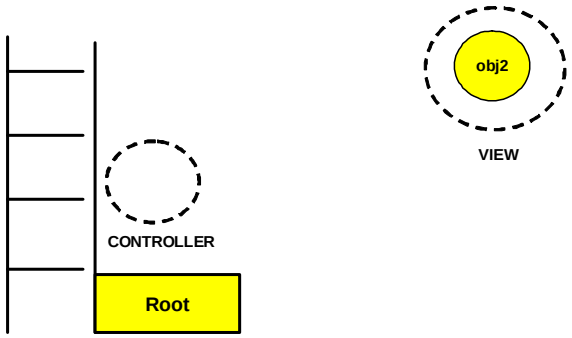

Figure 4.3: Tabbed editor showing the top-level domains.

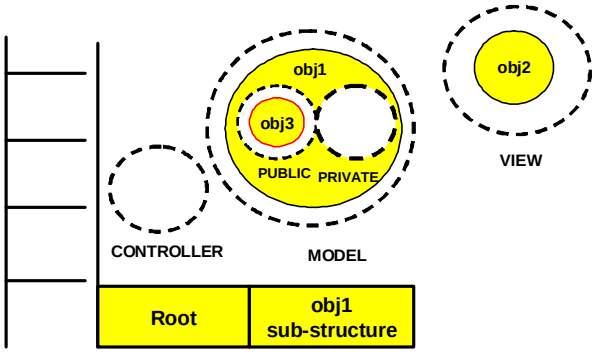

Figure 4.4: Tabbed editor showing the sub-structure of **obj1**.

4.4 Example

Like in the earlier example (see Sec. 1.5), we will illustrate our approach on Micro-Draw.

Flat graph. Many tools extract flat object graphs [25, 57], which are often overly complex for developers to navigate and use. An automated analysis however can infer that `fSelectionListener` is strictly encapsulated in the `drawingView` object (Fig. 4.5).

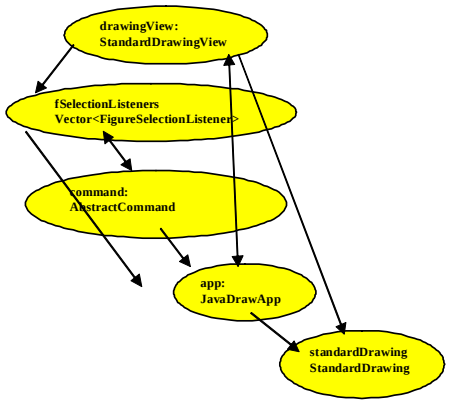

Figure 4.5: Flat object graph of MicroDraw.

Initial object graph. Even an automated extraction algorithm will extract a mostly flat object graph, where all the objects are in one top-level tier (Fig. 4.6), since the architectural intent of multiple tiers does not exist in the code. When the developer decides to convey the Model-View-Controller design pattern, she renames the top-level tier to `MODEL`.

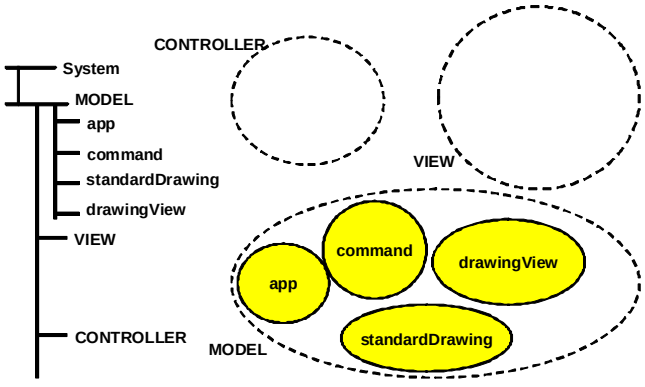

Figure 4.6: Initial extracted object graph of MicroDraw.

Creating domains. She also adds two other top-level domains, `VIEW` and `CONTROLLER`. This is demonstrated in the figure below (Fig. 4.7).

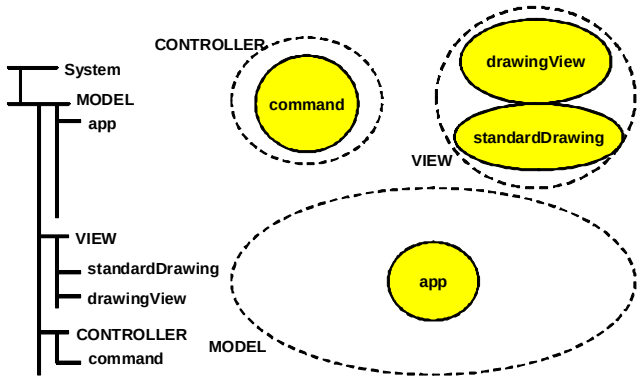

Figure 4.7: Refined object graph with added top-level domains `VIEW` and `CONTROLLER`.

Moving objects. The developer then moves the `drawingView` and the standardDrawing objects into the `VIEW` domain, and moves the `command` object into the `CONTROLLER` domain. This is represented in the figure below (Fig. 4.8).

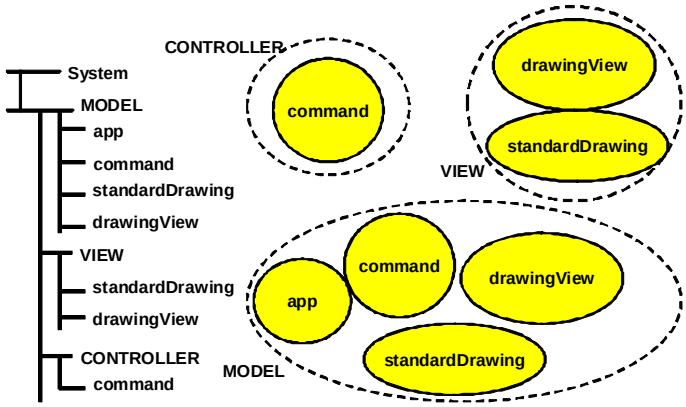

Figure 4.8: Refined object graph with objects moved from the `MODEL` domain into the top-level domains `VIEW` and `CONTROLLER`.

Controlling level of detail. Then the developer decides to examine the `drawingView` object in more detail and exposes its sub-structure (Fig. 4.9). For example, this diagram highlights that the `drawingView` listens to notifications from

other objects such as `command`. This information would be valuable for another developer performing a code modification task.

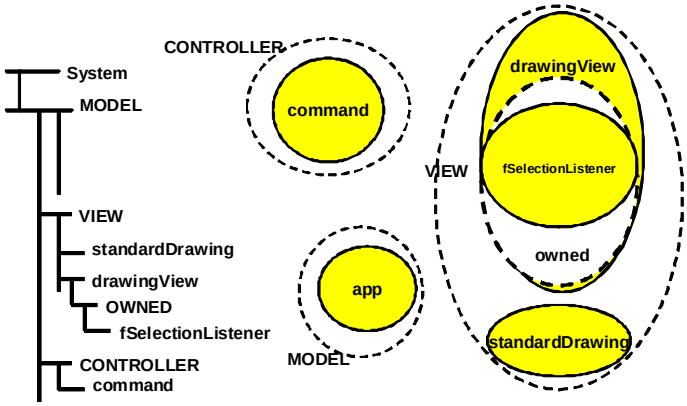

Figure 4.9: Object `drawingView`'s sub-structure is exposed.

4.5 Ownership Domains Data Model

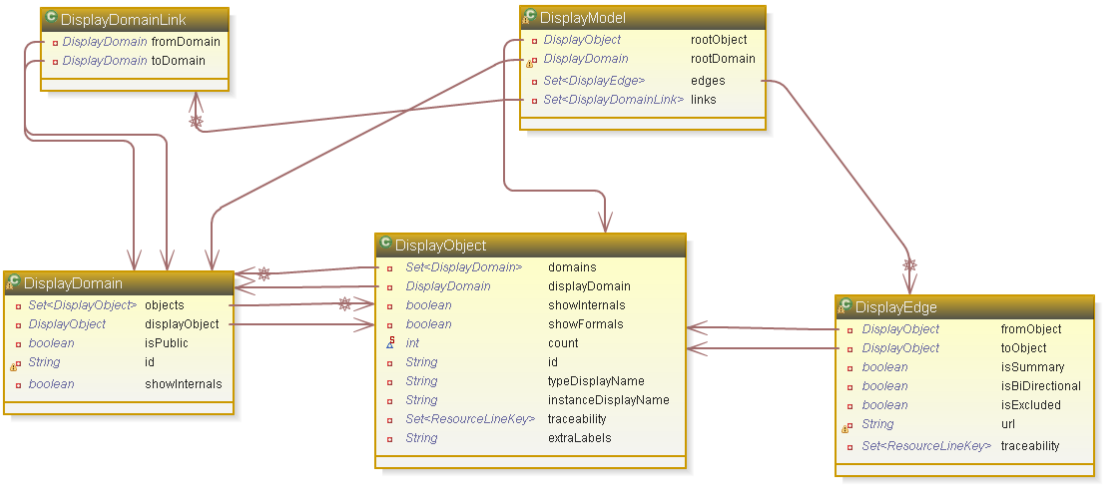

Figure 4.10: Data model for OOGIE extracted using AgileJ [9].

The data model is the domain logic which is separate from the user interface. The same data model is used in the treeviewer and in the graph, so data model changes are represented in both the treeviewer and the graph. Fig. 4.10 is a representation of the ownership domains data model using a UML class diagram. It is made up of the following elements:

- **DisplayModel:** A `DisplayModel` contains a root `DisplayObject`. The root `DisplayObject` has nested zero/or more `DisplayDomains`, which has nested zero/or more `DisplayObjects`, so by containing the root `DisplayObject`, it also contains the set of all `DisplayObjects`, and the set of all `DisplayDomains`. It also contains the set of all `DisplayDomainLinks` and the set of all `DisplayEdges`.
- **DisplayDomainLink:** `DisplayDomainLinks` indicate relationships between `DisplayDomains`.
- **DisplayEdge:** `DisplayEdges` indicate points-to relationships between `DisplayObjects`.

- **DisplayDomain:** `DisplayDomain`s represent ownership domains. A `DisplayDomain` does not directly contain other `DisplayDomain`s. A `DisplayDomain` has a single parent `DisplayObject`, and can contain zero/or more `DisplayObject`s.

- **DisplayObject:** `DisplayObject`s represent runtime objects. A `DisplayObject` does not directly contain other `DisplayObject`s, and can contain zero/or more `DisplayDomain`s.

4.6 System documentation

We represented the implementation of the tool as object diagrams.

`nestedObjectDiagram` (Fig. 4.11) is the main object of system. It is the class that builds the graph. `graphAdapter` converts the data model in the `treeviewer` into a format that is usable by the graph. This allows the treeviewer and the graph to work in sync. `collectiveLayoutActivity` collects together all of the objects that are responsible for the layout of the graph, and turns off the layouts when changes are made to the graph to prevent concurrent modification exceptions. `parameterObject` puts together some of the objects that are used by many other objects. `prefuseMakeInitial` builds the initial view of the graph.

`nestedObjectDiagram` instantiates `nestedControlObject` (Fig. 4.12), which is responsible for responding to users interacting with the graph, including producing the context menu that appears when users click on the graph, and also highlighting selected graph components, when a component is selected. When a developer right-clicks on domains and objects in the graph, `popUpMenu` is instantiated, which produces a context menu that gives developers a list of ways to manipulate the representation.

`codeWizard3` (Fig. 4.13) builds the treeviewer and it instantiates the `nestedObjectDiagram` object. It also instantiates the data model that is used to represent the different components in the runtime structure. It instantiates the classes that respond to users interacting with the treeviewer, including producing the context menu when components are selected in the treeviewer, and also highlighting the selected components in the treeviewer. When a developer right-clicks on an object or domain in the treeviewer, a context menu appears which gives the developer a list of way to manipulate the representation.

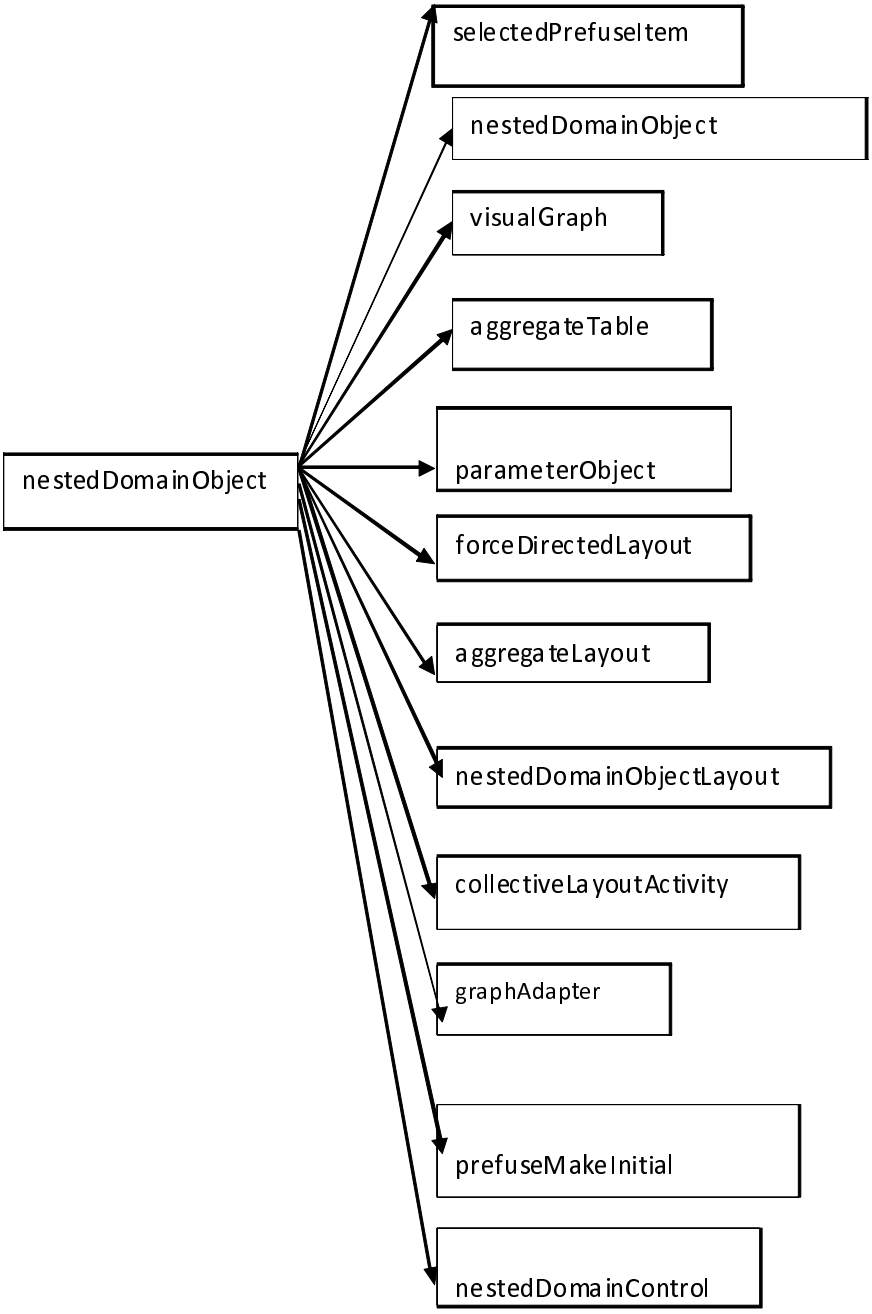

Figure 4.11: Object diagram `nestedObjectDomain`.

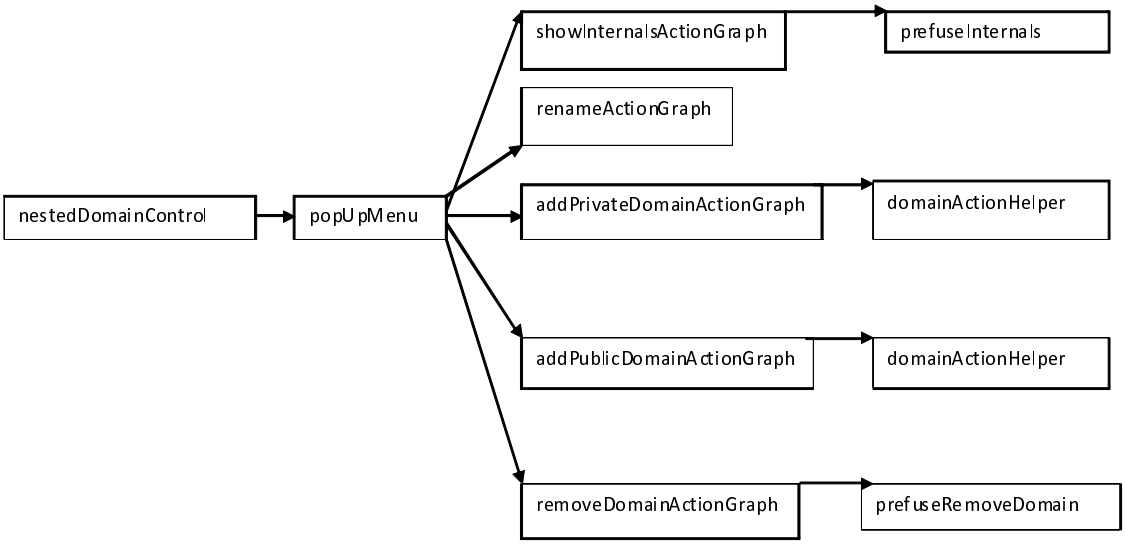

Figure 4.12: Object diagram `nestedControlObject`.

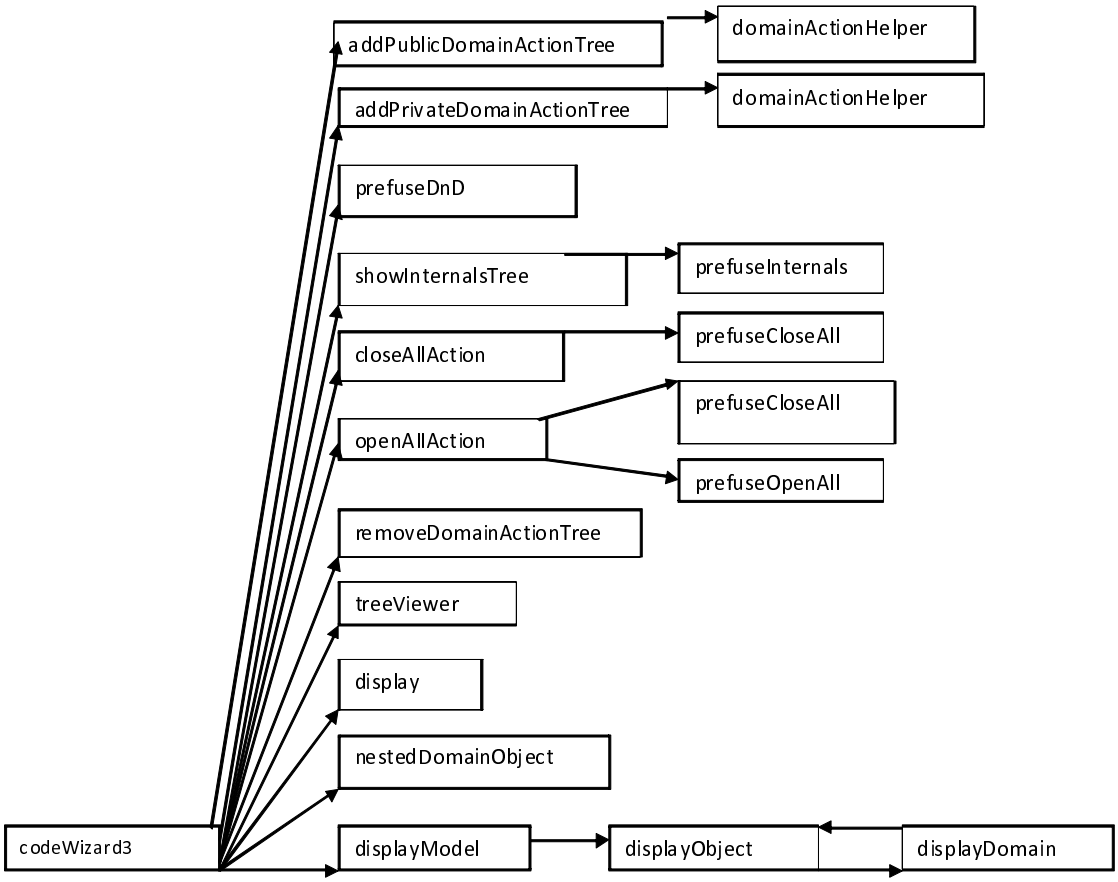

Figure 4.13: Object diagram for `codeWizard3`.

Chapter 5: Evaluation

5.1 Self-Evaluation using Cognitive Dimensions

One framework that can be used for evaluating interactive devices that uses HCI cognitive models is the Cognitive Dimensions Framework [21]. In this section, we use the Cognitive Dimensions Framework to evaluate OOGIE for quality attributes. We followed Scaffidi et al. [47] to tailor the framework for software visualization. This analysis was useful to find certain limitations in the tool that should be addressed before conducting a larger evaluation.

Closeness of Mapping. Closeness of mapping refers to the mapping between the problem and the system's representation of the problem. OOGIE has a strong closeness of mapping to the problem domain because the initial extracted object graph does match closely the developer's mental model [8], and our tool adds support to refine the object graph to match their mental model to an even higher degree. But because we have not yet demonstrated that OOGIE preserve soundness, OOGIE does not guarantee that the refinements a developer makes to the abstraction of the runtime structure, will always be faithful to the actual runtime structure.

Abstraction Gradient. Abstraction refers to putting elements together to be represented as a single entity. Abstraction gradient refers to how much the system allows for more abstraction. OOGIE gives the developer the opportunity to manipulate the hierarchy by grouping more objects together which changes the level of abstraction. The developer can also choose how much abstraction the graph should represent. Developers can choose a high-level view which has a high degree of abstraction, or they can choose to examine all of the sub-structures which has a low level of abstraction. The developer can also change the abstraction through the manipulation of domains.

This indicates that OOGIE is an abstraction-hungry tool [21].

Visibility. The visibility dimension refers to whether necessary information is easily to access without the developer having to put in a lot of effort. More specifically, it is the number of steps necessary to make something visible. OOGIE has some limitations when it comes to visibility. A developer can easily examine an object's sub-structure by selecting an object and clicking on the context menu. But a developer cannot yet make a hidden node visible, without opening its parent's sub-structure. This means that in order to make a hidden object visible, a developer must first find its parent object and then open the sub-structure of the parent object. This creates cognitive overhead, and must be addressed.

Juxtaposability. Juxtaposability refers to seeing two different parts of a program next to each other. OOGIE supports this property by having a treeviewer which represents the runtime structure next to the graphical representation of the runtime structure. For example, the developer could look at a graphical representation of the runtime structure at its highest level, but still examine the sub-structure of the runtime structure through the treeviewer. Another example of juxtaposability is the trace-to-code feature since the developer can examine a high-level view of the runtime structure, but also examine the underlying lines of code that corresponds to it.

Error-proneness. Error-proneness refers to reducing the potential for programmers to make analysis errors. OOGIE does take many steps to prevent the developer from making errors, but it could take more. OOGIE does prevent errors by turning off the feature of domain creation when a domain is selected since they can only be added to objects. The feature to rename domains is turned off when objects are selected. It prevents developers from using the multiple selection feature to select both objects and domains since it should only work for objects or domains. One major

short-coming is that is does not prevent developers from adding private domains when they should be public because the objects in the domain are not strictly encapsulated.

Premature Commitment. Premature commitment refers to the developer having to make a decision about the final outcome before all of the information is available. In OOGIE, the developer is given leeway to experiment with the abstraction, and undo some changes that they made to the extracted object graph, such as change the name of a domain, or to delete a domain after it is added. But there is no feature that simply allows the developer to undo recent changes because OOGIE does not record the steps that the developer took. We plan on addressing this issue in the future. We also plan on implementing a feature that would allow developers to save their work, and reload old projects.

Viscosity. Viscosity refers to resistance to change. Too much viscosity will make a software system less usable. OOGIE was designed in a way to make it easy for developers to make changes and reduce viscosity. It makes moving object from one domain to another very easy because it is done through a drag-and-drop functionality in a tree which is intuitive. Also, in order to perform most operations, the developer must simply use the context menu in the treeviewer or in the graph. Also, because clicking on an object in the tree selects the object in the graph, and vice versa, it is easy for developers to go from one to the other. All of these features reduces cognitive overhead.

Hard Mental Operations. Hard mental operations refers to forcing developers to solve unnecessarily hard problems. OOGIE produces a default runtime structure decomposition, and makes it easy for developers to make changes to the software visualization through simple UI activities such as drag and drop and context menu clicks, which reduces the need for developers to produce hard mental operations.

Progressive Evaluation. Progressive evaluation refers to regularly allowing developers to evaluate their progress. OOGIE supports progressive evaluation because the developer can see the results of all the changes of the graph, and some of the changes are reversible, such as being able to rename domains.

5.2 Expert UI Review

We had one-on-one consultations with user interface experts at Carnegie Mellon University, with one professor of human-computer interaction and one graduate student in the field of human-computer interaction. The following is a list of their recommendations and our responses to their recommendations, which we incorporated into the requirements section (Sec. 3).

- **Right-click should be available both in the tree and on the graph:** This feature has been implemented.
- **Selecting an element in the tree should also select it in the graph:** This feature has been implemented.
- **Show top-level context menu in the graph:** This feature has been implemented.
- **Support drag-and-drop in the graph:** This feature has not been implemented because we think that it would be best to first improve problems with graph layout before we implement this feature.
- **Enable trace from diagram element to code:** This feature has been implemented.
- **Support trace from code to diagram:** This feature is left for future work. One element in the code could map to multiple elements in the diagram, so to implement this feature, it may need to suggest a list of possibilities and have the developer pick one.
- **Implement functionality to elide object:** This feature is left for future work since it is infrequently used in practice.

- **Implement check boxes to be able to re-add elided objects:** The consultants did not agree on this feature. The professor said there is no point in complicating the tree, for the rarely used feature of eliding objects.

- **Support ability to hide an object, without specifying or caring where it goes:** This is an alternative to pushing an object underneath another in which the developer must identify the owning object. The need for this feature was mentioned in a previous study that we did [8].

- **Support ability to make an object 'shared':** This feature is left for future work.

- **Pin object down for tabbed mode feature:** This feature is left for future work.

- **Support resize handle on nodes:** This feature is left for future work.

- **Support renaming domains in the graph:** This feature is left for future work.

- **Re-add pan and zoom in the graph:** This feature is left for future work. We are not sure that this feature is needed or helpful.

- **Support ability to collapse to top-level domains:** This feature is left for future work.

5.3 Pilot Evaluation

5.3.1 Method for Pilot Study

We conducted a pilot lab study in which one participant worked on one task for two hours. The participant had experience with the subject system that we used. We had the participant think-aloud during the study, and we recorded her activities through note-taking to understand how she accomplished the designated task.

Subject System

We used the DrawLets [16] runtime structure for our experiment. DrawLets is a framework for creating figures. It includes a drawing canvas which the user adds figures to. The user can interact and manipulate the figures. The figures that can be displayed includes lines, shapes, and free-hand figures. DrawLets is around 8,000 lines of code, and includes over 100 classes. It is rich in design patterns [41].

Study Design

For the pilot study, we recruited a Wayne State University masters student who has experience working with DrawLets. We designed the pilot in this way because we wanted to get a picture of how a person with knowledge about a system, enough to form a mental model, could use our tool to iteratively refine an ownership object graph, and we did not have to rely on the participant's word about their skill level, and did not want to train someone in the subject system. We chose one participant instead of many because it is a pilot study, and just wanted to make sure that our study design is correct and find changes that need to made to the tool before conducting a larger evaluation.

Participant

The participant spent many months of research using DrawLets. She has a few years of industry experience, in which she coded in the Java programming language. She is knowledgeable about object graphs and has used them in the past, and is familiar with the concept of ownership domains.

Tasks

We had the participant do a number of preliminary activities including adding a domain, moving an object from one domain to another, renaming a domain, and expanding/collapsing an object, so that we could document these activities for future qualitative analysis. We also had the participant refine the initially extracted object graph using OOGIE. We did not prep the participant on how to use OOGIE, because we want to make sure that the tool is intuitive, and not difficult for novice users to learn. We had the participant do a think-aloud while she was refining an initial object graph so that we could later perform a qualitative analysis.

Tools and Instrumentation

The participant was provided with an Eclipse IDE. The participant was allowed to view Java and DrawLets documentation. OOGIE is an Eclipse plug-in, so the participant used an Eclipse workspace. She was asked to think-aloud during the experiment. She was prompted regularly with the reminder, "Please think aloud". We took notes on what she said during the course of the experiment on her activities.

Procedure

The participant was not briefed or prepped for this evaluation since we already knew that she is knowledgeable about DrawLets and the Eclipse IDE.

First, the participant was given a list of activities to complete using OOGIE. These activities include adding a domain, moving objects from one domain to another, and renaming domains. She was given 30 minutes to complete these tasks, though she required a lot less.

Next the participant was asked to use OOGIE to refine the initial extracted model to better match her mental model. This portion took one hour.

5.3.2 Results of Pilot Study

The participant was able to complete the task of manipulating the initial object graph to better fit her mental model. She was able to figure out how to use OOGIE without much trouble, indicating that the tool is intuitive. She was able to understand the hierarchical structure of the graph, indicating that she also understood ownership hierarchy. She also was able to understand abstraction by types. We met all of the success criteria from the hypothesis.

The participant made the following suggestions which we incorporated into the requirements section (Sec. 3):

- **Fix layout:** The participant had trouble figuring out whether nodes were in different aggregates that overlapped, or whether nodes were in the same aggregate. This suggests that the layout requires some fine-tuning in order to clearly represent what is intended. We have not addressed this issue yet, but we plan to in the future.

- **Distinguish between private and public domains in the treeviewer:** The participant wanted the treeviewer to indicate whether a domain was private or public. We addressed this issue by making the names of domains blue and the names of objects black in the treeviewer.

- **Add more features in graph:** She also wanted to invoke the right-click

context menu from the graph, and not just the tree. We have since implemented this feature.

- **Include object type in the label:** The participant thought that it was difficult to understand what each object was without the type being included in the label. We addressed this concern by including the type of object in the treeviewer.

- **Add feature to show all of an object's outgoing and incoming edges:** The participant wanted to be able to see all of the relationships that an object had so that she could move the object into another object's domain that it was related to.

During the second half of the experiment, the participant spent most of the time exploring the runtime structure visualization. There were components that she felt were misplaced. More specifically, she felt that the `toolbar` object, which was located in the top-level domain, should be more nested. In order to verify her hypothesis, she asked for different views of the runtime structure.

With these resources she was able to confirm her hypothesis that a component was misplaced. So she moved the component to a better location. This indicates that OOGIE is capable of helping developers to refine a software visualization graph to better fit their mental model.

Sometimes she wanted to see versions of the runtime structure visualization with more nodes, and sometimes versions of the runtime structure visualization with less. This indicates that it is useful to have a tool that can easily switch between views with more or less abstraction.

5.3.3 Threats to Validity

The experiment was only a pilot, and the results are only helpful for designing the next experiment and figuring out some limitations of the tool that needs to be addressed. Some of the threats to validity include the fact that there was only one participant. Many more are needed to draw real conclusions about the usefulness and usability of the tool. Another threat to validity is that we did not use a video or an audio recorder. Instead, we recorded the participant's activities and think-aloud by hand. This made it hard to verify whether the experimenter was correct in what they recorded, and whether they recorded everything. Another issue is that we did not have a control group to test a tool that extracts flat object graphs and to see whether our tool is better at helping developers realize their mental model of the runtime structure than the flat object graph extractor. Also, the participant was already familiar with ownership object graphs, so it does not help us understand whether the concept can be easily adapted by developers without previous knowledge of ownership object graphs. The participant also had previously worked with the Scholia approach so she already had experience working with tools that visualize ownership object graphs, so the pilot does not demonstrate that the tool is intuitive for a novice user. Despite these threats, the experiment did indicate to us what features we needed to add to the tool for a larger experiment. In the future, we plan on conducting a larger evaluation with participants who are not familiar with ownership object graphs.

5.4 Future Evaluation Example

Abi-Antoun and Nammar [5, 6] extracted various OOGs from a real object-oriented system, DrawLets. They initially extracted an OOG, then refined the OOG to better fit the mental model of a developer performing code modification tasks. We studied the evolution of the DrawLets OOGs to determine if OOGIE could be used to refine

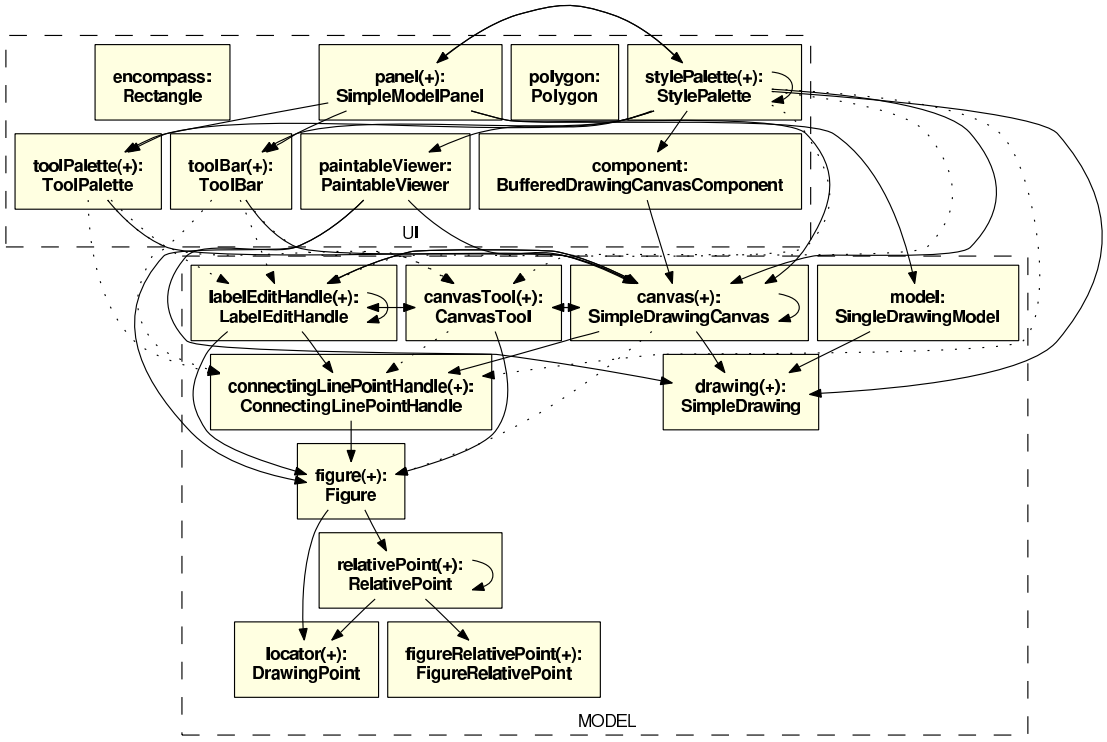

Figure 5.1: Extracted OOG with less relevant objects showing in the top-level domains.

the OOG by direct manipulation. Figs. 5.1, and 5.2 were not produced from OOGIE, but from the previous tool, which is a read-only viewer. They are just used to demonstrate how a previous evaluation that was done on the stand-alone viewer from previous work (Sec. 1.4) could be used on OOGIE to show what kind of evaluations we plan on conducting in the future. More specifically, we will discuss how participants could use the different features in OOGIE to refine an initial Drawlets OOG (Fig. 5.1) into one that conveys the DrawLets architectural intent (Fig. 5.2).

Abstraction by types

A participant decides that though the `panel:Panel` object was the result of merging objects that are in the same domain and have the `Panel` supertype, it is actually more helpful to show these objects. The participant could accomplish this using OOGIE (this feature has not yet been implemented but is listed in the requirements).

A participant decides that the objects `relativePoint:RelativePoint`, `figureRelativePoint:FigureRelativePoint`, and `locator:DrawingPoint` are not architecturally significant, and since they are in the same domain and share the same supertype `Locator`, they could be merged into one object called `locator:Locator`. This could be accomplished using OOGIE (this feature has not yet been implemented but is listed in the requirements).

Make an object shared

A participant decides that the `encompass:Rectangle` and `polygon:Polygon` objects are not architecturally significant enough to be in top-level domains. OOGIE would help the developer move these objects into a `shared` domain (this feature has not yet been implemented).

Manipulate domains and the ownership hierarchy

A participant decides that the OOG should convey the observer pattern. The participant decides to add a `SUBS` public domain inside `SimpleModelPanel` object and to move the observers into the new domain. The participant could use OOGIE to do this by selecting the `SimpleModelPanel` object, and then choosing "Add Public Domain" in the context menu. The participant could then use the drag-and-drop feature in the treeviewer to move the observer objects into the `SUBS` domain.

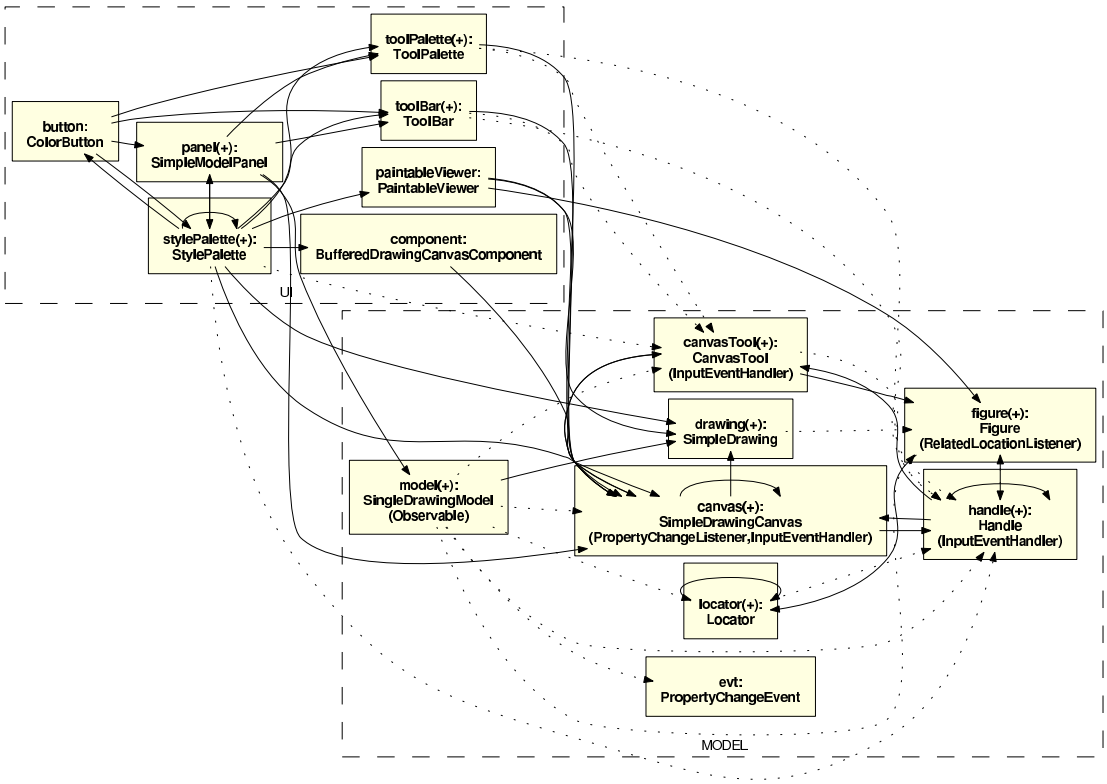

Figure 5.2: OOG after being refined based on input by the developer.

Chapter 6: Discussion and Conclusion

6.1 Validation of Hypotheses

The goal of this thesis was to present the front-end of a tool that could help developers iteratively refine an object graph using directly manipulation. We divided our goal into four hypotheses. In the following section, we will discuss how we satisfied these hypotheses.

6.1.1 H1: Ownership hierarchy manipulation

We added a number of features in order to allow developers to manipulate the ownership hierarchy. For one, we allow developers to add/remove domains, to enable developers to move objects up and down the hierarchy. We also allow developers to move objects from one domain to another through a drag-and-drop feature in the tree. Our evidence that developers can use these features is from the pilot evaluation (Sec. 5.3).

Our participant was able to add/remove domains, and to move objects from one domain to another. The participant was able to use the features to better fit her mental model of the runtime structure because she found an object that she felt was less architecturally relevant, and she was able to move the object to a lower-level domain in the hierarchy. This indicates that this hypothesis was satisfied.

6.1.2 H2: Level of detail control

In order to allow developers to control the level of detail of a runtime structure, we added a feature to allow developers to expand/collapse the sub-structure of an object. We have not yet implemented abstraction by types, so this hypothesis was not completely satisfied. During the pilot evaluation (Sec. 5.3), we found that the

developer was able to expand/collapse objects to change the level of visual detail in the representation.

6.1.3 H3: Direct manipulation to accomplish H1 and H2

All of the features previously listed (Sec. 6.1.1 and Sec. 6.1.2) are supported by direct manipulation, either in the treeviewer or in the graph. The developer can either use a context menu which appears in the treeviewer and in the graph, or use a drag-and-drop feature in the treeviewer to move objects from one domain to another. These types of manipulation are easy for developers and do not require much cognitive overhead. A self-evaluation based on cognitive dimensions (Sec. 5.1) indicated that these features do not require much cognitive overhead. In the pilot evaluation, when the participant was asked to add/remove domains, move objects from one domain to another, and collapse/expand objects, she was able to do so without being taught how. This indicates that the tool's features are easy to use, assuming that we properly accounted for the threats to validity. This is evidence that this hypothesis has been satisfied.

6.1.4 H4: User-friendly to developers

OOGIE includes features to make it more user-friendly for developers. This includes highlighting components that are selected in both the treeviewer and in the graph. This improves the developers ability to navigate the representation. We also chose different colors for different kinds of components, including the color used to highlight a component that has been selected. This makes it easier for developers to distinguish between components, reducing cognitive overhead. What makes the tool less user-friendly are the problems related to graph layout (Sec. 6.3.1). This issue came up during the pilot evaluation. Another problem is that developers cannot yet undo manipulations that they made to the representation. This was pointed out during

the self-evaluation (Sec. 5.1). This indicates that this hypothesis has not yet been completely satisfied, and in the future, the user-friendliness of the tool needs to be improved.

6.2 Missing Back-End

The present work focused on the front-end or user interface of an interactive editor for manipulating object graphs. As a result, the current implementation is not that of a complete tool, in that it still lacks the back-end processing.

6.2.1 Convert edits into annotations

Currently, OOGIE does not convert the developer's graphical edits into annotations. OOGIE simply records the changes to the ownership relationships. The expectation is that a person who knowledgeable with the annotation process changes the ownership annotations consistently to reflect the corrected ownership relationship, then re-extracts a hierarchical object graph. This issue makes using OOGIE in a production environment tedious and time-consuming. However, OOGIE is currently not unlike dynamic analysis tools, which instrument a system, and allow a user to manipulate one or more traces of execution.

6.2.2 Maintain diagram soundness

The current implementation makes no guarantees of preserving the diagram's soundness for all of the iterative refinement operations (Sec. 3.4). Moving an object to an "outer level" or inside another object may have many effects, because of the notion of ownership parameters. In some cases, the tool may need to update many edges associated with the object that was moved. We also cannot simply give an input to OOGIE with a list of allowable changes at this point because moving an object in or

out of a domain can cause extra edges due to domain parameters.

For future work, for each operation we provide, we will formally specify an algorithm for performing that operation on a portion of a diagram, resulting in a new diagram. We will then prove that each and all of our operations preserve soundness: if the original diagram is a sound abstraction of the runtime structure, then the updated diagram will be sound as well.

6.2.3 Reflect concurrent changes to the code

We will research a mechanism to update the extracted object graph if the code changes. In this way, the developer can get an updated view of the architecture without having to redo all the operations that transformed the original structure into one that better represents the architect's intent.

6.3 Current Limitations in Front-End

The current OOGIE front-end still has some important limitations.

6.3.1 Graph Layout

Graph layout is a very difficult problem. We used Prefuse's force-directed layout. There is still a good deal of overlapping of objects and domains. This creates confusion and cognitive overhead. The force-directed layout also produces too much movement. In the future we plan on improving the layout by reducing the movement and making the representation of the relationships of the domains and the objects clearly defined. We will also use an algorithm that places the components in a way that is easy for developers to understand.

6.3.2 Visualization Hacks

The layout uses many hacks. These hacks include invisible edges and invisible nodes of varying sizes. These hacks make the system more complicated and difficult to manage. They also make the forced-directed layout more difficult to manage because the invisible nodes and edges also have forces which act on other graphical components, so the components might not be visible, but their effects on other components are visible, causing a distortion. It will be harder for future developers to use and extend the system because of these hacks. In the future, better solutions to the problems that the hacks were used for, need to be implemented. This may or may not involve using a framework other than Prefuse.

6.4 Satisfaction of the Requirements

In this section we discuss how the requirements from Chapter 3 were satisfied.

6.4.1 General Requirements

- **RQ 01-Support iterative refinement:** See Sec. 6.4.4.
- **RQ 02-Must be an Eclipse plug-in:** OOGIE satisfies this requirement.
- **RQ 03-Must load and save states:** To load and save states, the data model is persisted to XML.
- **RQ 04-Must be easy to use:** OOGIE satisfies this requirement through the fulfillment of requirements HCI1,HCI4,HCI5.
- **RQ 05-Must invalidate incorrect manipulations of the OOG:** OOGIE partially satisfies this requirement. It does not allow developers to add domains to other domains. It does not allow developers to give domains names that are already used by other domains. It does not allow developers to delete non-empty

domains. But it does not yet check to make sure that only strict encapsulated objects are in private domains. We discussed previously the reason that this has not been implemented (Sec. 6.2.2).

- **RQ 06-Support the data model of ownership domains:** OOGIE satisfies this requirement because it supports all of the components of the ownership domain data model (Sec. 4.5).

- **RQ 07-Maintain diagram soundness:** OOGIE partially satisfies this requirement. It does not allow developers to add/delete an object or edge.

6.4.2 Human Computer Interaction Requirements

- **RQ HCI1-Create visual distance between semantic ideas:** OOGIE creates visual distance in many ways. The objects are yellow and the domains are white. The public domains have thin dotted lines and the private domains have thick doted lines. When clicked on, the outside domain lines turn blue, the outside object lines turn red, and the edges turn purple. In the future, we will implement the domains as squares and the objects as circles to increase visual distance.

- **RQ HCI2-Make use of hierarchies:** OOGIE satisfies this requirements by making use of hierarchy in both the treeviewer and the graph itself.

- **RQ HCI3-Limit number of on-screen components:** OOGIE will satisfy this requirement in the future by limiting the number of recursive levels the graph can expand to.

- **RQ HCI4-Implement easy navigation:** OOGIE satisfies this requirement because when the developer clicks on a component in the treeviewer, it shows

up in the graph, and vice versa. This allows the developer to go in between the graph and treeviewer easily.

- **RQ HCI5-Implement orientation cues:** OOGIE satisfies this requirement because when the developer clicks on a component, it is highlighted.
- **RQ HCI6-Implement an undo feature:** This feature was left for future work.

6.4.3 Tool Support

- **RQ TS1-Display inheritance hierarchy:** This feature is left for future work.
- **RQ TS2-Collapse/expand sub-structures:** OOGIE satisfies this requirement through context-menu features to "Expand/Collapse", and "Expand All/Collapse All" which expose or hide the sub-structures of selected objects or of all the objects in the diagram.
- **RQ TS3-Control node labels:** Through the rename domain feature, this requirement is satisfied.
- **RQ TS4-Trace to code:** OOGIE includes a trace to code feature.
- **RQ TS5-Search for an object:** OOGIE includes a feature to search for an object in the treeviewer. The developer types a regular expression to match an object's name or type and the treeviewer is filtered to highlight the matching elements.
- **RQ TS6-Distinguish between private and public domains:** The tool distinguishes between public and private domains in the graph by making the lines of the private domain thicker than the lines of the public domain, and the tool distinguishes between public and private domains in the treeviewer by using different colors.

- **RQ TS7-Include object type in the label:** The treeviewer shows the type of the object along with its name.
- **RQ TS8-Show all of an object's outgoing and incoming edges:** This feature is left for future work.
- **RQ TS9-collapse domains:** This feature is left for future work. The tool only collapses to the top-level domains with the top-level objects.

6.4.4 Iterative Refinement

- **RQ IR1-Manipulate the object hierarchy:** OOGIE has a drag-and-drop feature to move objects from one domain to another, which changes the object hierarchy.
- **RQ IR2-Manipulate domains:** OOGIE satisfies this requirement through the features adding domain, renaming domain,and the drag and drop of objects.
- **RQ IR3-Abstract objects by type:** OOGIE does not yet satisfy this requirement. This is because this feature requires manipulating annotations so it is out of the scope of this thesis.
- **RQ IR4-Summarize objects as connectors:** OOGIE does not yet satisfy this requirement. This feature is not often needed so it is left for future work.
- **RQ IR5-Support lifting edges:** OOGIE automatically adds and removes lifted edges, and thus satisfies this requirement.
- **RQ IR6-Make an object shared:** This feature is left for future work.

6.5 Conclusion and Broader Impact

Software architecture is important for many software evolution tasks. It serves as a road-map so that developers can orient themselves in a large code base. Software architectures can help speed up the process for developers to learn a code base well enough so that they can perform basic maintenance tasks, which saves resources such as time and money. It is not enough just for developers to understand the code structure. They must also understand the runtime structure. Runtime structure exacerbates the problems of turning large software systems into graphs because runtime structures show the runtime code components and all of their potential relationships. This creates the need to hide many parts of the runtime structure so that the developers can focus on area that they are making changes to.

The Scholia approach extracts the runtime structure and produces an ownership object graph (OOG). Because in an ownership object graph, objects contain other objects, it reduces the size of the graph that is depicted, but is still sound. But the Scholia approach was not designed with usability in mind. It produces a default decomposition. Iteratively refining the decomposition is tedious.

The front-end of OOGIE is a tool that is designed to improve the usability of the Scholia approach. It takes an initial extraction of the runtime structure, and allows developers to iteratively refine the depiction in a way that is intuitive and user-friendly. Though it is not yet integrated into the Scholia approach and takes as input an OOG in an XML file, it demonstrates that the iterative refinement of an OOG can be accomplished through direct manipulation. The methods that OOGIE uses will improve the potential and usefulness of the Scholia approach and object graph extractors in general because the nature of object graphs necessitates that they include architectural abstraction, but architectural abstraction means that many views are possible since the hierarchy is not implicit in the code. This means that developers will need to refine the abstraction of an initial object graph. The front-end

of OOGIE is presented as a way to allow them to do so.

REFERENCES

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

[2] Prefuse. `www.prefuse.org`, 2007.

[3] Abi-Antoun, M. *Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure.* PhD thesis, Carnegie Mellon University, 2009. Available as Technical Report CMU-ISR-09-119.

[4] Abi-Antoun, M., and Aldrich, J. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2009).

[5] Abi-Antoun, M., and Ammar, N. A Case Study in Evaluating the Usefulness of the Run-time Structure during Coding Tasks. In *Workshop on Human Aspects of Software Engineering (HAoSE), co-located with the ACM International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)* (2010).

[6] Abi-Antoun, M., Ammar, N., and Khazalah, F. A Case Study in Adding Ownership Domain Annotations. Tech. rep., Wayne State University, 2010.

[7] Abi-Antoun, M., and Selitsky, T. Interactive Refinement of Runtime Structure. In *Workshop on Flexible Modeling Tools (FlexiTools), co-located with the ACM International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)* (2010).

[8] Abi-Antoun, M., Selitsky, T., and LaToza, T. Developer Refinement of Runtime Architectural Structure. In *Workshop on SHAring and Reusing architectural Knowledge (SHARK)* (2010).

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

[10] Alphonce, C., and Ventura, P. QuickUML: a tool to support iterative design and code development. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2003), ACM, p. 81.

[11] Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. *Documenting Software Architecture: View and Beyond*. Addison-Wesley, 2003.

[12] De Pauw, W., Helm, R., Kimelman, D., and Vlissides, J. Visualizing the Behavior of Object-Oriented Systems. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (1993).

[13] De Pauw, W., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J. M., and Yang, J. Visualizing the Execution of Java Programs. In *Revised Lectures on Software Visualization, International Seminar* (2002).

[14] De Pauw, W., Kimelman, D., and Vlissides, J. M. Modeling Object-Oriented Program Execution. In *European Conference on Object-Oriented Programming (ECOOP)* (1994).

[15] De Pauw, W., and Sevitsky, G. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *European Conference on Object-Oriented Programming (ECOOP)* (1999).

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

[17] Fahmy, H., and Holt, R. C. Using Graph Rewriting to Specify Software Architectural Transformations. In *Automated Software Engineering* (2000).

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

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

[20] Gargiulo, J., and Mancoridis, S. Gadget: a Tool for Extracting the Dynamic Structure of Java Programs. In *Software Engineering and Knowledge Engineering* (2001).

[21] Green, T., and Petre, M. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing 7*, 2 (1996), 131–174.

[22] Gschwind, T., and Oberleitner, J. Improving Dynamic Data Analysis with Aspect-Oriented Programming. In *European Conference on Software Maintenance and Reengineering (CSMR)* (2003).

[23] Hill, T., Noble, J., and Potter, J. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *Journal of Visual Languages and Computing 13*, 3 (2002).

[24] Huang, M. L., and Eades, P. A fully animated interactive system for clustering and navigating huge graphs. In *Graph Drawing*, S. Whitesides, Ed., vol. 1547 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1998, pp. 374–383.

[25] Jackson, D., and Waingold, A. Lightweight Extraction of Object Models from Bytecode. *IEEE Transactions on Software Engineering 27*, 2 (2001).

[26] Jerding, D. F., Stasko, J. T., and Ball, T. Visualizing Interactions in Program Executions. In *International Conference on Software Engineering (ICSE)* (1997).

[27] JFace. `http://wiki.eclipse.org/JFace`.

[28] Kollman, R., Selonen, P., Stroulia, E., Systä, T., and Zundorf, A. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In *Working Conference on Reverse Engineering (WCRE)* (2002).

[29] Koskimies, K., and Mössenböck, H. Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. In *International Conference on Software Engineering (ICSE)* (1996).

[30] Kramer, C., and Prechelt, L. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. *Working Conference on Reverse Engineering (WCRE)* (1996).

[31] Krikhaar, R. L. Reverse Architecting Approach for Complex Systems. In *International Conference on Software Maintenance (ICSM)* (1997).

[32] Lange, D. B., and Nakamura, Y. Interactive Visualization of Design Patterns Can Help in Framework Understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (1995).

[33] Mitchell, N. The Runtime Structure of Object Ownership. In *European Conference on Object-Oriented Programming (ECOOP)* (2006).

[34] Moody, D. The Physics of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering* (2009), 756–779.

[35] Müller, H., and Klashinsky, K. Rigi – a System for Programming-In-The-Large. In *International Conference on Software Engineering (ICSE)* (1988).

[36] Noble, J. Visualising Objects: Abstraction, Encapsulation, Aliasing, and Ownership. In *Revised Lectures on Software Visualization, International Seminar* (2002).

[37] Oechsle, R., and Schmitt, T. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams using the Java Debug Interface (JDI). In *Revised Lectures on Software Visualization, International Seminar* (2002).

[38] Pacione, M. J., Roper, M., and Wood, M. A Novel Software Visualisation Model to Support Software Comprehension. In *Working Conference on Reverse Engineering (WCRE)* (2004).

[39] Perry, D., and Wolf, A. Foundations for the Study of Software Architecture. *Softw. Eng. Notes 17*, 4 (1992).

[40] Potanin, A., Noble, J., and Biddle, R. Checking Ownership and Confinement. *Concurrency and Computation: Practice and Experience 16*, 7 (April 2004).

[41] Rajlich, V., and Gosavi, P. A Case Study of Unanticipated Incremental Change. In *International Conference on Software Maintenance (ICSM)* (2002).

[42] Rayside, D., and Mendel, L. Object Ownership Profiling: a Technique for Finding and Fixing Memory Leaks. In *Automated Software Engineering* (2007).

[43] Rayside, D., Mendel, L., and Jackson, D. A Dynamic Analysis for Revealing Object Ownership and Sharing. In *Workshop on Dynamic Analysis (WODA)* (2006).

[44] Reiss, S. P., and Renieris, M. Jove: Java as it Happens. In *ACM Symposium on Software Visualization* (2005).

[45] Richner, T., and Ducasse, S. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. In *International Conference on Software Maintenance (ICSM)* (1999).

[46] Salah, M., and Mancoridis, S. A Hierarchy of Dynamic Software Views: From Object-Interactions to Feature-Interactions. In *International Conference on Software Maintenance (ICSM)* (2004).

[47] Scaffidi, C., Myers, B., and Shaw, M. Fast, accurate creation of data validation formats by end-user developers. *End-User Development* (2009), 242–261.

[48] Schauer, R., and Keller, R. K. Pattern Visualization for Software Comprehension. In *International Workshop on Program Comprehension (IWPC)* (1998).

[49] Sefika, M., Sane, A., and Campbell, R. H. Architecture-Oriented Visualization. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (1996).

[50] Selitsky, T. OOGIE: Ownership Object Graph Interactive Editor. In *Conference Companion for the ACM International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)* (2010), pp. 215–216.

[51] Shaw, M., and Garlan, D. *Software Architectures: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[52] Smith, M. P., and Munro, M. Runtime Visualisation of Object Oriented Software. In *VISSOFT* (2002).

[53] Souder, T., Mancoridis, S., and Salah, M. Form: a Framework for Creating Views of Program Executions. In *International Conference on Software Maintenance (ICSM)* (2001).

[54] Storey, M.-A., Best, C., and Michaud, J. SHriMP Views: An Interactive Environment for Exploring Java Programs. In *International Workshop on Program Comprehension (IWPC)* (2001).

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

[56] Storey, M.-A. D., Müller, H. A., and Wong, K. Manipulating and Documenting Software Structures. In *Software Visualization* (1998), P. Eades and K. Zhang, Eds.

[57] Waingold, A., and Lee, R. SuperWomble Manual. `http://sdg.lcs.mit.edu/womble/`, 2002.

[58] Walker, R. J., Murphy, G. C., Freeman-Benson, B., Wright, D., Swanson, D., and Isaak, J. Visualizing Dynamic Software System Information through High-Level Models. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (1998).

ABSTRACT

A FRONT-END FOR AN OWNERSHIP OBJECT GRAPH INTERACTIVE EDITOR

by

TALIA SELITSKY

December 2010

Advisor: Dr. Marwan Abi-Antoun

Major: Computer Science

Degree: Master of Science

Runtime views which show runtime structure, is a type of object graph. They show components as groups of objects and data structures. Runtime views are useful for tasks related to performance, reliability, and security. Most previous work on extracting object graphs has produced flat object graphs which are not scalable. Ownership object graphs (OOGs) increase the scalability of object graphs because it nests objects, creating hierarchy. Recent work has shown that *sound* extraction of OOGs from object-oriented systems is technically feasible. Soundness means that in any execution of the program, every object can be mapped to exactly one component in the graph. The recent work is a read-only viewer and shows a default decomposition. In order for developers to change the default decomposition, they must change the annotations. This is very tedious.

In order to allow developers to iteratively refine an OOG, we propose the front-end of an editor to support this functionality, OOGIE. The OOGIE tool only supports operations that intuitively support soundness. For example, objects cannot be deleted, and edges cannot be added. The tool allows developers two kinds of operations to change the decomposition, *abstraction by ownership hierarchy* and *abstraction by type*. Abstraction by ownership hierarchy means that the decomposition shows ar-

chitecturally significant objects near the top of the hierarchy and less architecturally significant objects such as data structures further down. Abstraction by types allow objects to be collapsed further according to their declared types. The work in this thesis is the first stage in addressing the usability problems of the read-only viewer. At this stage, OOGIE takes an XML file that contains an initial OOG produced by the extraction tool, and records the changes made to the OOG by the developer in the XML file. In the future, we plan on integrating the OOGIE tool with the extraction tool, and having OOGIE manipulate the annotations directly so that the developer does not have to. Having a user-friendly method of abstracting and manipulating OOGs increases their usefulness since developers can pick the decomposition that best suits their needs.

AUTOBIOGRAPHICAL STATEMENT

TALIA SELITSKY

EDUCATION

- Master of Science (Computer Science), November 2010
  Wayne State University, Detroit, MI, USA
- Bachelor of Arts (Political Science), June 2007
  University of Michigan, Ann Arbor, MI, USA

PUBLICATIONS

1. Abi-Antoun, M., Selitsky, T., and LaToza, T. Developer Refinement of Runtime Architectural Structure. In *Workshop on SHAring and Reusing architectural Knowledge (SHARK)* (2010).
2. Abi-Antoun, M., and Selitsky, T. Interactive Refinement of Runtime Structure. In *Workshop on Flexible Modeling Tools (FlexiTools), co-located with the ACM International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)* (2010).
3. Selitsky, T. OOGIE: Ownership Object Graph Interactive Editor. In *Conference Companion for the ACM International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)* (2010), pp. 215–216.