Interactive Refinement of Runtime Structure

Marwan Abi-Antoun Talia Selitsky
Department of Computer Science, Wayne State University
{mabiantoun, talias}@wayne.edu

Abstract

We propose the front-end editor which allows developers to refine an initial object graph into a hierarchical object graph that matches their mental model of the runtime structure. The developer can expand or collapse selected hierarchies to control the level of visual detail. Such views of the runtime structure can be useful for code modification tasks.

1. Introduction

Understanding the structure of a software system is important for many evolution tasks, such as isolating and fixing defects, adding new functionality, optimizing performance, or identifying and addressing security vulnerabilities. Software architecture addresses the need to capture the high-level structure of a software system [11] by abstracting a system's organization into components and their interactions.

The different ways of looking at a system's structure are called architectural views [4]. Different types of 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 as groups of objects and data structures. This is useful for tasks related to performance, reliability and security [2]. Runtime views are increasingly important in object-oriented code.

Recent work has demonstrated that the extraction of *sound* runtime structure from object-oriented code is possible [1]. What makes the diagram *sound* is that each runtime object has exactly one representative in the object graph, and all the possible runtime points-to relations between those objects are represented by edges. Because many decompositions are acceptable, and because the extraction's default decomposition might not exactly match the developer's mental model, tool support is needed to allow developers to refine the initial extraction.

Our work is the development of the front end of OOGIE (Ownership Object Graph Interactive Editor), which is a tool that allows developers to interactively refine an ownership object graph.

FlexiTools@SPLASH October 18, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM . . . $10.00
Reprinted from FlexiTools@SPLASH, , October 18, 2010, Reno/Tahoe, Nevada, USA., pp. 1–4.

2. Background

A system's runtime structure often has many objects, and the resulting object graph is often large and complex. To mitigate this problem, hierarchy is often used [6]. Hierarchy allows collapsing many nodes into one, and allows collapsing or expanding selected elements to support both high-level and detailed understanding [13].

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 designated as 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 an Ownership Object Graph (OOG) [1], by allowing objects to contain other objects.

In an OOG, an object can 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.

Because architectural hierarchy is not directly expressible in a general purpose programming language, we require the use of annotations. Developers must pick a top-level object as a starting point, then use local, modular ownership annotations in the code. Then the analysis to extract a sound OOG statically begins, following the SCHOLIA approach [1].

An important issue is that the developer may need to visualize an OOG to determine the best annotations to add. But without annotations, an OOG cannot be extracted. So, the common scenario is to add an initial, plausible set of annotations, extract an OOG, then refine both the annotations and the OOG, until the OOG matches the developer's mental model. This process of refining the extracted OOG is somewhat awkward. When the extracted OOG does not match the conceptual model, the developer 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 re-extract the OOG. This issue makes using SCHOLIA tedious and time-consuming.

We propose to address this issue by allowing the user to more directly and interactively manipulate an extracted OOG. Although fully automated approaches can produce useful results at a minimal cost, it is likely that the reverse-engineered abstractions will not match the developer's intent [14]. So we use tool support to allow the developer to interactively refine the extracted OOG to bring it closer to his design intent—without, of course, making the diagram unsound in the process.

A previous study revealed that developers do think in terms of objects and relations. The study also suggested that an extracted object diagram might not initially match a developer's mental model, and what kind of tool support could help the developer iteratively refine an initial object graph to better match their mental model of the runtime object-structure [2]. This study led us to the requirements for iterative refinement.

3. Requirements for Iterative Refinement

From our previous research [2], we developed the following requirements for iterative refinement:

- **Manipulate ownership domains:** The developer can divide the architecture into tiers of components. The developer can create new domains (Fig. 1), and merge existing domains.

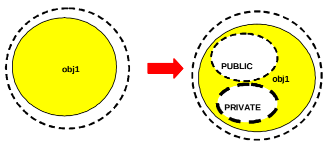

Figure 1. The developer creates the domains `PRIVATE` and `PUBLIC` inside object `obj1`.

- **Manipulate the object hierarchy:** The developer can change the way that objects are grouped into components by moving them from one domain to another, including domains at different levels of the hierarchy (Fig. 2).
- **Abstract objects by type:** By default, an object graph has a component for every type of object at every level in which it is created. To reduce the number of nodes further, the tool supports operations that merge components into one and split a into several components, each of which includes different types of objects. Typically, the types of these objects share a common super-type.
- **Summarize objects as connectors:** Connectors can be simple references between objects, or they can be implemented by one or more objects in the system. When the developer elides an object, the tool may generate a summary edge to account for the transitive communication through the elided object (Fig. 3), effectively pushing that object into the summary edge.

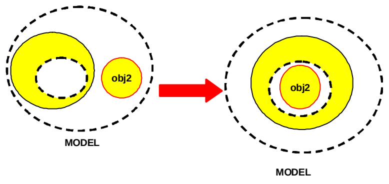

Figure 2. The developer moves the object `obj2` from a top-level domain to a lower-level domain nested inside an object.

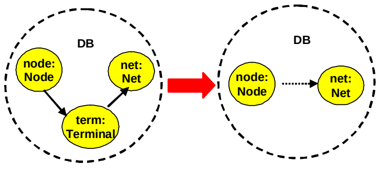

Figure 3. The developer elides the `term` object, which leads to a summary edge between the `node` and `net` objects.

4. Tool Implementation

We implemented OOGIE 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 boxes to indicate containment. Both sides express the runtime structure as a hierarchical graph.

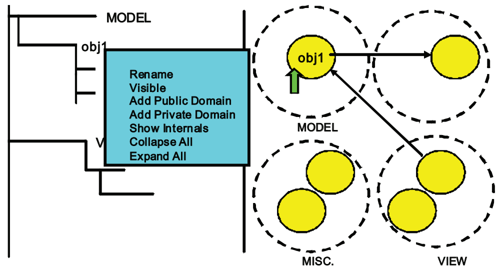

Figure 4. Prototype for OOGIE.

4.1 Example

We illustrate our approach on MicroDraw which is representative of JHotDraw, an open source framework. MicroDraw follows a Model-View-Controller architecture [5]. `MODEL`, `VIEW`, and `CONTROLLER` are the top-level tiers.

Flat object graph. Many tools extract flat object graphs [8], which are often overly complex for developers to navigate and use. Instead, our tool extracts an automated analysis that can infer strict encapsulation, producing a semi-hierarchical object graph. In this case, an automated analysis would indicate that `fSelectionListener` is strictly encapsulated in the `drawingView` object (Fig. 5).

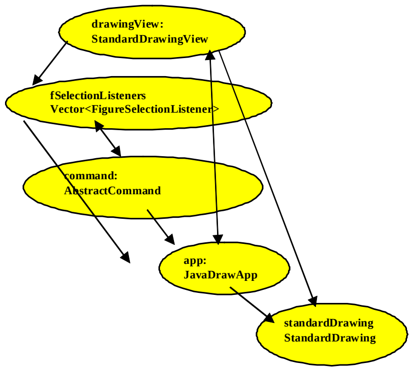

Figure 5. Flat object graph of MicroDraw.

Initial object graph. An automated extraction algorithm will extract a mostly flat object graph, where all the objects are in one top-level tier, since the architectural intent does not exist in the code. The initial extracted model might have all of the objects initially in a top-level tier (Fig. 6).

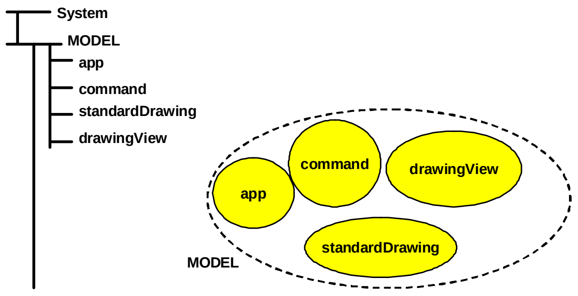

Figure 6. Initial extracted object graph of MicroDraw.

Creating domains. In this example, the developer decides that the architecture should reflect the Model-View-Controller design pattern. To convey the Model-View-Controller design pattern in the object graph, she renames the top-level tier to `MODEL`. She also adds two other 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(Fig. 7).

Controlling level of detail. Then the developer decides to examine the `drawingView` object in more detail and exposes its sub-structure (Fig. 8). The sub-structure highlights that the `drawingView` listens to notifications from other objects

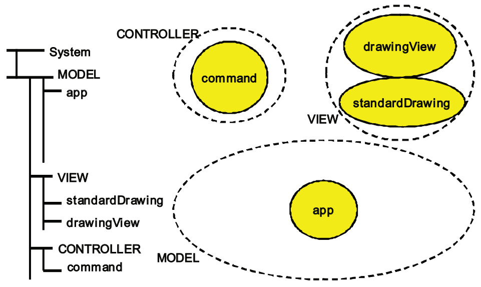

Figure 7. Refined object graph with objects moved into from `MODEL` into top-level domains `VIEW` and `CONTROLLER`.

such as `command`. Such information would be valuable during code modification tasks.

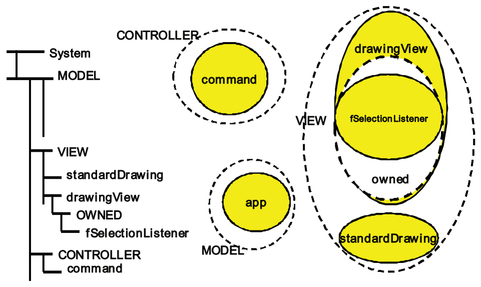

Figure 8. `drawingView`'s sub-architecture is exposed.

5. Evaluation

We did some preliminary evaluations in order to understand the weaknesses in the tool, and how to design more thorough evaluations. We first did a cognitive dimensions analysis. We followed Scaffidi et al. [10], tailoring the framework for software visualization. Through the cognitive dimensions analysis, we found some important limitations that must be addressed. We then conducted a pilot study with one subject who was already familiar with the testing code. From the pilot study, we found problems with the usability of the tool which we plan on addressing. In the future, we plan on conducting a user study then a field study to test whether outside developers will be able to use our interactive editor to refine an extracted architecture to better fit their mental model.

6. Related Work

Runtime Structure. Many dynamic analyses focus on visualizing the object structures of a running system [12]. 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 to help developers understand a program.

Ownership Structure. More closely related are dynamic analyses that infer the ownership structure of a running program based on its heap structure [6]. Dynamic analyses have the advantages of being more scalable and more precise than their static counterparts. However, previous such analyses assume a strict owner-as-dominator model where a higher-level object cannot collapse underneath it not many low-level objects, so they end up cluttering the top-level diagram.

Hierarchical Views. There are examples of tools like OOGIE, that produce hierarchical views of the code architecture that can expose or collapse sub-architecture such as the RIGI visualization system [9] and its follow-up SHRIMP VIEWS [15]. But they do not allow for the visualization to be modified, just explored from different angles. This means that the developer can only view the code from the default decomposition, which may not reflect the developer's mental model.

Iterative Refinement. DA-TU [7] is another software system that allows developers to interactively refine the representation of the data. It does so through clustering and navigation. It allows people to select nodes to group together using a force layout. But the graph is not hierarchical, and is not scalable.

Another class of tools that often include iterative design features are UML tools such as QuickUML [3], 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 [2], and they must be operations that preserve diagram soundness.

7. Limitations

The current implementation of OOGIE is still incomplete.

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 is knowledgeable about 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.

Maintain diagram soundness. The current implementation makes no guarantees of preserving the diagram's soundness for all of the operations in Section 3. 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. In 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 sound, then the updated diagram will be sound.

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.

References

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

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

[3] C. Alphonce and P. Ventura. QuickUML: a tool to support iterative design and code development. In *OOPSLA Companion*, 2003.

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

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

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

[7] M. Huang and P. Eades. A fully animated interactive system for clustering and navigating huge graphs. In *Graph Drawing*, pages 374–383. Springer, 1998.

[8] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *TSE*, 27(2), 2001.

[9] H. Müller and K. Klashinsky. Rigi – a System for Programming-In-The-Large. In *ICSE*, 1988.

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

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

[12] T. Souder, S. Mancoridis, and M. Salah. Form: a Framework for Creating Views of Program Executions. In *ICSM*, 2001.

[13] M.-A. Storey, C. Best, and J. Michaud. SHriMP Views: An Interactive Environment for Exploring Java Programs. In *IWPC*, 2001.

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

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