Tool Support for the Compile-Time Execution Structure of Object-Oriented Programs with Ownership Annotations

Marwan Abi-Antoun
School of Computer Science
Carnegie Mellon University
marwan.abi-antoun@cs.cmu.edu

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
jonathan.aldrich@cs.cmu.edu

ABSTRACT

Ownership domain annotations enable obtaining at compile-time the system's execution structure from the annotated program. The execution structure is sound, hierarchical (and thus more scalable) and conveys more design intent than flat object graphs obtained by existing static analyses that do not rely on annotations.

Categories and Subject Descriptors: D.2.2 [Design Tools and Techniques]: Object-oriented design methods

General Terms: Design, Documentation

Keywords: runtime structure, dynamic structure

1. INTRODUCTION

When modifying a complex program, a developer often needs to understand both the code structure (static hierarchies of classes) and the execution structure (dynamic networks of communicating objects). Class diagrams extracted from source using existing tools are often sufficient to understand the code structure. However, existing static or dynamic analyses produce raw graphs of objects and relations between them that do not convey design intent or readily scale to large programs.

Ownership domain annotations enable a compile-time sound visualization of the execution structure. The visualization is both hierarchical (and thus more scalable) and conveys more design intent than flat object graphs obtained by existing static analyses that do not rely on annotations.

2. OWNERSHIP DOMAINS

Ownership domains [3] control aliasing and ownership through source code annotations. We re-implemented the ownership domains type system using Java 1.5 annotations (as opposed to language extensions) and the Eclipse open source development environment that has become popular with researchers and practitioners [2]. Using annotations imposes a few restrictions but generally improves the adoptability of the technique through improved tool support (refactoring, syntax highlighting, etc.) and the ability to incrementally and partially annotate large code bases.

We implemented various Eclipse plug-ins to automatically add reasonable defaults, incrementally and manually annotate code, typecheck the annotations, and partially annotate the Java Standard Library using external files [2].

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.
ACM 978-1-59593-882-4/07/0011.

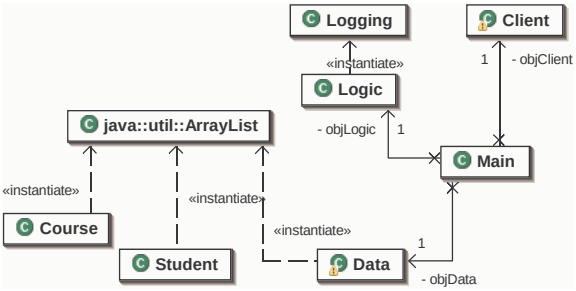

Figure 1: Class Diagram for CourseSys, obtained from the code using Eclipse UML [4].

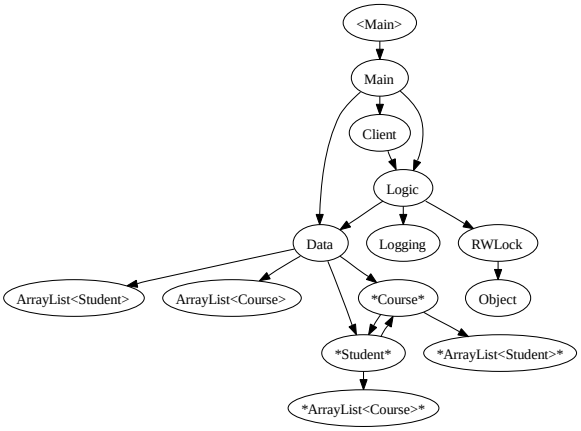

Figure 2: The raw object graph for CourseSys obtained from the code using Pangaea [5].

3. EXECUTION STRUCTURE

We give the intuition behind the visualization of the execution structure by example. CourSys is a simple course registration system to keep track of courses, students, and register students for courses, implemented following a three-tiered architectural style.

A class diagram (Figure 1) might mislead the reader into thinking that the same `ArrayList` object is shared between `Data`, `Course`, and `Student` objects, when at runtime, there are distinct `ArrayList` instances: (a) the list of all available `Course`s; (b) the list of `Student`s registered in a `Course`; (c) the list of `Course`s for which a `Student` has registered; and (d) the list of `Course`s a `Student` has previously completed.

Raw Object Graph. Obtaining at compile time a finite

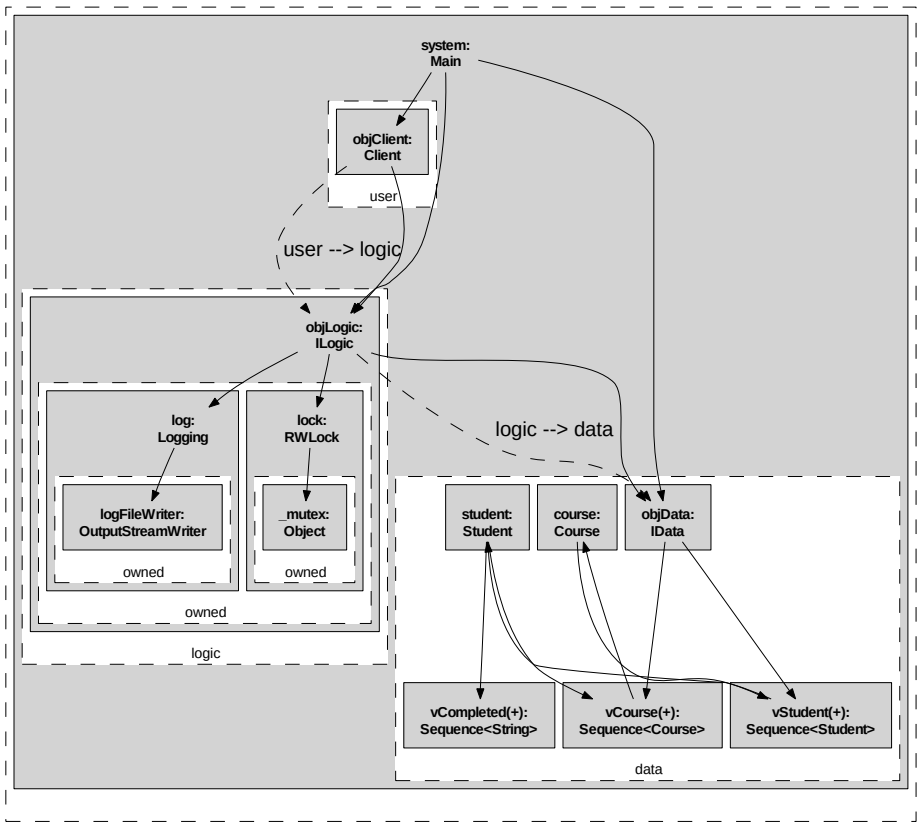

Figure 3: The Ownership Object Graph for CourSys. The edges correspond to field references.

and conservative abstraction of all possible runtime object graphs is more challenging because of aliasing, precision and scalability issues. Static analyses [5] that approximate the runtime object graph often produce non-hierarchical graphs that do not provide design intent or readily scale to large programs (See Figure 2).

Ownership Object Graph. Ownership domain annotations enable a novel compile-time visualization of the execution structure [1]. First, ownership domains provide a coarse-grained ownership structure of an application with a granularity larger than an object or a class. Second, ownership organizes a flat object graph into an ownership tree, and a strict tree hierarchy is needed to achieve scalability and attain both high-level understanding and detail. Third, different ownership domains and different places in the hierarchy provide precision about inter-domain aliasing and conservatively describe all aliasing that could take place at runtime. The type system guarantees that two objects in two different domains cannot be aliased. Thus, the analysis can distinguish between instances that would be merged in a class diagram. Fourth, ownership domain names are specified by a developer and therefore can convey abstract design intent more than arbitrary aliasing information obtained using a static analysis that does not rely on annotations.

Ownership domains are visualized as in Figure 3. A dashed border white-filled rectangle represents an actual ownership domain. A solid border grey-filled rectangle with a bold label represents an object. A dashed edge represents a domain link. A solid edge represents a reference relation between two objects. An object labeled "obj : T" indicates an object of type T as in UML object diagrams.

Figure 3 shows explicitly object `objClient` in a `user` tier, object `objLogic` in a `logic` tier, object `objData` in a `data` tier, and objects of type `Student`, `Course` and lists thereof also in the `data` tier. Furthermore, it shows that objects `log` and `lock` are *owned* by `objLogic`.

Ownership hierarchy is different from the hierarchy found in class diagrams that use packages to organize related classes. Placing the `Logic` and the `Logging` classes in a `coursys.logic` package indicates that they belong to the same layer in the static source code organization. However, when the execution structure indicates that the `log` object is *owned* by the `objLogic` object, `log` is hidden in `objLogic`'s representation and inaccessible to other objects in the system — even if both are in the same `logic` architectural tier at runtime.

The details of the construction of the Ownership Object Graph are discussed elsewhere [1].

4. REFERENCES

[1] M. Abi-Antoun and J. Aldrich. Compile-Time Views of Execution Structure Based on Ownership. In *IWACO*, 2007.

[2] M. Abi-Antoun and J. Aldrich. Ownership Domains in the Real World. In *IWACO*, 2007.

[3] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, pages 1–25, 2004.

[4] Omondo. EclipseUML. `http://www.omondo.com/`, 2006.

[5] A. Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, FU Berlin, 2002.