A Field Study in Static Extraction of Runtime Architectures[1]

Marwan Abi-Antoun **Jonathan Aldrich**

June 2008
CMU-ISR-08-133

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We recently developed a static analysis to extract runtime architectures from object-oriented programs written in existing languages. The approach relies on adding ownership domain annotations to the code, and statically extracts a hierarchical runtime architecture from an annotated program.
We present promising results from a week-long on-site field study to evaluate the method and the tools on a 30-KLOC module of a 250-KLOC commercial system. In a few days, we were able to add the annotations to the module and extract a top-level architecture for review by a developer.

[1]A shorter version is to appear as: Abi-Antoun, M. and Aldrich, J. A Field Study in Static Extraction of Runtime Architectures. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2008.

This work was supported in part by NSF CAREER award CCF-0546550, DARPA contract HR00110710019, the Department of Defense, and the Software Industry Center at Carnegie Mellon University and its sponsors, especially the Alfred P. Sloan Foundation.

Keywords: runtime architecture, architecture recovery, ownership types, field study

Contents

List of Figures

"An object-oriented program's runtime structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program's runtime structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa." (Gamma et al., 1994 [17]).

1 Introduction

Software architects describe a system using different architectural views. A *code architecture* or *module view* shows code entities in terms of classes, packages, layers and modules. A *runtime architecture* or *runtime view* of a system models runtime entities and their potential interactions [13].

Many tools automatically extract module views from source code [24], but the support for runtime views is less mature [22, 37]. Intuitively, many have preferred dynamic analyses to extract runtime architectures. But, a dynamic analysis extracts partial descriptions that cover interactions between objects from a few program runs. To be most useful, an architecture must capture a complete description of a system's runtime structure. This requires a static analysis that is *sound*, i.e., one that reveals all entities and relations that could possibly exist at runtime.

Previous static analyses extract low-level non-hierarchical object graphs that do not provide architectural abstraction [31, 26, 19, 38]. Other approaches use radical language extensions [7, 34] or mandate architectural middleware or frameworks [28]. To handle existing systems, an approach must support existing languages, common design idioms, and existing frameworks and libraries. But adding annotations to clarify the design intent might be acceptable.

We have been applying ownership domain annotations for architectural extraction [1, 3]. Ownership types were originally proposed to control aliasing [12, 6, 14], but also enable the static extraction of runtime architectures, because they track instances instead of types. In our architectural recovery method, a developer adds annotations to clarify the architectural intent related to object encapsulation, logical containment and architectural tiers.

The annotations specify and enforce the sharing of data between objects, a key challenge in extracting a runtime architecture. This state sharing is often not explicit in object-oriented programs, rather, it is implicit in the structure of references created at runtime.

Using annotations to recover design from code is not new [26]. But previous systems did not support hierarchy, and thus did not scale to large systems at multiple levels of abstraction, nor did they support critical language constructs like inheritance.

Our approach does have the overhead of adding annotations to a program, which is currently done mostly manually. Precise and scalable ownership inference is a separate problem and an active topic of ongoing research [29].

In this paper, we present promising results from an on-site field study to demonstrate the approach's feasibility on real code and users. The paper is organized as follows. Section 2 gives an overview of the approach. Section 3 discusses the methodology we followed. Section 4 discusses the results. Finally, we survey related work in Section 5 and conclude.

2 Overview

In this section, we give the intuition behind the static extraction of runtime architectures by example.

2.1 Mapping Source to High-Level Models

Architectural extraction maps source code entities to entities in a high-level model. Consider a Document-View system where a `BarChart` and a `PieChart` render a `Model`.

We used AgileJ to extract a module view from the program [5]. Fig. 1(a) shows classes, inheritance and association relations. For instance, classes `BarChart` and `Model` implement a `Listener` interface. The view also shows associations from `Model` and `BaseChart` to `Listener`. But it does not explain the instance structure of the application. For instance, it is not clear if a `Model` object and a `BaseChart` object share the same `Listener` object at runtime. It is also unclear whether instances of `PieChart` and `BarChart`, which inherit from `BaseChart`, share one `Listener` object.

In general, an object-oriented program's runtime structure often bears little resemblance to its code structure. One code element can appear as multiple elements in a runtime view. Alternatively, one element in a runtime view can correspond to multiple code elements.

At runtime, `BarChart` and `Model` objects each contain a `List` of `Listener` objects. Moreover, the `Listener` object inside a list `List<Listener>` maps to multiple design elements, based on the context (Fig. 1(b)). For example, inside a `Model` object, a list element of type `Listener` refers to a `BaseChart` object or one of its subclasses. But inside a `BaseChart` object, a list element of type `Listener` refers to a `Model` object. In Fig. 1(b), dashed white-filled boxes represent runtime tiers. Solid-filled boxes represent objects. Solid edges represent field references between objects.

An analysis for object-oriented code must handle inheritance. Typically, `PieChart` and `BarChart` extend `BaseChart`, and `BaseChart` declares a `listeners` field. In addition, there is possible aliasing. In Fig. 1(b), if the `listeners` fields of `BarChart` and `Model` referred to the same object at runtime, the architecture would be deceptive; a correct architecture must show them as one object.

Conceptually, each view has a separate `listeners` object, and the `listeners` object of a `pieChart` is distinct from that of a `barChart` (Fig. 1(c)). So we may not want `listeners` in the top-level tiers. In a runtime view, we model these lists as *part of* a `barChart` or `model`. Fig. 1(c) uses the nesting of boxes to indicate hierarchical containment. The thick dashed borders indicate that these instances are *owned* or strictly *encapsulated* by their outer objects.

Ideally, an architecture "can be read in 30 seconds, in 3 minutes, and in 30 minutes" [25]. Hierarchy enables eliding information at any level to show overviews of the system architecture at various levels of abstraction [39]. The (+) symbol on an object's label indicates that its sub-structure is elided. Dotted edges summarize any solid edges by lifting them from elided objects to visible ones (Fig. 1(d)).

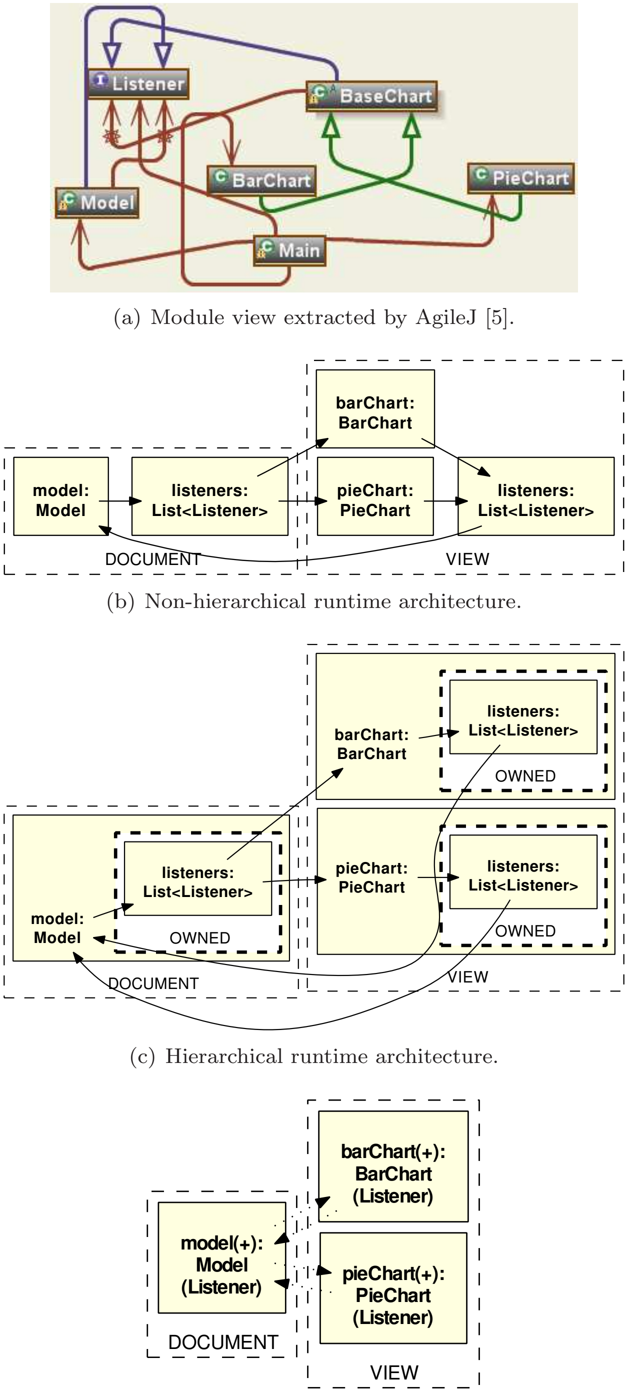
2.2 Ownership Domains

An ownership domain is a *conceptual group of objects* with an explicit name and explicit policies that govern how it can reference objects in other domains. Each object belongs to a single ownership domain that does not change at runtime.

Fig. 2 shows the annotations that a developer might add to an implementation of the above example. The actual system uses existing language support for annotations [2], but here, we use a simplified syntax that extends the language. A developer indicates what domain an object is part of by annotating each reference to that object in the program (Line 19). Domain names are arbitrary, and ideally, convey design intent. We often use capital letters to distinguish domains from other program identifiers.

Ownership domains may be declared at the top level of the application (Line 17) or within an object (Line 4). Each object can declare one or more *public* or *private* domains to hold its internal objects, thus supporting hierarchy. A private domain provides strict encapsulation. But a public domain provides logical containment, and makes its objects accessible to any object that can access the outer object [6]. An object `model` can access objects in a domain `VIEW` by declaring on the class of `model` a formal *domain parameter*, e.g., `V` (Line 11), and *binding* that formal domain parameter to the actual domain `VIEW` (Line 19).

The type system defines a few special annotations [6]: `unique` indicates an object to which there is only one reference, such as newly created objects, or objects that are passed linearly from one domain to another. One ownership domain can temporarily lend an object to another and ensure that the second domain does not create persistent references to the object by marking it `lent`. Objects that are `shared` may be aliased globally but may not alias non-`shared` references.



(a) Module view extracted by AgileJ [5].

(b) Non-hierarchical runtime architecture.

(c) Hierarchical runtime architecture.

(d) Overview architecture.

Figure 1: A Document-View architecture.

2.3 Static Analysis

We developed a static analysis to extract from an annotated program a hierarchical runtime object graph, one that provides architectural abstraction by ownership hierarchy and by types [1, 3]. Thus, only architecturally

```
interface Listener { }
class BaseChart<M> // Declare domain parameter M
     implements Listener {
  domain OWNED; // Declare protected domain OWNED
  // Declare reference to List object in OWNED
  // Inner annotation M is for the list elements
  OWNED List<M Listener> listeners;
}
class BarChart<M> extends BaseChart<M> { }
class PieChart<M> extends BaseChart<M> { }
class Model<V> implements Listener {
   domain OWNED;
   // Inner annotation V is for the list elements
   OWNED List<V Listener> listeners;
}
class Main {
  domain DOCUMENT, VIEW; // Top-level domains
  // Bind domain parameter V to actual domain VIEW
  DOCUMENT Model<VIEW> model;
  VIEW BarChart<DOCUMENT> barChart;
  VIEW PieChart<DOCUMENT> pieChart;
}
```

(a) Code snippets with ownership domain annotations.

`DOM Type obj`: declare object `obj` of type `Type` in domain `DOM`;
[public] domain `DOM`: declare private [or public] domain `DOM`;
`class C<D_PARAM>`: declare domain parameter `D_PARAM` on `C`;
`C<DOM> cObj`: bind actual domain `DOM` to domain parameter;

(b) Simplified syntax for ownership domain annotations.

Figure 2: Two-tiered system with annotations.

relevant objects appear in the top-level domains. Each of those objects has nested domains and objects representing sub-architectures, and so on, until low-level less architecturally relevant objects are reached. The architecture collapses nodes based on the ownership structure, not according to where objects were declared in the program, some naming convention, or a graph clustering algorithm.

3 Field Study

A field study is a generally accepted research method to evaluate how well a software tool or method works with real code and users [23]. In our field study, we extracted the architecture of a portion of a large Java system. We selected a target portion of the system, communicated with the original developers of the code to understand their design intent, added annotations to the code, typechecked the annotations, ran our static analysis to extract an architecture, and showed snapshots to the developers.

Some of the research questions we wanted this case study to help answer include:

How to annotate a real object-oriented system?

How much effort will it take?

Can one add annotations for the top-level architecture, then extend those annotations down to the rest of the system?

How can we improve the usability of the tools?

3.1 Setup and Methodology

We refer to the person who conducted the field study, this paper's first author, as the *experimenter*. The *developer* is the person who was familiar with the code being analyzed.

Pilot Constraints. Our tools consist of plugins in the Eclipse Java development environment. So we required a module that is Java-based. Since we were adding the annotations manually, we required a module under 50 KLOC. In earlier evaluations, we sometimes refactored the subject systems while adding the annotations [3]. Here, we wanted to extract the *as-is* architecture. We also did not want to explain the annotations or the static analysis to the developers, nor did we expect to involve them with the tools.

The Plan. Most architecture recovery starts by gathering or eliciting documentation from developers who are familiar with the code. Ideally, a developer would document the as-designed runtime architecture, but realistically, we knew that we may have to settle for a class diagram.

Data Collection. The experimenter measured the effort by keeping track of the different activities in a time log, and measured the end-to-end time, minus interruptions. He also kept a log of annotation cases that revealed facts about the code such as representation exposure or tight coupling.

The experimenter kept track of the iterations, and what he changed between iterations, such as changing the settings or inputs to the tools. He saved intermediate snapshots of the extracted architecture. He also wrote detailed notes to simulate the thinkaloud protocol (he could not actually speak as he was sitting with others in an open-floor space). After the study, we used the Eclipse history data for each file to analyze how the annotations evolved.

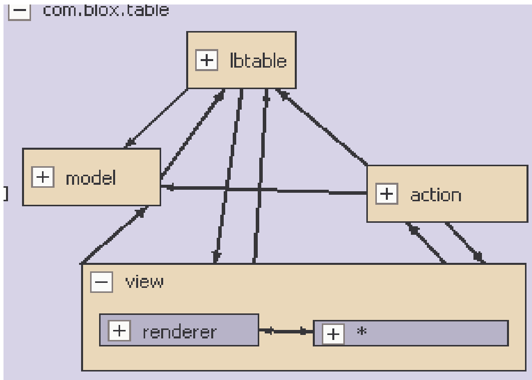
Subject Selection. The experimenter ran the jMetra [18] code measurement tool on the Java code base, and identified a module of around 30KLOC, excluding unit test code, which we refer to as LbGrid. LbGrid is a multi-dimensional feature-rich grid control that consists of around 300 classes (jMetra includes only static inner classes in the class count, and LbGrid uses inner classes extensively).

In previous evaluations, we used code bases developed prior to Java 1.5 and refactored them to use generics to improve the precision of the analysis [3]. In this case, the code already used generic types. As a bonus, a developer who was familiar with that module would be available.

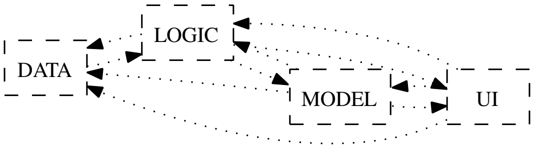
Static Analysis. At no time during the study did the experimenter run the system. That would have required an elaborate setup of a complex client-server system, and training on how to use the system to get good coverage. So using static analysis simplified the setup considerably.

Plan vs. Actual. The study did not go exactly as planned. The developer familiar with LbGrid was not available on the first and the last days of the study. Generally, the experimenter had limited access to the developer. We estimate the developer spent around 4 hours, including the initial meeting, designing and discussing the code architecture, answering occasional questions, examining snapshots and responding to our emails.

Target Architecture. The experimenter met with the developer for two hours, and gave him an overview of the architectural views we were extracting. The developer said he used and liked tools that extracted class diagrams from code. The experimenter asked the developer to draw the LbGrid as-designed runtime architecture. The experimenter wanted to use the as-designed architecture as a guide to add the annotations, by following the same top-level architectural tiers and the same architectural decomposition. Unsurprisingly, the developer drew an abstracted class diagram showing the core types. The developer's diagram and the extracted architecture are in the appendix.



(a) High-level module view, obtained using Lattix LDM [27]. A box represents a Java package.



(b) A 30-second high-level runtime architecture. The dotted edges summarize inter-tier references.

Figure 3: High-level module and runtime views.

3.2 Extraction Process

We now discuss the process the experimenter followed to annotate LbGrid and extract its runtime architecture.

Isolating the Module. The experimenter set up stop-analysis configuration files to analyze only the compilation units from a list of selected packages and exclude others.

Tool Support. The experimenter used a tool to generate initial default ownership domain annotations for the selected Java files [2]. He then completed the annotations mostly manually, as we discuss in the next section. At times, he used a utility to globally find and replace annotations across several files. He used mainly two tools. First, a typechecker validates the annotations and displays warnings in the Eclipse problem window. A second plugin extracts a hierarchical runtime architecture.

The extraction tool also allows the developer to control the visualization of the extracted architecture, select the projection depth, and elide the substructure of selected objects.

Adding Annotations. The best annotations produce a view comparable to what an architect might draw for an architecture. Just as there are multiple architectural views of a system, there is no single right way to annotate a program.

Because the developer did not draw an as-designed architecture, the experimenter studied the developer's diagram and suggested organizing the runtime architecture according to the following top-level domains: `UI`, `MODEL`, `LOGIC` and `DATA`. The developer confirmed that the proposed architectural tiers seemed reasonable. Another senior developer who is familiar with other parts of the system also agreed with this high-level organization of the LbGrid architecture.

Then, the experimenter started mapping objects to domains. As a first approximation, he mapped types to domains. Of course, not all the instances of a type, such as `List`, always appear in the same domain. Also, LbGrid has several classes that have single instances, e.g., `Workspace`. In many cases, he used the package declaration as a guide. For instance, an instance of a class from a `data` package often belonged to the `DATA` tier. The trickier cases were instances of classes from nondescript utility packages that gave no indication about which runtime tier they belonged to. The experimenter organized the core types as follows:

`UI`: instances of `LbTable`, etc.;
`MODEL`: instances of `LbTableModel`, etc.;
`LOGIC`: has instances of `PivotManager`, etc.;
`DATA`: has instances of `Workspace`, `Predicate`, etc.

Once the experimenter figured out the top-level domains, he propagated them as domain parameters, as needed, using the mnemonic domain parameter names: `U` for `UI`, `M` for `MODEL`, `L` for `LOGIC`, and `D` for `DATA`.

Typechecking the Annotations. The experimenter followed an iterative process. After each round of annotations, he ran the typechecker, examined the warnings, and addressed them from the most to the least important ones.

Prioritizing the Warnings. To some extent, the annotations are partially and incrementally specifiable. For the duration of the field study, the experimenter turned off domain link checks, as he was not planning on adding those annotations. Except for the implicit defaults or those added by the tool, every reference type must be annotated. Enabling all the annotation checks at once would generate tens of thousands of warnings in the Eclipse problem window, and bring Eclipse to a standstill. (the experimenter was running the tools on a modest Intel Pentium 4 (2 GHz) with 1.5 GB of memory). Moreover, one missing or incorrect annotation in the code could potentially produce several warnings. So the experimenter gradually enabled various annotation checks, and addressed annotation warnings from most to least important.

First, check undeclared domains or domain parameters. On lines 11-12 (Fig. 2), the domain parameter `V` and domain `OWNED` must be declared before an object can be annotated with it. Then, check unbound domain parameters at field and variable declarations. E.g., lines 20-21 bind the `M` domain parameter to `DOCUMENT`. Then, check domain parameter inheritance. On line 9, the domain parameter `M` on `PieChart` is bound to the domain parameter on `BaseChart`. Next, look for disallowed assignments: a field reference annotated with `DOCUMENT` cannot be assigned to another one annotated with `VIEW`. Finally, check that a `lent` variable, which denotes a temporary alias, is not stored in a field.

Once the experimenter established the top-level domain and domain parameter structure, he set the domain annotations for array elements and elements stored inside containers.

Extracting Architectures. The extraction tool works in the presence of annotation warnings, but warns that the extracted architecture may not reflect all objects and relations. In the early iterations, we placed most objects in one of the domain parameters, `U`, `M`, etc. Since each domain parameter was transitively bound to a top-level domain, e.g., `U` to `UI`, `M` to `MODEL`, these early architectures showed many objects in the top-level domains. But these early diagrams helped the experimenter refine the annotations and move a few objects between the top-level domains. In later iterations, he defined several private and public domains, and moved secondary several objects from a top-level domain to a private or public domain of a primary object, to reduce the clutter in the top-level domains,

Refining the Annotations. Our approach achieves the desired number of objects in each top-level domain, primarily through *abstraction by ownership hierarchy*. The strategies include:

Using the strict encapsulation of private domains;
Using logical containment with public domains;
Passing low-level objects linearly using the `unique` annotation.

In addition, one can reduce clutter further and use *abstraction by types*, which merges fewer or more objects, in a given domain, based on the architectural relevance of their declared types [3].

A rule of thumb in architectural documentation is to have 5 to 7 components per tier [25]. So the experimenter followed one of the above strategies to minimize the number of top-level objects and the number of annotation warnings.

Strict Encapsulation. The experimenter identified encapsulated objects that are used only inside an object and not returned by any accessor and placed them in private domains. In some cases, specifying a strict encapsulation to avoid the representation exposure required a small change to the code, namely, to return a copy of an internal list instead of an alias.

Logical Containment. The developer's feedback helped the experimenter define several public domains with architecturally meaningful objects. Using logical containment often involved only localized changes to the annotations. For instance, the public domain `RENDERERS` on `LbTable` holds objects of type `TextCellRenderer` and `ColorCellRenderer`. The `EDITORS` domain holds objects of type `TextCellEditor` and `ColorCellEditor`. In contrast, the module view shows all these types in one `renderer` package (Fig. 3(a)). Careful developers seem to chose their package structure to designate some architectural intent. But in general, one cannot represent the dynamic structure of a runtime view in the static source code organization.

Other architecturally significant public domains include:

- `LbTableModel` has a `HEADERS` domain to hold `HeaderGridPath` and `HeaderGroup` objects, among others;
- `TableActionManager` has an `ACTIONS` public domain for `LockCellsAction` and `FillCellAction` objects, etc.

Linear Objects. He used the `unique` annotation when a method performed a query, allocated a container to store the query result objects, and then another object iterated the container elements then discarded the container without storing a reference to it.

Questions to Developer. The experimenter had limited interaction with the developer. Occasionally, he asked the developer the following questions.

- Does an instance of type `T` belong to tier `D`?
- Is object `X` in tier `D` conceptually part of object `Y`, so `X` can be pushed down underneath `Y`?

The first question helped the experimenter identify components that appear in the wrong conceptual tier. The second question guided the abstraction of the runtime object graph by ownership hierarchy.

The experimenter also asked the developer to identify the root object from which to derive the runtime architecture. The developer pointed him to a unit test class.

4 Results

4.1 Quantitative Data

Of the time spent on-site, the experimenter spent about 30 hours adding the annotations, typechecking them, and examining snapshots of the extracted architecture. After the experimenter returned from the field trip, the developer emailed him some comments on a snapshot of the architecture. The experimenter spent another 5 hours adjusting the annotations to incorporate the developer's suggestions and address high-priority annotation warnings. At that point, the top-level architecture still did not fit on one letter-size readable page, such as the developer's code architecture (see appendix). And there were still around 4,000 annotation warnings, most of them minor.

4.2 Qualitative Data

We observed the following facts.

The developer understood assigning objects to runtime tiers. The developer seemed comfortable thinking with a granularity coarser than an object or a class. He drew layers in his diagram that roughly correspond to packages, similarly to a high-level module view (Fig. 3(a)). He understood mapping runtime objects to tiers, and even suggested moving some objects from one tier to another.

> The following components should move to different containers: "[Move] `AxisLayoutInfo` [from `MODEL`] to `LOGIC`."

The developer grasped the intuition behind abstraction by ownership hierarchy, namely, that the goal is to only show primary objects in the top-level tiers:

> "The following are too low-level to be at the outermost tier: `CellPosition`, ..."

The key abstraction mechanism in our approach is to push secondary objects underneath primary objects. The developer understood that. For example, he recommended objects of type `TableHeaderGroup` and `TableHeaderGridPath` be pushed underneath the `LbTableModel` object in the `MODEL` tier. When provided with printouts of the extracted architecture, he expressed interest in viewing an object's sub-structure. At the time of the study, we did not have a standalone viewer. Since then, we implemented an interactive viewer that allows drilling into an object's substructure, zooming, scrolling and panning.
He noticed when a top-level domain showed many objects:

> "All components in `DATA` are also too low-level to be at the outermost tier, but I can't think of a larger component that you can expand to get to them. Not sure how to represent this."

To address the developer's last comment about the `DATA` tier, it is possible to elide a domain's structure, as in Fig. 3(b). The tool currently shows summary edges between collapsed domains. In future work, we will implement a feature to show edges between an object and a collapsed domain.

The developer understood object merging. By design, our runtime architecture conservatively maps to one object all the objects within a domain that may alias, based on their type information. For instance, objects in the `VIEW` domain referenced by the `Listener` interface, the base class `AbstractChart` or its concrete subclass `BarChart`, are merged into a `BarChart` object, because they may alias (Fig. 1(c)). The ownership domains type system guarantees that two objects in different domains can never alias, however, so the analysis keeps those objects as separate.

Riehle posited that designers often use the following techniques to abstract their code architectures. They merge interface and abstract implementation class — although important for code reuse, such a code factoring is often unimportant from a design standpoint. They also subsume similar classes under representative classes, to avoid the clutter of showing many similar subclasses that vary in minor aspects [33, pp.139–140]. Indeed, the developer seems to have used the above techniques in his own class diagram. For example, he used both "XXX" in the name of a few classes to represent multiple elided subclasses. He also used a multiplicity-like symbol to designate many more subclasses that he did not show on the diagram.

So it is unsurprising that these heuristics seemed also intuitive in a runtime view. However, our approach achieves similar results by merging objects in a domain based on their type, to soundly handle possible aliasing.

The extracted architecture shed some light into dark corners of the system. Upon examining the extracted architecture, the developer identified several classes that were candidates for deletion.

> "... `FormulaEditor` (will be deleted shortly)."

The developer seemed unsure about certain object communication. Developers often have a conceptual model of their architecture that is mostly accurate, but may be a simplification of reality [30, 7]. He drew some connections with question marks. The extracted architecture might help him confirm the presence or absence of communication.

A runtime architecture may help with certain coding tasks, but not with others. The developer was skeptical of the runtime architecture (we recorded his opinion below before we gave him a standalone interactive viewer):

> "To step back a little and look at the diagram itself, so far, I can't see the value of a runtime view. I suspect that this will make more sense if I were to be able to drill down into the components. Or do you think that I should be able to see something in the outermost tier itself?"

We emphasized to the developer that the intent of a runtime architecture is to complement, not replace, the code architecture. Since he mentioned sequence diagrams, we explained to him that a sequence diagram is a kind of runtime view that shows method invocations for a specific use case, and is not a complete architecture. A more closely related diagram would be an object diagram which shows object instances exclusively, which the Gang of Four book uses to explain the standard design patterns [17]. We suggested to the developer that he thinks of our runtime view as an object diagram where each box is an aggregate of objects.

To address the developer's comment, we showed him how hierarchy enables obtaining a high-level runtime architecture. For example, Fig. 3(b) makes explicit several global structural constraints that are implicit in the code, e.g., that objects in `DATA` do not reference objects in `MODEL`.

When reasoning about modifiability, a code architecture may be more helpful than a runtime architecture. The developer may have been focused on such tasks because he drew a detailed class diagram mostly from memory, and referred to Eclipse only occasionally to verify the name of a type. He seemed apologetic about the current design having many subclasses and a parallel inheritance hierarchy. In the current design, `GridTable` extends `BloxTable` extends `LbTable`. A parallel inheritance hierarchy exists between `GridTableManager`, `BloxTableManager` and `LbDefaultTableManager`.

He mentioned that one could refactor away some of those classes and move their functionality into their super-classes (the rationale for the current factoring is that super-classes are oblivious to accessing data from a workspace). He even asked the experimenter if he could think of a design that did not require proliferating sub-classes.

Since he was very familiar with the LbGrid code, he did not immediately see the value of a runtime architecture. We posit that because the runtime architecture abstracts away the factoring into interfaces, base classes and subclasses, it may actually be simpler to explain to someone completely unfamiliar with the code, such as a new hire.

A runtime architecture can help explain listeners. A runtime architecture can help answer questions such as: What instances point to what other instances? As a result, a runtime view can help explain what objects get notified during a change notification. In many cases, UML class diagrams or call graphs do not help answer such questions, because they do not show instances. For example, Figs. 1(c),1(d) highlights the reference structure between `pieChart`, `barChart` and `model` better than Fig. 1(a).

LbGrid uses listeners heavily. Several classes have lists of listeners and implement various listener interfaces. Neither the developer's diagram nor an automatically generated class diagram, explain how the listeners work in LbGrid. We posit that this aspect of the architecture would be particularly challenging for a new hire. In future work, we will identify LbGrid bug reports or enhancement requests that require understanding the listener architecture, and for which the extracted runtime architecture would be useful.

Picking the right labels for architectural elements is crucial, to avoid obtaining a model that developers do not recognize [30]. The developer insisted on specific labels for the various tiers, e.g., use `UIMODEL` instead of `MODEL` (we still use `MODEL` here for consistency with prior documentation). He seemed to always look for instances of the core types from his diagram:

> "Where is `GridPanel`? I don't see it here."

Each object in an extracted architecture merges at least one field or variable reference declared in the program. An object might have multiple types, and the analysis picks one of those types as the label. We already provide a feature to search by name or by type for an object in the hierarchy of objects that constitutes an architecture. The tool can also trace each node or edge in an architecture to a set of nodes from the program's abstract syntax tree, down to the corresponding lines of code in the program.

In response to the developer's feedback, we implemented a feature to allow the user to specify a list of labeling types. For example, in Fig. 1(d), the decoration `(Listener)` is added to an object's label, if it merges at least one object of that type, as is the case for `pieChart`, `barChart` and `model`.

The developer expected to see multiplicities on the runtime architecture. Indeed, the developer's diagram has specific multiplicities on several associations. Many UML reverse engineering tools also show multiplicities. The developer suggested that this information would be useful, so we will implement this feature in future work.

The developer expected the tools to render a judgement on the recovered architecture. This was an unexpected request. It might be possible to compute various dynamic coupling metrics [8]. Another avenue would be to check and measure the structural conformance of the as-built architecture against the as-designed one [30], but this requires establishing the target runtime architecture.

The developer seemed to favor an unsound abstracted task-specific view over a sound runtime architecture. A tool that extracts a class diagram automatically would show at least 300 classes for LbGrid, organized by packages. However, the developer's diagram had many fewer types. So the question is whether a runtime architecture should soundly reflect all objects and relations that exist at runtime, or only those that are of current interest to the developer. In our principled approach, the primary means of abstraction is through ownership hierarchy. One changes the annotations to push secondary objects under primary objects, and sometimes changes the code to implement strict instance encapsulation. An unprincipled approach would allow eliding any object or domain in the extracted architecture. In future work, we will consider ways to make a runtime architecture more concretely reflect the types that are of interest to a developer, while maintaining soundness.

4.3 Validity

We identify the following confounding factors.

Experimenter Bias. The experimenter understood ownership domain annotations and designed several tools that were used in the field study. Moreover, he had access to the code for the tools and customized them to the task to minimize data entry by loading settings from a file. However, a typechecker kept him honest, i.e., he could not just insert any annotation or manipulate the extracted architectures. In a few instances, the experimenter backtracked on certain annotations he had just inserted.

Code Unfamiliarity. The experimenter was completely unfamiliar with the code. A developer who is familiar with the code could perhaps add better annotations faster.

Developer Motivation. The field study occurred in a workweek during which the developers were busy meeting a product ship deadline. As a result, they were less motivated to help the experimenter. Moreover, the developer seemed skeptical about the method and the tool.

Domain Familiarity. LbGrid was somewhat similar to the JHotDraw subject system the experimenter studied previously, in that they are both GUI-based applications that used the Java Swing and AWT libraries [3]. The experimenter also had some experience with the application domain, having previously developed a reusable grid control.

External validity. We reported here the results of a single project study. While developing and refining the approach, we evaluated it on several extended examples totaling 38 KLOC [3]. While the sizes may still seem small, the static analysis of runtime architectures is not yet mature compared to module architectures. For instance, the most relevant previous work used one 1.7 KLOC system [26].

5 Related Work

Runtime Architectures. Several researchers have called for a better way of understanding a system's runtime architecture. Kirk et al. emphasize that "understanding the dynamic behavior of a framework is more challenging, particularly given the separation of the static and dynamic perspectives in the object-oriented paradigm" [22]. Shull et al. concur that both "the static and dynamic structures must be understood and then adapted to the specific requirements of the application [...] For a developer unfamiliar with the system to obtain this understanding is a non-trivial task. Little work has been done on minimizing this learning curve" [37]. Our work is taking in step in that direction.

Architectural Recovery. Many architectural recovery approaches use information such as naming conventions and directory structures [32], and obtain abstracted code architectures, not runtime architectures [10]. Many studies use trial and error with graph clustering algorithms [21, 11] or various sources of information extrinsic to the code.

Clustering methods are complementary to this approach and may help with annotating an unfamiliar system. For instance, two strongly connected clusters may suggest creating two top-level domains corresponding to the two clusters. A small cluster that interacts with almost all others may indicate `shared` objects.

In our study, we only added annotations, typechecked them, and occasionally, discussed a snapshot with a developer. The measurable success criteria in our approach are to first minimize the number of objects in the top-level domains, and second, to minimize the remaining annotation warnings.

Several dynamic analyses have been proposed [36, 35, 16]. DiscoTect [35] recovers a non-hierarchical runtime architecture from a running program, one that shows one component for each instance created at runtime. A dynamic analysis produces a partial description for particular inputs and exercised use cases. In our approach, the remaining annotation warnings give some indication of how soundly the extracted architecture represents the runtime structure.

Reflexion Models. Murphy el al. produce a mapping of a source to a high-level model using the Reflexion Models (RM) approach [30, 20]. In RM, the developer assigns component families to classes. There are several important differences with RM. First, the RM source extractors are lightweight and originally designed for procedural code. The object-oriented version of the RM method maps classes to components [20]. Such a mapping is more suitable for a code architecture. More specifically, RM cannot map the same code entity to multiple design elements, depending on context. Second, RM uses non-hierarchical high-level models and maps, whereas our approach produces hierarchical representations. Third, a developer writing the map must ensure that a type and its subtypes are mapped to the same entity in the high-level model. She must also map all the references that may alias to the same high-level entity. In our approach, a type system checks that the annotations are consistent, and that the code is consistent with the annotations, and the architectural extraction analysis handles possible aliasing and inheritance. Producing the mapping file in the RM approach appears more straightforward than adding ownership annotations, but it is not amenable to type inference. More sophisticated source abstraction methods are needed to extract a runtime architecture from object-oriented code soundly in the presence of inheritance and aliasing.

Object Graph Analyses. Several static analyses produce non-hierarchical object graphs without using annotations. Pangaea [38] produces a flat object graph without an alias analysis and is unsound. Womble [19] uses syntactic heuristics and abstraction rules for container classes to obtain an unsound object model, which has multiplicities. Ajax [31] uses an alias analysis to build a refined object model as a conservative static approximation of the heap graph reachable from a given set of root objects. However, Ajax does not use ownership and produces flat object graphs. Its output was manually post-processed to remove "lumps" with more than seven incoming edges [31, p. 248]. Ajax's heavyweight but precise alias analysis does not scale to large programs. In general, flat objects graphs do not provide architectural abstraction or scale, because the number of top-level objects in the architecture increases with the program size.

Annotation-Based Systems. Lam and Rinard [26] proposed a type system and a static analysis (which we refer to here as LR) whereby developer-specified annotations guide the static abstraction of an object model by merging objects based on *tokens*. LR supports a fixed set of statically declared global tokens, and the result of their analysis is showing a graph with which objects appear in which tokens. Using token parameters, the same code element can be mapped to different design elements depending on context.

Domains are declared on a class, but fresh instances of these domains are created for each instance of that class, i.e., `obj1.DOM` and `obj2.DOM` are distinct for fresh `obj1` and `obj2`. LR uses a statically fixed number of tokens, all of which are at the top level, so LR cannot show hierarchy such as a `listeners` object nested within a `Model` object (Fig. 1(c)). LR's only case study was on one 1.7KLOC module. If we were apply LR to LbGrid, LR would show at least 300 objects in the top-level tokens. In contrast, our system applies abstraction by ownership hierarchy and types to show an order of magnitude fewer objects in the top-level domains.

Confined types track classes not instances [9], and have a lower annotation overhead than ownership types. But a *package* in confined types is roughly a package-level static ownership domain and thus coarser than a token.

Language Extensions. Specifying the architecture directly in the code using language-based solutions simplifies the static architectural extraction [7, 28, 34]. The ArchJava research language specifies a component-and-connector architecture directly in code [7], and often requires re-engineering existing implementations [7, 4].

The first author previously re-engineered a 15-KLOC Java program to ArchJava [4]. The re-engineering required code changes, such as making fields be `private` (a `component class` cannot have `public` fields). ArchJava also prohibits returning references to instances of `component class`es, which required more invasive changes such as changing the application's initialization order [4]. We do not believe it would have been feasible to re-engineer the LbGrid module in the same few days that it took to add the ownership domain annotations, even after accounting for possible tool and language familiarity. Thus, there is compelling evidence that an annotation-based approach is likely more adoptable than radical language extensions such as ArchJava or others [7, 34], at least for existing systems.

Diagram Evaluation. Several researchers have evaluated empirically the usefulness of various design diagrams, e.g., [15]. Unfortunately, these evaluations mostly focus on module views such as class diagrams, or partial runtime views such as sequence diagrams. We hope to see more evaluations of runtime architectures. For instance, Gamma et al. used *object diagrams* which show object instances exclusively, to explain many of the standard design patterns [17].

6 Conclusion

Our field study yielded promising results in the use of ownership domain annotations and static analysis to extract a hierarchical runtime architecture for a real system, to complement the code architectures produced by existing tools.

Acknowledgments

Funding was provided by NSF CAREER award CCF-0546550, DARPA contract HR00110710019, and LogicBlox Inc. The authors also thank Molham Aref and the developers from LogicBlox Inc., for hosting the weeklong on-site field study. Brad A. Myers and William Scherlis offered useful advice on conducting a field study. Thomas LaToza, Brad A. Myers and Christopher Scaffidi gave us helpful comments on earlier drafts of this paper.

References

[1] M. Abi-Antoun and J. Aldrich. Compile-Time Views of Execution Structure Based on Ownership. In *Intl. Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, pages 81–92, 2007.

[2] M. Abi-Antoun and J. Aldrich. Ownership Domains in the Real World. In *Intl. Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, pages 93–104, 2007.

[3] M. Abi-Antoun and J. Aldrich. Static Extraction of Sound Hierarchical Runtime Object Graphs. Technical Report CMU-ISR-08-127, Carnegie Mellon University, 2008.

[4] M. Abi-Antoun, J. Aldrich, and W. Coelho. A Case Study in Re-engineering to Enforce Architectural Control Flow and Data Sharing. *J. Systems and Software*, 80(2):240–264, 2007.

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

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

[7] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *ICSE*, pages 187–197, 2002.

[8] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic Coupling Measurement for Object-Oriented Software. *TSE*, 30(8):491–506, 2004.

[9] B. Bokowski and J. Vitek. Confined Types. In *OOPSLA*, November 1999.

[10] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a Case Study: its Extracted Software Architecture. In *ICSE*, pages 555–563, 1999.

[11] A. Christl, R. Koschke, and M.-A. Storey. Equipping the Reflexion Method with Automated Clustering. In *WCRE*, 2005.

[12] D. Clarke, J. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, 1998.

[13] P. Clements et al. *Documenting Software Architecture*. Addison-Wesley, 2003.

[14] W. Dietl and P. Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8), 2005.

[15] W. Dzidek, E. Arisholm, and L. Briand. A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. *TSE*, 34(3):407–432, May-June 2008.

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

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

[18] hyperCision Inc. jMetra. `www.hypercision.com`, 2008.

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

[20] jRM Tool. `http://jrmtool.sourceforge.net`, 2003.

[21] R. Kazman and S. J. Carriére. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Softw. Eng.*, 6(2), 1999.

[22] D. Kirk, M. Roper, and M. Wood. Identifying and Addressing Problems in Object-Oriented Framework Reuse. *Empirical Software Engineering*, 2006.

[23] B. Kitchenham, L. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, 1995.

[24] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zundorf. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In *WCRE*, pages 22–32, 2002.

[25] H. Koning, C. Dormann, and H. van Vliet. Practical Guidelines for the Readability of IT-Architecture Diagrams. In *SIGDOC*, 2002.

[26] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP*, pages 275–302, 2003.

[27] Lattix Inc. LDM tool. `www.lattix.com`, 2008.

[28] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems. *TSE*, 31(3):256–272, 2005.

[29] A. Milanova. Static Inference of Universe Types. In *Intl. Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, 2008.

[30] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *TSE*, 27(4):364–380, 2001.

[31] R. W. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie Mellon University, 2001.

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

[33] D. Riehle. *Framework Design: a Role Modeling Approach*. PhD thesis, Federal Institute of Technology Zurich, 2000.

[34] J. Schäfer, M. Reitz, J.-M. Gaillourdet, and A. Poetzsch-Heffter. Linking Programs to Architectures: An Object-Oriented Hierarchical Software Model based on Boxes. In *The Common Component Modeling Example: Comparing Software Component Models*, 2008.

[35] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. *TSE*, 32(7):454–466, 2006.

[36] M. Sefika, A. Sane, and R. Campbell. Architecture Oriented Visualization. In *OOPSLA*, 1996.

[37] F. Shull, F. Lanubile, and V. R. Basili. Investigating Reading Techniques for Object-Oriented Framework Learning. *TSE*, 26(11):1101–1118, 2000.

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

[39] 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.

APPENDIX

Figure 4: Developer's diagram.

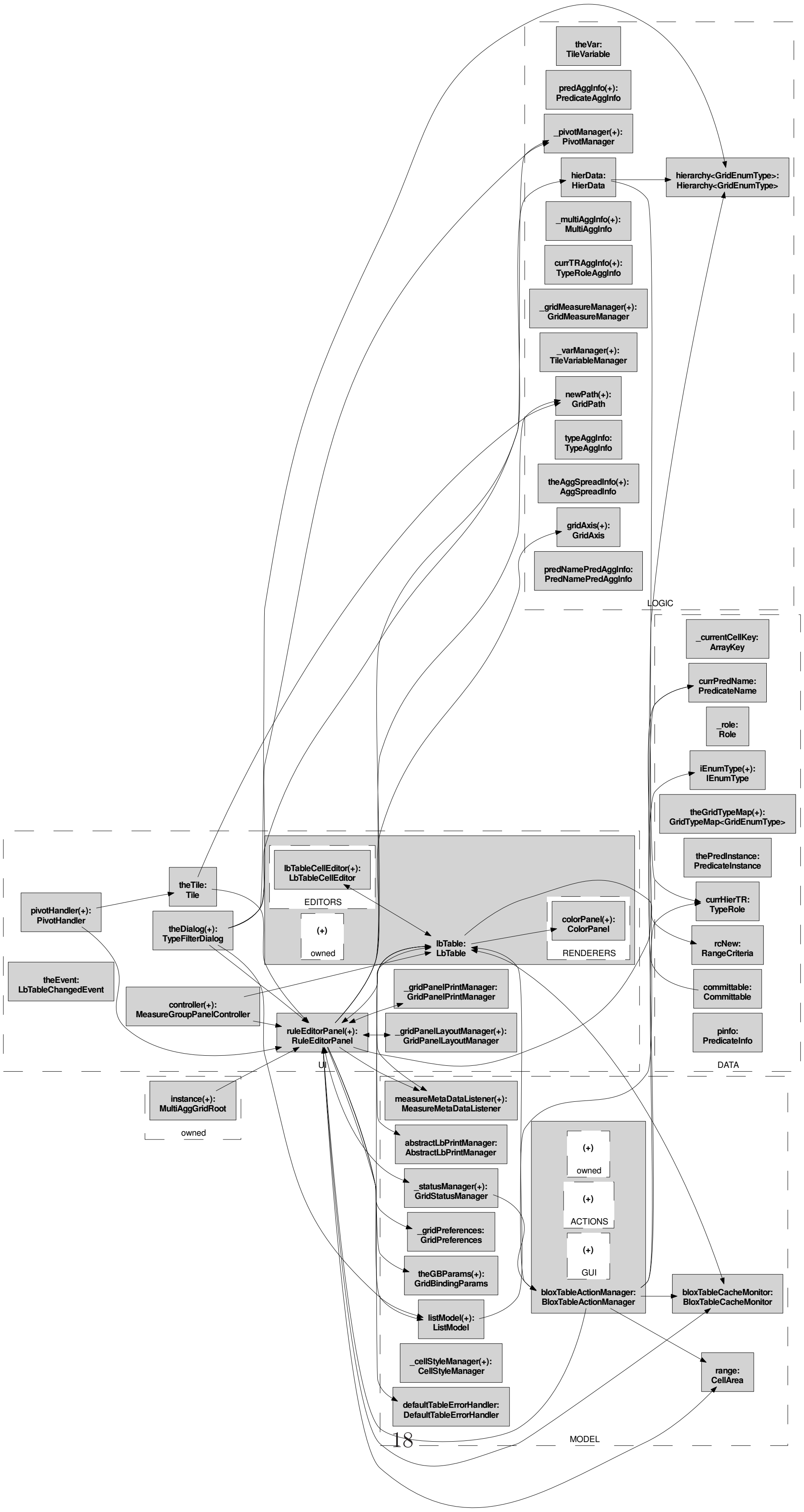


Figure 5: Extracted runtime architecture.