Qualitative Assessment of the Usefulness of Diagrams of the Run-time Structure to Answer Developer Questions during Code Evolution Tasks

Nariman Ammar  Marwan Abi-Antoun

December 2010

Department of Computer Science
Wayne State University
Detroit, MI 48202

Abstract

During code evolution tasks, developers often use high-level views of the system. Different architectural views are often used to describe a software system, including code views and run-time views. A code view describes class dependencies at compile time, whereas a run-time view describes interactions between different objects at run-time. Understanding these object interactions can help developers answer some of their questions about the object structure of the system, while doing code modifications. We conducted a longitudinal case study where we observed the architectural extractor add annotations and extract diagrams of the runtime structure. We then observed a developer perform a code modification task using information from the extracted diagrams. In this paper, we describe how the architectural extractor incrementally annotated the system to extracted diagrams of the runtime structure. Then, we qualitatively assess the usefulness of the extracted diagrams for developers, by describing how these diagrams can be refined to reflect the developer's mental model of the system, and how developers can use the refined diagrams in addition to diagrams of the code structure to perform code modifications.
Keywords: case study, empirical evaluation, ownership domains, runtime structure, object diagrams
1 Introduction

During code evolution tasks, developers often use diagrams to gain a high-level understanding of the system. Different architectural views are often used to describe a software system, including code views and run-time views. A code view describes class dependencies at compile time, and is typically depicted by a UML class diagram. A run-time view describes interactions between different objects at run-time. Understanding these object relations is essential to answer developers questions about the object structure while doing code modifications.

Some tools display partial views of the run-time structure of a system, based on running and monitoring the system, e.g., [17]. We use the approach by [3] to annotate an object-oriented system, and use static analysis to extract a hierarchical object graph (OOG) that soundly approximates a run-time object graph that any program run may generate. An extracted OOG is sound in two respects. First, each run-time object has exactly one representative in the OOG. Second, the OOG has edges that correspond to all possible run-time points-to relations between those objects.

Most of the information available in a reverse-engineered UML class diagram can also appear on an OOG. In fact, the OOG displays even more precise information which a class diagram cannot provide. A node on a class diagram represents a class with the attributes and the methods used by all instances of that class, whereas the nodes on an OOG represent the different instances of the classes that appear in the class diagram. Moreover, an OOG has the ability to show a hierarchy of objects which neither a class diagram nor a flat object graph can provide. Finally, a class diagram groups classes into packages, modules or layers, whereas the OOG groups objects into run-time tiers.

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

To assess the usefulness of a reverse engineered diagram, one often compares the extracted diagram to a desired target architecture, or ask the original designers of the system if the extracted diagram reflects the design intent, [16]. Here, we evaluate whether the extracted OOG can be useful for code modification tasks, so we refine the extracted diagram to reflect the developers mental model. OOG1, OOGn in Figure 1 refer to the different versions of the extracted diagrams that we provide to developers in addition to diagrams of the code structure to perform the code modification.

This paper’s contributions consist of:

• A longitudinal case study evaluating the usefulness of diagrams of the run-time structure for a developer doing code modifications;
• A detailed description, using examples from the actual code, of how the code was incrementally annotated to extract diagrams that can help answer developers questions about object relations;
• A description of the refinement process of the extracted diagrams to reflect the developer’s mental model.

1
This paper is organized as follows: in Section 2 we provide brief description of the subject system and the SCHOLIA approach. We describe the study setup and methodology in Section 3, and provide a qualitative analysis of the results in Section 4. We then discuss the study limitations, lessons learned, and future work in Section 5. Finally, we talk about related work in Section 6 and conclude.

2 Background

In this section, we briefly describe the subject system used in this study. Then, we introduce the type system and the SCHOLIA approach to extract diagrams of the run-time structure illustrated by examples from the code.

2.1 The DrawLets Framework

The subject system in this case study is DrawLets [10] version 2.0, which consists of 115 classes, 23 interfaces, and 12 packages. DrawLets is an object-oriented framework implemented in the Java language for building graphical applications. DrawLets supports a drawing canvas that adds figures to a drawing and lets users interact with them using tools. Figures include lines, freehand lines, rectangles, rounded rectangles, triangles, pentagons, polygons, ellipses, and text boxes. Drawing canvases created with DrawLets can be part of a larger application and need a GUI specific placeholder to allow them to reside within the application’s GUI. In the version that we used for DrawLets, DrawingCanvasComponent was used to allow the drawing canvas to reside within an AWT application. Figure 2 is a UML class diagram of the core interfaces in DrawLets and their relations.

Locators. Locators are the preferred means of specifying position in DrawLets. When combined with LocationListeners, they allow constraints to be applied to Figures based on location. Three interfaces define the functionality which Locators provide: Locator itself is able to provide its x, y, r and theta; MovableLocator can additionally manipulate these values; and RelativeLocator is a Locator whose position is based on some other position. In the example of FigureRelativePoint(Fig 17), which implements

---

Figure 2: A Class diagram for DrawLets. Source: [14]
RelativeLocator (which extends Locator), the position returned is based on the position of a Figure to which the FigureRelativePoint is related.

Figure 3: An Ownership Object Graph of DrawLets (OOG3).

LocationListeners. To update a figure’s position dynamically, DrawLets uses the notion of a LocationListener. Basically, when a Figure A decides to depend on another Figure’s (B) position, through a RelativeLocator, A will register itself as a LocationListener with B, and when B’s location changes, it will notify A. A is then responsible to update itself accordingly, through asking a FigureRelativePoint for its new coordinates.

Tools. Tools are InputEventHandlers that act on drawing canvases and modify the figure attributes, such as size and location. Tool classes include the SelectionTool and the ConstructionTool, and these extend the CanvasTool class. Several classes extend the ConstructionTool class such as the ShapeTool which has several subclasses (e.g. EllipseTool, RectangleTool, and PolygonTool).

Handles. Handles vary in behavior. They provide the ability to do things ranging from resizing Figures to creating Lines. Handles can be attached to any part of a Figure, and they move when the Figure moves. Several types of general purpose handles are already included with DrawLets. The core types are the Handle interface and the CanvasHandle class.

2.2 The Ownership Domains Type system

[3] previously proposed SCHOLIA where the architectural extractor uses the ownership domains type system, implemented using Java 1.5 annotations and the Eclipse infrastructure, to annotate the references of non-primitive types in an object-oriented Java code. The annotations explicitly express the architectural intent related to object encapsulation, logical containment, and architectural tiers, which are not explicit constructs in general-purpose programming languages. The annotations also specify and enforce the sharing of data between objects, which is a key challenge in extracting run-time structures. This state sharing is often not explicit in object-oriented programs, rather, it is implicit in the structure of the references that are created at run-time. The architectural extractor then uses static analysis to extract a hierarchical Ownership Object Graph (OOG) from the annotated code. The OOG is a diagram that depicts the run-time structure of the system, by showing instances rather than types in the code.

2.2.1 Ownership Domains

An ownership domain is a conceptual group of objects with an explicit name and explicit policies that govern how a domain can reference objects in other domains. Each object is assigned to a single ownership domain,
but it can declare multiple domains which can be either public or private. Each object has a default private
domain owned and can declare one or more public domains to hold its internal objects. Objects inside the
owned domain are strictly encapsulated, so they can be accessed by external objects only through their
parent object. On the other hand, objects inside a public domain are logically contained and can be ac-
cessed by any object which can reference the parent object. For example, figure 3 illustrates that the object
VIRT_observerList:Vector<Observer> is strictly encapsulated within the model:SingleDrawingModel object,
and is represented by a white box with a thick dashed border. The adapter:ValueAdapter object is
logically contained since model.SUBS is a public domain inside model:SingleDrawingModel and is repre-
sented by a white box with thin dashed border.

Domain names. For domains, the architectural extractor can choose names that convey design intent such
as MODEL, UI, and SUBS. He can also use one of the following special annotations that are defined in the type
system and need not be declared:
- lent annotation. This indicates a temporary alias of an object within the current method. It is used to
  annotate objects that are passed as method parameters which gives the method access to the parameter
  objects in a time-bounded way, i.e., for the duration of the call without storing a persistent reference
to the object.
- unique annotation. This is used on unaliased objects, such as newly created objects or those objects
  passed linearly from one domain to another. It is considered the best annotation for unshared objects
  in an object-oriented system.
- shared annotation. Objects annotated with shared are shared globally across the system. This
  annotation is considered the worst case annotation for objects that have no owning object, and was
  mainly designed to deal with legacy code or third-party libraries.
- owner annotation. The type system implements the owner domain, and adds it to the list of domain
  parameters declared on each class (declaring type). The owner parameter always occurs as the first
  element in the list of domain parameters. The architectural extractor can use the owner implicit
  parameter to make the annotations less verbose. For example, he do not need to explicitly declare
  the V domain parameter on the type SimplePanel, instead, he can use the implicit domain parameter
  owner to indicate that object references inside SimplePanel that should be part of the UI domain are
  part of the owning domain of SimplePanel (i.e. UI).

2.2.2 The Annotation Language

We summarize the types of annotations used in the type system illustrated by examples from the DrawLets
code in Figure 4.
- @Domains declares one or more domains. For example, the SingleDrawingModel type declares two
domains: owned and SUBS using the annotation @Domains(owned, SUBS).
- @DomainParams declares a set of formal domain parameters on a type (e.g. SimpleModelPanel declares
  one domain parameter M using the annotation @DomainParams(M)). Domain parameters are necessary
  to provide object sharing across domains as discussed in Section 3.2.
- @Domain is added on an object declaration to place that object in the corresponding domain
  (e.g. @Domain(owned)). It can also be used to bind an actual domain to a previously declared formal
domain parameter (e.g. @Domain(M<owner>).
- @DomainInherits specifies parameters for supertypes by binding the current type’s formal parameters
to the parameters of supertypes. For example, SimpleModelPanel inherits from SimplePanel,
so it binds its domain parameter M to the SimplePanel domain parameter M using the annotation
@DomainInherits(SimplePanel<SimplePanel<M>>).

Second layer of annotations. The ownership type system has a second layer of annotations to enforce
architectural constraints between run-time tiers, and include the following:
- @Domainlinks declares domain link specifications to give objects in one domain permission
  to access objects in the private domains of other objects. For example, the annotation
  @DomainLinks(SUBS->owned, SUBS->V) on the SingleDrawingModel type gives permission for
public class SimpleModelPanel extends SimplePanel{
    protected @Domain("owned") Panel commandPanel;
    protected final @Domain("M<owner>") SingleDrawingModel model = new SingleDrawingModel();
    ...
}

public class SingleDrawingModel{
    ...
}

public class SimplePanel extends Panel {
    protected @Domain("M<owner>") DrawingCanvas canvas;
    protected @Domain("owner<M>") ToolPalette toolPalette;
    protected @Domain("owner<M>") StylePalette stylePalette;
    protected @Domain("owner<M>") ToolBar toolBar;
}

Figure 4: Examples on the annotation language using snippets from the DrawLe ts code.

objects in the SUBS domain to access objects in the owned domain and the V domain parameter.
• @DomainAssumes declares domain link assumptions. For example, SingleDrawingModel assumes that instances in the owner domain has access to instances in the V domain parameter.

2.3 The Ownership Object Graph (OOG)

An OOG is composed of nodes and edges that show the interactions between these nodes. Figures 3, 16, 18, and 19 show different extracted OOGs for DrawLe ts. Figure 3 illustrates how a node can be either an object (e.g. model:SingleDrawingModel) or a domain within this object (e.g. SUBS). Each object node has a unique parent domain, and each domain node has a unique parent object. Edges between objects, correspond to points-to relations that show how these objects are related, and edges between domains (not showing) enforce architectural constraints between run-time tiers. A points-to relation on the graph can be traced to a field declaration in the code. The OOG is hierarchical, meaning that an object on the OOG can contain one or more objects as part of its substructure. For example, in Figure 3 the plus symbol indicates that the panel:Panel object has internal objects. Nested objects often reside in domains which could be either public or private.

3 Case Study

3.1 Tools and Instrumentation

During the case study, the architectural extractor and the developer doing code modifications used several tools, [3], discussed below.

The Defaulting Tool. This tool was implemented in Eclipse to reduce the annotation burden on architectural extractors.
The Type Checker, ArchCheckJ. This tool implements the ownership domains type system, using Java 1.5 annotations and the Eclipse infrastructure. The architectural extractor adds annotations to the code, then he runs the type checker to check for warnings. The type checker is modular, so the architectural extractor can use it to type check the annotated code one class at a time. The tool also has features such as checking domain links and supports annotating external library code using XML files, AliasXML.

The OOG Extraction Tool, ArchRecJ. This is another static analysis tool that scans the annotated program’s abstract syntax tree and produces an OOG. The extracted OOG can then be given to the developer both in XML and PDF formats. The tool also has features that enable the architectural extractor to manipulate the extracted OOG before he gives it to the developer.

The OOG Viewer Tool. Figure 5 show a screenshot of this tool. It is a modeless dialog that allows developers navigate the OOG while concurrently editing code in Eclipse. The developer can load the XML files generated by the OOG Extraction tool to visualize the extracted OOG for interactive usage.
3.2 The Annotation Process

In this section, we discuss the annotation process and the extraction and refinement of the OOGs. We illustrate by actual examples from DrawLets, how the defaulting tool adds default annotations to the code and how ownership domain annotations can be refined to express and enforce the design intent related to object encapsulation and communication.

1. Generating XML files. The architectural extractor started by generating XML files from Java standard library classes to be able to add annotations. Most of the time he used XML files from previously annotated systems. Figure 6 shows that metrics object has a shared annotation, and is then assigned to ToolKit.getDefaultToolkit().getFontMetrics() which is part of java.awt.Toolkit library. Figure 7 shows the corresponding XML file which shows that the return type of getFontMetrics() method is annotated with shared.

```java
class BasicStringRenderer{
    protected @Domain("shared") FontMetrics metrics;
    public void setFont(@Domain("lent") Font font) {
        if (font != getFont()) {
            metrics = ToolKit.getDefaultToolkit().getFontMetrics(font);
            reset();
        }
    }
}
```

Figure 6: Example of external library code.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<class id="Ljava/awt/Toolkit;" name="java.awt.Toolkit">
    ...
    <method id="..." name="getFontMetrics">
        <param domain="" id="Ljava/awt/Font;" name="Font" />
        <return domain="shared" type="java.awt.FontMetrics"/>
        <receiver domain="" paramActuals="" paramArrays="" />
    </method>
</class>
```

Figure 7: Part of the annotations added to java.awt.Toolkit.xml.

2. Inferring generic types. The architectural extractor then used the refactoring tool in Eclipse to infer generic types on some loosely typed code (e.g. Object). After that he added annotations on the container elements (e.g. Vector and HashTable), and to handle generic types, he used nested parameters (e.g., @Domain("owned<owner<V>>") in Figure 8). Figure 9 shows portion of the extracted OOG after inferring generic types. Notice that he got a more precise OOG since it shows that the canvas:SimpleDrawingCanvas object has distinct Vector instances.

3. Applying the defaulting tool. The architectural extractor then applied the defaulting tool to add most of the annotations automatically to the code. Figure 10 illustrates how the defaulting tool annotates temporary local variables within method bodies (e.g. button) and method formal parameters (e.g. label) with lent. It also annotates private fields (e.g. canvas) and return types of private methods (e.g. return type of addButton()) with owned. Variables of type String are annotated with shared.

4. Organizing core types into top-level domains. In previous studies on ownership domain annotations, the architectural extractor relied on design documents to help him understand the design intent in the code. He had such documents either as class diagrams or sketches drawn by the original designers.
public class SimpleDrawingCanvas {
    // Before
    @Domain("owned") Vector handles;
    @Domain("owned") Hashtable figureHandles;
    // After
    @Domain("owned<owner<V>>") Vector<Handle> handles;
    @Domain("owned<owner,owned,owned>")
    Hashtable<Figure, Handle[]> figureHandles;
    ...}

Figure 8: Inferring generic types on object declarations in SimpleDrawingCanvas class.

Figure 9: Part of the OOG showing the substructure of canvas:SimpleDrawingCanvas object after inferring generic types.

of the system. In the case of DrawLets, he did not have access to any of those documents, but he relied on his domain knowledge, the code structure, and informal comments in the code to capture the design intent. For example, he knew that many applications can be organized into three tiers: User Interface, Logic and Data tiers. He also knew that applications that follow the Model-View-Controller architecture can be optimized using the Document-View architecture. So, he decided to use two top-level tiers: MODEL and UI. Then, to organize the core types in DrawLets into these tiers, he relied on the code structure. For example, the type hierarchy for SimpleModelPanel suggested that this class ultimately inherits from the AWT Panel class which is a GUI component, so he decided to place instances of SimpleModelPanel inside the UI tier. Whereas Drawing and DrawingCanvas are part of the MODEL. A Drawing has a list of Figures, and the DrawingCanvas has a list of Handles to allow user interactions. Figure 19 shows the extracted OOG of DrawLets after annotating the instances of the core types with the corresponding ownership domains as described below. The figure also shows how each tier can be represented as an ownership domain.

5. Creating a root class. After selecting the top-level tiers, the architectural extractor needs to declare these tiers as domains. So he starts by annotating the main class in the system which is SimpleModelPanel in the case of DrawLets. However, the annotation system prohibits an object from placing itself in an ownership domain that it declares. So, he had to add an extra level of indirection by creating a fake top-level class (Main) in which he declared an instance of SimpleModelPanel. Figure 11 illustrates how the architectural extractor declared the top-level domains (MODEL and UI), and how he used the @Domain("UI") to declare the reference panel of type SimpleModelPanel in the UI domain. The architectural extractor then ran the extraction tool to get the OOG in Fig. 12. The figure shows a partial OOG with the dashed box labeled with lent indicating the lent domain. This
public abstract class CanvasPalette extends Panel{
    protected @Domain("owned") DrawingCanvas canvas;
    protected @Domain("owned") Button addButton(@Domain("shared") String label) {
        @Domain("lent")
        Button button = new Button(label);
        ...
        return button;
    }
}

Figure 10: Code automatically annotated using the defaulting tool.

public class Main {
    @Domain("UI < MODEL >")
    SimpleModelPanel panel = new SimpleModelPanel();
    public Main() {
    }
    public void run() {
        try {
            @Domain("lent")
            java.awt.Frame frame = new ExitingFrame( "");
            frame.add("Center", panel);
            frame.setSize(panel.getSize());
            frame.setVisible(true);
        } catch (@Domain("lent") Throwable exception) {
            System.err.println("Exception occurred in main()";
            exception.printStackTrace(System.out);
        }
    }
    public static void main(...) {
        @Domain("lent") Main system = new Main();
        system.run();
    }
}

Figure 11: Annotations added to the root class Main.

domain contains the system object that is represented by a solid box. The system object has a domain called UI that in turn has the object reference panel of type SimpleModelPanel.

6. Running the type checker one module at a time. After annotating the top-level class, the architectural extractor relies mostly on the type checker. The type checker is modular which helps the architectural extractor annotate the remaining code one class at a time starting with super types and moving to their subtypes by chasing the inheritance hierarchy. The type checker helps answer local questions and get hints on how to annotate the remaining code. For example, the architectural extractor started annotating the type SimpleModelPanel which is a subtype of SimplePanel, so he knew that he should first annotate SimplePanel then go back to SimpleModelPanel.

7. Declaring domain parameters. Once the architectural extractor declares the two top-level ownership domains, Model and UI, he passes the corresponding domain parameters to the various types to allow object sharing across domains. The architectural extractor used domain parameters M, to represent the MODEL, and V, to represent the UI. Figure 4 shows how the architectural extractor used domain parameters to annotate objects inside SimpleModlPanel. For example, he knew that the field model
should be declared in the MODEL domain, so he annotated this object with the domain parameter \( M \).

Figure 4 shows how he also declared a formal domain parameter \( V \) on the type `SingleDrawingModel` to hold instances in the UI domain.

8. Binding domain parameters. The need for domain parameters propagates the annotation warnings introduced by the type checker which requires the architectural extractor to bind more parameters to resolve these warnings. Figure 11 shows how the architectural extractor added `<MODEL>` to the annotation on the `panel` instance to bind the `SimpleModelPanel::M` formal parameter to `Main::MODEL` actual domain. The figure also shows how he mapped the `SingleDrawingModel::V` formal parameter (Fig 4) to the implicit domain parameter `owner` inside the `SimpleModelPanel` class which represents the UI domain. Recall that `SimpleModelPanel` extends `SimplePanel` and this explains the use of `@DomainInherits("SimplePanel\(<M>\)")` on the class declaration. Figure 13 shows the extracted OOG after applying these annotations. Notice the direction of communication between objects in the different domains, since the architectural extractor intended to allow objects in domain UI to refer to objects domain MODEL, but not vice versa.

9. Modifying annotations added by the defaulting tool. The defaulting tool is not a smart inference tool, so the architectural extractor had to manually modify some of the imprecise default annotations. Figure 11 shows that `SimpleModelPanel` class has two fields: model reference of type `SingleDrawingModel`, and commandPanel reference of type `Panel`. Notice that both `model` and `commandPanel` are protected fields, however, using ownership domain annotations the architectural extractor can make instances either strictly encapsulated, logically contained, or he can move them in a separate domain. For example, `commandPanel` is strictly encapsulated within the substructure of `SimpleModelPanel` instance, so did not change the `@Domain("owned")` annotation added by the defaulting tool which placed this object in the private domain `owned` of `SimpleModelPanel`. On the other hand, the `model` instance is not strictly owned by `SimpleModelPanel` instance, as it should be accessible by objects in the other domains (e.g. `SimpleDrawingCanvas`). To express that, the architectural extractor changed the default annotation `owned` on this object to the `M` domain parameter to move it to the MODEL domain.

Figure 14 shows the extracted OOG. Notice that more objects are now displayed on the extracted OOG since they have more precise annotations. The figure shows that instances of type `SimpleModelPanel`...
Figure 14: The extracted OOG after changing default annotations in SimpleModelPanel and SimplePanel. 

(which is a subtype of SimplePanel) also point to instances of types ToolPalette, StylePalette, and ToolBar. Recall that the formal domain parameter SimplePanel::M was initially declared to hold instances of DrawingCanvas, which was bound to the domain Main::MODEL, and this explains why these objects now appear in the MODEL domain on the extracted OOG.
public class SimpleModelPanel extends SimplePanel {
  protected final @Domain("M<owner>") SingleDrawingModel model = new SingleDrawingModel();
  @Domain("model.SUBS<M<owner>,M<owner>>")
  ValueAdapter adapter = new ValueAdapter(model, "getDrawing", canvas, "setDrawing");
}

@Domains("owned","SUBS")
public abstract class BasicObservable {
  protected @Domain("owned") Object observerList;
  private @Domain("owned<SUBS>") Vector<Observer> VIRT_observerList = new Vector<Observer>();
}

@Domains("owned")
@DomainParams("M","T")
public class ValueAdapter implements Observer {
  @Domain("M") Object model;
  @Domain("T") Object target;
}

Figure 15: Annotations added in SimpleModelPanel, BasicObservable, and ValueAdapter to express the Observer design pattern.

10. Extracting OOGs. The architectural extractor often finds it useful to use the OOG extraction tool to visualize the added annotations. Since the purpose of the study was to extract an OOG that can be useful for developers, he wanted to get an OOG that is at an adequate level of abstraction yet expresses the design intent, and shows all possible interactions between objects. To express these concepts in the extracted diagram, the architectural extractor refined the annotations on object declarations in the code. The added annotations resulted in the diagram in Figure 19.

The remaining warnings The annotation process is iterative, so the architectural extractor repeated steps 6 through 10 until he fixed most of the annotation warnings produced by the type checker. In the end, he was able to annotate most of the code with only 56 remaining warnings, which are either due to inherent expressiveness challenges in the type system, or bugs in the tools that we are currently fixing.

3.3 Refinement of the OOG to Match the Developer’s Mental Model

To confirm that the extracted OOG was useful to developers, we asked a developer to use the OOG and report whether it is useful or not. In this section we report the developers feedback and the refinement of the OOG to reflect the developer’s mental model of the system. We first give a general overview of the two types of architectural abstraction that can be applied to an OOG. Then, we highlight some of the cases where the architectural extractor used these two types of abstraction to get an OOG that reflects the developer’s mental model of the system.
3.3.1 Architectural Abstraction of the OOG

An OOG provides two types of abstraction that make it scale. The first form of abstraction is by ownership hierarchy, by pushing more architecturally relevant objects, such as data structures, underneath more architecturally relevant ones, such as objects that are application specific. The architectural extractor can do that by modifying the annotations on object declarations, which causes these objects to move between domains at different levels of the ownership hierarchy. To annotate architecturally significant objects, the architectural extractor used more precise annotations such as `owner`, `UI`, and `MODEL`. He treated less architecturally significant objects (e.g. `Method` and `Color` in Fig 22) with less precise annotations such as `shared` and `lent`. The other form of abstraction is by types, where the architectural extractor can use the notion of subtyping to specify the architecturally relevant types, such as core framework interfaces, and use these types to merge related objects that reside in the same domain to get a less cluttered graph. In the study, the architectural extractor worked on improving the quality of the extracted diagram based on the developers mental model of the system either by using more precise annotations or by using special features in the OOG extraction tool as discussed in the following sections.

Manipulating the ownership hierarchy. The developer thought that the OOG in figure 16 was showing the `encompass:Rectangle` and `polygon:Polygon` objects that are not architecturally significant, and not worth appearing in the top-level domains. To solve this problem, the architectural extractor annotated these objects with the `shared` annotation which caused them to be moved to the shared domain, which is a global domain that is not displayed by default on the extracted OOG.

Merging objects that share a common supertype. The architectural extractor thought that it would be better to get a more abstract diagram by showing only the `Panel` instance instead of all different instances of `Panel`. The architectural extractor annotated `SimplePanel`, `StylePalette`, and `ToolPalette` instances with `owner` which caused these objects to be treated as siblings to each other and appear in the same
enclosing domain, UI (see Fig 13). Object merging occurs when two or more objects of the same type (i.e. same declared type or subtype thereof compatible types) reside in the same domain, which causes these objects to be collapsed as one object on the extracted OOG.

Controlling the merging of objects using abstraction by types. The OOG extraction tool has a feature to abstract objects by types and enable the architectural extractor to refine the extracted OOG using the notion of subtyping to specify the architecturally relevant types and choose those types as representatives for the merged objects. To choose the relevant types the architectural extractor has two options: abstracting objects by trivial types, and abstracting objects by design intent types. Design intent types are the types of objects that should be considered when the OOG is extracted, whereas trivial types are those types that should not be considered.

The architectural extractor used this feature several times during the study. For example, to get a more abstract OOG than the one in Figure 13, he looked at the type hierarchy of the Panel interface, which suggested that SimplePanel, StylePalette, and ToolPalette implements ultimately the Panel interface. So, he selected the type Panel to represent the three instances as one box panel:Panel on the extracted OOG. However, when the developer started using the diagram, she considered showing Panel alone instead of instances of all different subtypes of Panel to be not so useful, especially since Panel is considered part of Java AWT. To address this concern, the architectural extractor fine-tuned the abstraction by types to control the excessive merging between objects, and have the OOG display those objects as separate boxes (Fig 19). Also, to add design intent types to the extracted OOG, he studied the core interfaces in the DrawLets framework, such as Handle and Locator and added annotations to the different instances in the code which caused these instances to appear on the extracted OOG (Fig 16). But he wanted to get a more abstract diagram than the one in Figure 16 by reducing the clutter in the top level domains, so he fine tuned the annotations on locator and handle instances which caused these objects to appear in the same domain. Then, he used the abstraction by types feature to cause these objects to be merged under their super types based on the type hierarchy of the Locator interface (Figure 17). Figure 17 also shows the merged objects of these types under locator:Locator instance (i.e. relativePoint:RelativePoint, figureRelativePoint:FigureRelativePoint, and locator:DrawingPoint).

From the above two examples, we can see that the study identified two cases where object merging could be either useful or too abstract and cause confusion. The architectural extractor can always fine-tune the abstraction by types to reduce the excessive merging of objects, and have the OOG display those objects as either one box (e.g. Locator instance) or separate boxes (e.g. different instances of Panel) based on the developer’s feedback.

Figure 17: Merging different instances of Locator under their common supertype.
3.3.2 Decorating Objects with Labeling Types

The OOG extraction tool non-deterministically selects a label for a given object based on the name or the type of one of the references in the program that points to this object. If the architectural extractor specifies an optional list of labeling types for labeling objects, the tool then adds the type decoration to an object label if it merges at least one object of that type. For example, the architectural extractor added the `InputEventHandler` interface as a labeling type which caused it to appear as a decorator on the `tool: CanvasTool` object (Fig. 3).

3.3.3 Highlighting Implemented Design Patterns

In order to extract a diagram that expresses the observer design pattern the architectural extractor declared the `SUBS` public domain inside `SimpleModelPanel` and added the observers to that domain. The extracted diagram shows that the `ValueAdapter` (observer) listens to updates passed through the subject (`SingleDrawingModel`) which has a list of observers `VIRT.observerList`. When the developer highlights the edge between `model: SingleDrawingModel` and `adapter: ValueAdapter` instances and trace to the corresponding field declaration, they can clearly see that `SingleDrawingModel` class instantiates an `adapter` object and passes it the `model` and `canvas` objects as parameters. The diagram helps developers understand these relations only partially, so they can browse the code to understand the implementation details.

3.4 The Code Modification Task

We asked a developer to work on a code modification task on DrawLets. The task was to implement an “owner” for each figure: “An owner is a user who put that figure onto the canvas, and only the owner is allowed to move and modify it. At the beginning, each session declares a session owner, and this session
owner will own all new figures created in that session. No other user will be allowed to manipulate them. At the beginning of a session, user inputs ID and password. Any function that attempts to modify a figure must check that the figure owner and the current session owner are same. This change will allow the DrawLets framework to be used as a document collaboration tool”.[14].

In performing the task, the developer added two classes: a LoginDialog class to enable users to login to the system and a SessionOwner class which saves information about the current session. She also added a “New Session” button to the command panel inside SimpleModelPanel (or SimpleApplet) to enable the user to launch the login dialog and added the corresponding launchLoginDialog() method inside SimpleDrawingCanvas class. She also added the list of owners to both SimpleDrawingCanvas and AbstractFigure. To check the owner of a figure, she added isOwner() and isExisting() methods to the SimpleDrawingCanvas class. Finally, the developer added a call to the isOwner() method inside any method that attempts to change a figure attribute including SelectionTool and SimpleDrawingCanvas classes.

**Testing the modification.** Several actions can modify a figure in DrawLets, including changing the figure’s position, size, text or color, or linking it to another figure. The developer tested only three use cases: a user who tries to remove a figure which he does not own, a user who tries to move a figure which he does not own, and a user who tries to resize a figure which he does not own. In all three cases, she prompted the user to enter the correct ID and password. The three cases above required her to add the isOwner() method to each
of the following methods: `SimpleDrawingCanvas.removeFigure()`, `SelectionTool.mouseDragged()`, and she only attempted adding checks when resizing a figure through its handles, and she thought that would be related to either `BoundsHandle.resize()` or `SimpleDrawingCanvas.mouseDragged()`.

### 3.5 Data Collection

For data collection, we asked the developer to record her thought process to simulate the think-aloud protocol in the form of a transcript. Figure 20 shows excerpts from the log recording the developer’s thought process. It also includes the “Navigation event” which could be: a navigation with no target (e.g., scrolling, stepping in debugger) where the developer was rapidly skimming through text without a clear target, navigation commands that changed which method was currently visible (e.g., `ReferencesTo`), and navigation commands that generated some list of things that could be navigated to (e.g., `References`). So in the first case, the targets are things that are now visible whereas in the second case the targets are something on which a command was executed. The “Navigation target” could be a class name, a method name qualified by the class name, an OOG, a class diagram, or a certain feature in the OOG viewer tool. The code column was added later during the analysis process.

**The qualitative measures.** The qualitative measures that we consider when analyzing the developers thought process from the transcripts include: the developers perception of the OOG and whether they find it useful, the frequency of usage of the OOG and the viewer tool, what questions they were trying to answer.

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

Figure 20: Excerpts of the developer’s thought process recorded in the transcripts.
and using which features in the tool, finally we measure their wishes and what other features they like to see in the OOG viewer tool. The transcripts also have the classes that developers visited, and which features in Eclipse or in the OOG viewer tool itself they use.

4 Qualitative Analysis

For the qualitative protocol analysis, we use a model that we had previously defined to code the types of questions about object structures that developers ask during coding activities [5]. Among the things that we analyzed in the DrawLets study was the different classes that the developer visited, the number of visits per class, the number of times she used Eclipse features including debugging, call graphs, and file search, which diagrams she used, and how many times as discussed below.

![UML class diagram of DrawLets](Figure21.png)

**Figure 21:** A UML class diagram of DrawLets. Source: [14].

4.1 OOGs are complementary to class diagrams

According to the transcripts, the developer looked at the class diagram showing the core interfaces in DrawLets (Fig 2), but she did not use it since she could get the same information using Eclipse call graphs. She preferred to use the other diagram showing the dependencies between different classes in DrawLets(Fig 21) and she referred to that diagram twice. On the other hand, she used the OOG viewer several times; Table 1 lists the tool features that she used and their frequency of usage.

For illustration purposes, we annotated a UML class diagram (Figure 21) taken from a previous study on DrawLets, [14]. The figure explains how the developer in the study was able to obtain most of the information available in a class diagram by looking at the OOG in Figure 3. For example, the developer was able to see the object’s type as part of the object label (e.g. drawing object is of type SimpleDrawing).
Also, the OOG viewer tool with which the developers interacted while doing the task, enabled her to view the inheritance hierarchy of the types of the field declarations that an object merged (the “Type Inspector” dialog in Fig 5), even though she did not use this feature a lot (Table 1). Moreover, polymorphism makes things harder to developers, and they often switch between object and type structures. When the developer asked who implements this interface, “who” could mean a class or an object, and she often answered this question either by using type hierarchy feature in Eclipse or by looking at the labeling types on the OOG. We describe how the developer used the other tool features in Table 1 to answer some of her questions about object structure in the following sections.

4.2 OOGs can help answer developers questions about object structure

In this section, we illustrate by real examples from the code, how a developer interpreted the OOG, and how she went about using the OOG and the OOG viewer tool in order to explore the source code and answer some of her questions about object structure.

4.2.1 Answering Is-In-Tier Questions

Most of the time, developers want their modifications to fit the architectural intent of the design of the system. However, plain Java code does not directly convey any of that information. In the study, the instances that appeared in the top-level domains helped the developer understand better how the different objects are organized in DrawLets. Because she understood ownership domains and OOG extraction, she knew where on the extracted diagram the objects that she added might appear.

**Refactoring the modification to match the design.** Once the developer started thinking in terms of architectural tiers, she thought that calling the method to check a figure’s owner should be done inside the UI tier, whereas the implementation of the checking logic itself should be inside the MODEL tier. As a result, she moved the `isOwner()` method that she had already added to `SimpleApplet` to `SimpleDrawingCanvas`.

4.2.2 Answering Is-Owned and Is-Part-Of Questions

During the study, the developer was looking for the best way and location to modify the code. After several attempts and after visiting several classes, she added a button to the drawing canvas. After she started using the OOG, she realized that the OOG could have saved her the effort of visiting many classes. She noticed that she could have highlighted the edge between `panel:Panel` object and its nested object `commandPanel:CommandPanel` and trace to code to get to `SimpleModelPanel` class, where she could clearly see a list of other commands, e.g. save, print, etc..

**Refactoring the modification to match the design.** Having realized this fact, she knew exactly how the different commands are handled in DrawLets, and she noticed a certain pattern for dealing with actions in the `actionPerformed()` method. However, the method she added initially did not follow that pattern. As a result, she moved the `launchLoginDialog()` method from `SimpleDrawingCanvas` to `SingleDrawingModel`, to be consistent with the design.

**Does DrawLets strictly follow the two-tiered architectural style?** This led us to wonder how the developer was initially able to violate the pattern described above. The developer said that she was able to get hold of the `canvas` object inside `SimpleModelPanel` and `SimpleApplet`, and did not need to go through `SingleDrawingModel`. This means that DrawLets does not strictly follow the two-tiered style where objects in the UI tier should not have direct references to objects in the MODEL tier.

4.2.3 Answering Points-To Questions

The developer in the study could highlight the edge between `panel:SimpleModelPanel` and `model:SingleDrawingModel` which linked her to the corresponding field declaration in the Eclipse text editor (Fig. 5). The developer could not only navigate to code from the graphical representation of the OOG, she was also able to use a tree representation where she could search for an object in the ownership
tree by type or field name. She was also able to search for all the incoming or outgoing edges of a selected object and choose the relation that she was looking for and go to the specific line of code associated with this relation. The developer relied heavily on this feature during the study (Table 1).

The developer also learned useful information by exploring the edges between two objects. For example, she was searching for the method that was responsible for changing the figure’s position. The edge between figure:Figure and canvasTool:CanvasTool took her to SelectionTool.mouseDoubleClicked() using the trace to code feature. She found that this method instantiates a figure object which gets its value from canvas.figureAt(x,y) and dug deeper into this method. This eventually led her to getTool().mouseClicked() inside SimpleDrawingCanvas which gave her an indication that she could do the modification either inside SimpleDrawingCanvas or SelectionTool especially since the OOG shows that both implement the InputEventHandler interface.

4.2.4 Answering Has-A Questions

DrawLets uses the Observer design pattern where SimpleDrawingModel is the subject being observed and SimpleDrawingCanvas is the observer notified of the change (Fig. 3). the OOG showed that the ValueAdapter (observer) listens to updates passed through the subject (SimpleDrawingModel) which has a list of observers VIRT.observerList. When she highlighted the edge between model:SimpleDrawingModel and adapter:ValueAdapter instances and traced to the code, she found that SimpleDrawingModel class instantiates an adapter object and passes it the model and canvas objects as parameters. The diagram helped her understand these relations only partially, so she browsed the code to understand the implementation details.

4.2.5 Answering Is-A Questions

The developer in the study was aware of the important interfaces and wanted to see those interfaces on certain objects. So she asked the architectural extractor to set additional labeling types. For example, the SimpleDrawingCanvas implements the PropertyChangeListener interface and the CanvasTool is an InputEventHandler (Fig. 3). With those additional labels, she no longer needed to go, as often, to the Eclipse Type Hierarchy to determine which classes implement a given interface.

4.3 Diagram understanding

This study in addition to a previous one, [5], highlighted some cases where developers misunderstood the extracted visualization which may have led to confusion and reduced the OOG’s usefulness.

Developers can misunderstand some points-to relations. For example, if the type of an object implements some listener interface and the OOG merges these two objects, this could result in a points-to relation because the merging occurred and not because there is a direct points-to relation between these two objects.

Developers seem to have difficulty with object merging in the OOG. In some cases, the object labels were counter-intuitive. A possible tool feature would be to allow the user to interactively edit the labels on some key objects, or have the tool suggest possible labels from which a developer can choose as the preferred label, i.e. difference between recognition and recall. We could also add an icon to an object to indicate that the object merges objects of different types. We can also add a multiplicity symbol similar to the one used in the UML notation when collapsing multiple objects, and give an indication to the developer to turn off the feature to abstract objects by types.

Also, the studies identified that when a group of related objects are merged, it is hard to distinguish between the different merged objects in the OOG. This could be a limitation that could be overcome by using fully qualified names of the domains by their declaring classes. For example, in the OOG of DrawLets, the substructure of the figure:Figure object, shows several nested domains, obviously because DrawLets supports different types of figures. By adding the fully qualified domain names, i.e. EllipseFigure::owned or RectangleFigure::owned, instead of owned, the OOG would be more expressive.
5 Discussion

In this section, we discuss the limitations to the study, the lessons that we learned, and how we are planning to overcome these limitations in future studies.

5.1 The Limitations

There were some limitations to this case study that were either related to the design of the subject system itself, or the tools that we used to annotate the code or to extract the OOGs.

**DrawLets Design.** DrawLets seems to have been designed by professional object-oriented programmers. Still, we found a few places where the DrawLets code did not follow the best practice of using type safe declarations. The code also includes a few hacks when dealing with reflection. As a result, several casts may fail with runtime exceptions. Moreover, the use of reflective code poses challenges for the ownership annotations and the static analysis. For example, the entities that the analysis may not understand must be manually summarized using virtual fields in order to preserve the soundness of the extracted diagrams.

The DrawLets code deals with listener interfaces from the Java standard library, such as `java.util.EventListener`. There were a few cases where the architectural extractor could not respond to the developer’s requests due to limitations in the code design. Figure 19 shows an OOG with the `button:ColorButton` object showing in the top level view. The developer thought that this object is not architecturally significant and should not appear in the top level domains. The architectural extractor attempted to annotate the `button` object in `StylePalette` with `owned`, to push this object down in the hierarchy and reduce the clutter in the top-level view. However, the type checker produced additional warnings due to annotations on listeners in `ColorButton` class (Fig 22).

**The Annotation Process.** During the process of annotating the subject system, the architectural extractor had to refactor the code due to limitations in Java 1.5 annotations. For example, he had to extract local variables to fix some problematic code patterns such as returning new expressions and casting expressions. These refactorings were minor and did not change the program semantics. Another example of refactoring is when the architectural extractor wanted to annotate the code to express more clearly the observer design pattern, since `model` has to be declared `final`, so he can refer to its public domain `SUBS` as `model.SUBS`. This is relatively minor refactoring, since it only means that the `model` object is no longer lazily instantiated. There could be issues, however, if the design must support changing the model object dynamically.

Finally, there are some limitations in the current tools that might make the extracted OOGs not as useful to developers as they could possibly be. The extracted OOG in figure 3 shows that `model:SingleDrawingModel` object has a list of observers `VIRT.observerList`. However, this does not actually correspond to any field declaration in the code which makes the OOG less useful in this case. This is a current limitation in the tool where the `VIRT.XXX` notation corresponds to a “virtual field” annotation in the code that is manually added since the static analysis does not understand some implementation details in the code(Fig 15).

**OOG Extraction and Refinement.** In SCHOLIA, the extracted OOG may not be useful either due to bad types in the code or bad annotations added to the code. The architectural extractor adds annotations to encode some of the architectural intent in the system. However, in the case of DrawLets, we noticed that DrawLets code did not use a careful type structure to implement certain features, e.g. commands were treated as `Strings`. So, we could not treat some of the fields, that could be useful to developers, as architecturally significant objects. As a result, these objects did not appear on the extracted OOG especially since we move `Strings` to the `shared` domain.

The static analysis implemented by the OOG extraction tool does not show objects in the `shared`, `lent`, and `unique` domains by default, which could result in a low quality diagram which could be unsound. For instance, the analysis assumes that all objects marked as `shared` are in one domain, and due to merging of objects for soundness, the analysis may excessively merge objects that are in the shared domain. Unless the developer requests otherwise, the architectural extractor often purposely does not display the shared domain in an OOG. Displaying the shared domain would be trivial, but would add many uninteresting edges to an
OOD and add more clutter. We often try to mitigate this problem by moving only architecturally significant
objects to the other non-shared domains to make them visible on the extracted OOG.

```java
@Domains({"owned"})
@DomainParams({"M"})
@DomainInherits({"CanvasPalette<M>"})
public class StylePalette extends CanvasPalette {
    protected @Domain("owned<shared>") Vector<Color> colors = new Vector<Color>(5);
    protected @Domain("owned<shared>") Vector<Method> setters = new Vector<Method>(5);
    @DomainReceiver("owner")
    public void addColor(@Domain("shared") Color color,
        @Domain("shared") String label) {
        colors.addElement(color);
        @Domain("owner")
        ColorButton button = new ColorButton(label, color);
        button.addActionListener(this);
        ...
        if (!areButtons) {
            currentColorButton = button;
        }
    }
}

@Domains({"owned"})
public class ColorButton implements MouseListener {
    protected @Domain("owner") ActionListener actionListener = null;
    public void addActionListener(@Domain("owner") ActionListener listener) {
        actionListener = AWTEventMulticaster.add(/*<owner>*/ actionListener, listener);
    }
}
```

Figure 22: Part of the annotations in StylePalette and ColorButton.

During the course of this study, we did not focus on the second layer of annotations that includes domain
links and domain link assumptions between different domains. Later on and after we got the developers
feedback, we found the DrawLets design does not strictly follow the two tiered architectural style. Future
work includes adding the necessary annotations to enforce architectural constraints.

Using the ownership domain annotations helped the architectural extractor highlight some of the design
vulnerabilities in the code which he believed needed major or minor refactoring which he would never realize
without annotating the code. However, he annotated the system as is and did not try to fix any of the design
flaws in the code. This implies that the extracted OOG with which we provided the developer was also
expressing the design of the system. The developers feedback on the extracted OOG gave us some insights
on how to improve the extracted OOG by expressing more objects, decreasing the clutter, and highlighting
some design patterns. The architectural extractor tried to improve the quality of the extracted diagram by
adding more precise annotations which caused objects to be moved between domains, pushed underneath
other objects, or merged under other objects of the same type that are in the same domain.

The architectural extractor annotated several applications previously. However, annotating these systems
only helped him annotate the system more effectively. JHotDraw, for instance, followed the Model-View-
Controller architecture which made the process of adding annotations to JHotDraw an overkill, especially
since the `owner` annotation was not implemented yet, which required him to add more domain parameters to
the system. In order to annotate a new system, the architectural extractor often relies on the type checker
to build his knowledge incrementally. Then he extracts OOGs from the annotated code and run them by
the original designers of the system, or compares them to a certain as designed architecture. In this study,
however, the architectural extractor did not compare the extracted OOGs to any as designed architecture.
Figure 23: The top level view of the OOG shows all different instances of Figure.

He relied on several sources of information such as tracking the core interfaces and their type hierarchy. He also refined the OOG to match the developer’s mental model which might not reflect the original developer’s intent, and could be different from what any other external developer would have especially if he is performing a different code modification task on DrawLets.

Figure 24: Fine tuning “abstraction by types” causes different Figure instances to be merged under their supertype.

5.2 Lessons learned

The case study identified some flaws in the study design that could be avoided in future studies. **Improve the annotation process.** The annotation process was not carefully recorded by the architectural extractor, so we tried to track the changes made in the subversion history to record how the different types
were annotated. Also, the process of adding annotations was not controlled since there were no timing constraints. Based on our previous estimates, adding annotations to the code should take 1 hour per 1KLOC which was not measured in this study. To better observe the annotation process, we should have asked the architectural extractor to record a time log, as well as the thought process while adding annotations.

For the extracted OOG to reflect the entire system, the entire code must have annotations that typecheck. Adding annotations is currently a manual process. As a result, the architectural extractor did not add annotations to the code that the developer added or modified, to shield her from the annotation process. To guarantee that both the developer and the architectural extractor are working on the same version of the code, and that the developer is working on an OOG that reflects the additional code, we will ask the architectural extractor to annotate the new code to let the developers see the newly created objects on the extracted OOGs. Unfortunately, this step will add extra overhead to the architectural extractor.

Also, this study identified that developers often have initial feedback on the extracted OOGs, such as moving objects up or down in the hierarchy by refining the annotations or using tool support. However, the developer in this study was not able to directly manipulate the OOG, and was using only a viewer tool. To make the OOG even more useful, we will provide developers with the ability to interactively refine the OOG while doing the code modification. This is a capability for which we started developing tool support, [6].

**Design a better code modification task.** One of the flaws in the design of this study was having the developer record her thought process manually, which caused the study to run longer, and could have interfered with the natural way of programming. To overcome this issue, we will use a video recording mechanism such as Camtasia, and require the developer and the architectural extractor to think-aloud which will also help us observe their behavior better. Also, at no point in the study did we try to compare between effectiveness of developers using OOGs and those using class diagrams. This can be better measured by conducting more controlled studies by having different control groups work on the same task using the two types of diagrams, and design more precise measures to compare the results.

### 5.3 Future Work

This study had threats to validity, [4], which we are planning to avoid in future studies. To achieve construct validity, we will avoid choosing developers who are familiar with JHotDraw or DrawLets, or those already familiar with the process of adding annotations and extraction of OOGs. Also, having a control group working on class diagrams will help us support our hypothesis that developers who have access to diagrams of the runtime structure are better off than developers who have access to only the code structure. To achieve the necessary generalization, which we could not achieve in the current study, we will use multiple-case study design according to [18] by following these two routes in order:

1. Achieving statistical generalization. We are currently designing a controlled user study which will involve 10-12 developers, divided into 2 groups. We will ask one group to do a code modification task using OOGs and the other to do the same task using only class diagrams.

2. Achieving analytic generalization. This type of generalization is often more desired for case studies, [18], and helps achieve the necessary replication. We are planning to conduct another longitudinal case study using a real system that has a number of outside users in academia and industry. During the case study, we will observe the architectural extractor annotating the subject system and extracting OOGs. Then we will get the original developer’s input on how to refine the extracted OOGs. Finally, we will ask developers to use the OOG to do code modifications on real tasks submitted against the system by outside users.

### 6 Related Work

**Our previous studies on OOGs.** [7] previously conducted a field study to observe how developers understand object relations, and what tool features they need to convey their mental model of the system.
They provided a professional developer with an initial run-time architecture and refined it to convey his intent, but they did not use the refined diagram to do a code modification. In this study, we provided the developer with several extracted architectures based on her evolving mental model and we did not consider any specific architectures to be the authoritative one. Since the focus of this study is on the usefulness of the run-time structure, we tried to get a diagram that reflects more the developers mental model to be able to answer their questions about object structure.

In previous work, we conducted an exploratory study to identify whether developers ask questions about object relations during coding tasks, [5]. Our findings confirmed that developers do ask questions about object relations such as: points-to, is-owned, and may alias questions. Since we got promising results, we conducted this study to investigate whether the run-time structure can help answer some of these questions. Previously, we designed certain tasks that required asking questions about objects. In this study, however, we asked the developer to perform coding tasks designed by others, [14].

**Previous evaluation of ownership domains.** We have access to several previously annotated systems using ownership annotations including CryptoDB, JHotDraw(15 KLOC developed by experts), HillClimber(15KLOC developed by undergraduates), Aphyds(12KLOC written by an electrical engineering professor for one of his classes), and the LbGrid module. One of the earliest studies on alias java was trying to answer questions such as: is the annotation system practical on realistic application code? and Does the annotation system help to encode application-specific architectural constraints? [8]. [1] conducted a case study on JHotDraw and HillClimber to evaluate ownership domain annotations on real world applications. They also conducted a field study on the LbGrid module where they added ownership domain annotations to the code and observed a real developer using the extracted OOG from the annotated code and compare it to the architecture drawn by the original developer of the system, [2]. In this study and future studies we are taking one step further by demonstrating that SCHOLIA is feasible, and can produce useful diagrams for developers by expressing the design intent and architectural constraints in the system. Previous studies on ownership domains were mainly either to evaluate the ownership domains on real object oriented java code or to compare the extracted architectures to the as designed ones. However, none of them was intended to do code modifications using diagrams extracted from the annotated code.

**Other studies on the run-time structure.** [17] developed an approach for visualizing the operation of an object-oriented system at the architectural level. Their approach builds on the Reflexion Models technique, but uses the running summary model rather than the complete summary model. They allow developers to flexibly define the structure of interest, and to navigate to the resulting abstracted views of the system’s execution. Approaches that rely on static information can often rely on the iterative mapping approach, and their approach relied on dynamic information which limits iteratively updating the mapping. [15] proposed a complementary approach to Walker’s work that uses both static and dynamic information to answer developers questions about object oriented code. Their study focused on reverse engineering HotDraw and trying to understand it, but did not involve any code modification task.

**Previous evaluation of diagrams of the code structure.** Several studies have been conducted about the contribution of diagrams in program comprehension: [9, 11, 13]. [12], conducted a study on developers comprehension of UML diagrams. Their study focused on how developers use several types of UML diagrams for program comprehension. They found that developers often need all types of UML diagrams and integrate the information they get from each one to understand and analyze the program. They also found that developers even sort diagrams by the type of information they can get from them such as using sequence diagrams to understand the dynamic behavior and class diagrams to study static relations. These studies focused on UML diagrams, and none of them used the runtime structure extracted by statically analyzing the code to do a code modification task even though previous work ([12] and [9]) used partial runtime views such as sequence diagrams.
7 Conclusion

We conducted a longitudinal case study which serves as empirical evidence that developers do benefit from having access to diagrams of the run-time structure of a system, to answer some questions about object structure. The developer in this study is considered a pilot to enhance the study design. Future work includes conducting a user study with a more rigorous study design. In future studies we will overcome the issues identified in this study both in the tool and in the approach that could have interfered with the usefulness of the diagram of the run-time structure to developers.

8 Acknowledgements

The authors would like to thank Fayez Khazalah for his contributions to this study. We would also like to thank Thomas LaToza for useful feedback on this paper and on future study designs. Special thanks to the HAoSE’10 workshop reviewers for their valuable advice.

References


