A Case Study in Adding Ownership Domain Annotations

Marwan Abi-Antoun   **Nariman Ammar**
Fayez Khazalah

October 2010

Department of Computer Science
Wayne State University
Detroit, MI 48202

Abstract

A software system can be described using different architectural views including code views and run-time views. A code view describes class dependencies at compile time, whereas a run-time view describes the interactions between the different objects at runtime, i.e., the run-time structure. We conducted a case study in adding ownership domain annotations to a real, object-oriented Java system. We used static analysis to type check the added annotations, and to extract diagrams of the run-time structure of the system. We describe, using examples from the actual system, how the code was incrementally annotated to extract diagrams of the run-time structure. Such diagrams can be useful for developers by making explicit the communication between objects across run-time tiers. We also describe how the annotation-based approach can be used to refine the extracted diagrams to reflect the developers mental model of the system.

Contents

List of Figures

1 Introduction

Architectural views, like code views and run-time views, are used to describe a software system. A run-time view describes the interactions between the different objects at run-time. Understanding these object interactions is essential for program comprehension and can help developers gain a better understanding of the system structure while doing code modifications. We use a previously implemented ownership type system [2] to annotate an object oriented code and use static analysis to extract diagrams that depict the run-time structure of the system for the use of developers doing code modifications.

In this report, we present a case study in adding ownership domain annotations to an object-oriented Java system to extract diagrams of the run-time structure of the system. We refined the extracted diagrams by refining the added annotations to get a diagram that can be useful for a developer doing a code modification task on the system. The report includes results from a preliminary study on a developer doing a code modification using the extracted diagram. We illustrate how the ownership domain annotations help get a diagram that expresses logical containment, strict encapsulation and architectural tiers and makes explicit the communication between objects across different run-time tiers. We also discuss the refinement of the extracted OOG and the developers feedback to get a diagram which reflects the developers mental model of the system.

The report is organized as follows. Section 2 gives an overview on the annotation-based approach. Section 3 discusses the study setup and methodology. Section 4 discusses the results. Finally, we discuss some limitations in Section 5 and conclude.

2 Background

In this section we briefly introduce the ownership domains type system and the annotation-based static analysis approach to extract the run-time structure. Abi-Antoun and Aldrich previously proposed an annotation-based static analysis approach [2] where architectural extractors use ownership types to statically extract a hierarchical Ownership Object Graph (OOG) of an object-oriented program. The OOG is a diagram that depicts the run-time structure of the system, by tracking instances rather than types in the code. To extract the OOG, the architectural extractor adds annotations to every object reference in the code to 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.

2.1 The Ownership Object Graph (OOG)

The Ownership Object Graph (OOG) is a hierarchical object graph that is composed of nodes and edges that show the interactions between these nodes. A node can be either an object, represented by a box, or a domain within an object, represented by a white box with a dashed border. Each object node has a unique parent domain, and each domain node has a unique parent object. Edges between objects, represented by solid arrows, correspond to points-to relations that show how these objects are related. Edges on the OOG can be traced to field declarations in the code.

The OOG provides two types of abstraction that make it scale. The first form of abstraction is by ownership hierarchy. To get a more abstract OOG, the architectural extractor can collapse many objects into one. This is illustrated in the upper part of figure 14, where the plus symbol indicates that `panel:SimpleModelPanel` object has a nested object `commandPanel:Panel` as part of its sub-structure. Nested objects often reside in ownership domains which could be either public or private. The other form of abstraction is by types, where the architectural extractor can use the notion of subtyping to specify the architecturally relevant types and use these types to merge related objects and get a less cluttered graph.

2.2 Ownership Domains

In an OOG, each object declares conceptual groups for its state, 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. The domains within an object express a sub-structure within this object, one that consists of other domains and objects that represent its parts. 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 accessed by any object which can reference the parent object. For example, figure 29 illustrates that the object `tmp:Vector` 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 represented by a white box with thin dashed border.

Next, we describe the annotation language used in the annotation-based approach, then we list the special annotations that are defined in the type system and how the objects with these special annotations are displayed on the extracted OOG.

2.3 The Annotation Language.

We summarize the types of annotations used in the annotation-based approach and we illustrate them with examples from the annotated code in Figure 1.

- `@Domains('domain-name')`: this annotation declares domains. For example the system's `Main` class declares two top-level domains: `MODEL` and `UI`.
- `@DomainParams`: This annotation declares formal domain parameters on a type. Domain parameters are necessary to provide the ability to access inner objects across different domains, so different objects can share each others objects. For example, `SimpleModePanel` class declares the formal domain parameter `M` since instances of this type are part of the `VIEW` domain and need to have access in the `MODEL` domain, whereas `SingleDrawingModel` class, whose instances are in the `MODEL` domain, declares the formal domain parameter `V`. When the architectural extractor encounters an instance of these classes, he needs to bind the parameters on this instance to the parameters of the declaring type using the `Domain` annotation.
- `@DomainInherits`: this annotation specifies parameters for supertypes by binding the current type's formal parameters to the parameters of supertypes. For example, in Figure 1 `SimpleModelPanel` inherits from `SimplePanel`, so it binds its domain parameter `M` to the `SimplePanel` domain parameter `M` using the annotation `@DomainInherits("SimplePanel<M>")`.
- `@Domain(''domain-name'')`: by adding this annotation to an object declaration we place the object inside this domain. This annotation can also be used to bind an actual domain to a formal domain parameter. For example, the `Main` class declares a `SimpleModelPanel` instance `model`, then binds the `MODEL` domain to the `M` domain parameter by adding the `Domain("UI<MODEL>")` annotation. Finally, this annotation can optionally be used to annotate array parameters (e.g., handles[] in Figure 1).
- `@DomainReceiver(''Domain-name'')`: The architectural extractor uses this annotation to declare the receiver of a constructor or a method (Figure 2)

Second layer of annotations. The ownership type system has a second layer of annotations that are used to enforce architectural constraints and these include:

- **@Domainlinks**: declares domain link specifications to give objects in one domain permission to access objects in the private domains of other objects. Figure 1 illustrates that by declaring the `@DomainLinks(''UI->MODEL'')` annotation on the `Main` class which gives permission for objects in the UI domain to access objects in the `Model` domain.
- **@DomainAssumes:** declares domain link assumptions. For example, in Figure 1, `SingleDrawingModel` assumes that the owner of the model has access to the `V` domain which has

instances of the `VIEW` tier such as `SimpleModelPanel` instances.

Domain names. For domain names, the architectural extractor can choose arbitrary strings such as `MODEL`, `UI`, and `SUBS`. He can also use the special alias types which are special domains that are already implemented in the ownership type system and need not be declared. These domains provide the architectural extractor with the ability to control aliasing between different objects in the system, and include the following:

- **The `lent` domain:** by annotating an object with the lent annotation, the architectural extractor declares a temporary alias of this object within the current method. This annotation provides the capability to share objects in a time-bounded way. The architectural extractor can use this annotation on objects that are passed as method parameters which gives that method access to the object for the duration of the call without storing a persistent reference to the object.
- **The `unique` domain:** the architectural extractor can use the `unique` annotation on unaliased objects, such as newly created objects(e.g., `rectangle` in Figure 3), or those objects passed linearly from one domain to another(e.g., `clipboard` in Figure 3). Unique is considered the best annotation for unshared objects in an object-oriented system.
- **The `shared` domain:** the architectural extractor can use this annotation to indicate that certain objects are shared globally across the system. The shared 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, but the architectural extractor can use it to annotate immutable objects like "String" objects or architecturally non-significant objects (Figure 24).

  The object graph analysis assumes that all objects marked as shared are in one domain. As a result, due to merging 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 (Figure5). Displaying the shared domain would be trivial, but would add many uninteresting edges to the OOG.
- **The `owner` domain:** the ownership domain type system implements the owner domain, and adds it to the list of domain parameters declared on each class (declaring type) in the program. The `owner` parameter always occurs as the first element in the list of domain parameters. Figure 4 illustrates how the architectural extractor can use the `owner` implicit parameter to make the annotations less verbose. For example, the `SimplePanel` type explicitly declares the `M` domain parameter and uses the implicit domain parameter `owner` instead of explicitly declaring the `V` domain parameter.

3 Case Study

3.1 The subject system

For the study, we selected the DrawLets subject system [3] version 2.0 (115 classes, 23 interfaces, 12 packages). DrawLets is an object-oriented framework implemented in the Java language for building graphical applications. DrawLets supports a drawing canvas that holds figures and lets users interact with them. The figures include lines, freehand lines, rectangles, rounded rectangles, triangles, pentagons, polygons, ellipses, and text boxes. Tools are InputEventHandlers that act on drawing canvases and modify the figure attributes, such as size and location. They implement the CanvasTool interface. SimpleDrawingCanvas adds Figures to a Drawing and the SimpleDrawingCanvas implements the DrawingCanvas interface.

3.2 Tools and Instrumentation

The architectural extractor used Eclipse IDE version 3.5.1 with the type checking tool, ArchCheckJ, installed as a plug-in to be able to add the annotations to the subject system. He also used the architectural extraction tool, ArchRecJ, which he also installed as a plug-in to extract OOGs from the annotated code.

The Defaulting Tool. The defaulting tool is a separate tool that was implemented in Eclipse to reduce the annotation burden on architectural extractors.

The Type Checker. The type checker, ArchCheckJ, implements the ownership domains type system, using Java 1.5 annotations and the Eclipse infrastructure. The tool has additional features such as checking domain links and supports annotating external library code using XML files, AliasXML. The architectural extractor uses the annotation system to add annotations to the object-oriented Java code, then he runs the type checker to check for annotation warnings.

The OOG Extraction Tool. The extraction tool, ArchRecJ, is another static analysis tool that scans the annotated program's abstract syntax tree and produces the OOG. The extracted OOGs can then be given to the developer both in XML and PDF formats. The developer can load the XML file in another tool, OOG viewer, which visualizes the OOG for interactive usage.

3.3 The Architectural Extractor

The architectural extractor was responsible for adding annotations to the DrawLets code, running the static analysis to extract OOGs, as well as fine-tuning the extracted OOGs. The architectural extractor provided the developer with the diagrams both as XML files (to be loaded into the viewer) and PDF files (to be viewed or printed). The architectural extractor was one of the developers of the approach and the tools to extract run-time views from a system.

3.4 The Developer

The developer was a graduate student in the computer science program who was asked to provide a preliminary feedback on the extracted run-time structure.

3.5 Procedure

The architectural extractor first used the defaulting tool to add most of the annotations automatically to the code, then he modified the added annotations manually as needed. He then used the typechecker to type check the added annotations and fix warnings. Fixing the warnings often required him to refactor problematic code and generate XML files to be able to annotate external library code. The architectural extractor worked in iterations. During each iteration, he used the extraction tool to visualize the added annotations and make sure he got the desired diagram. He kept adding annotations to the system until he was able to get what he believed is the most useful diagram for a developer doing a code modification task on DrawLets (Fig 30). After that, he gave the extracted run-time structure to the developer to provide initial feedback. The architectural extractor used the developer's feedback to refine the extracted run-time structure as discussed in the following section.

4 Results

4.1 The Annotation Process

In the following subsections, we discuss the annotation process and the extraction and refinement of the OOGs illustrated by actual examples from DrawLets. We discuss how the defaulting tool adds default annotations to the code and how ownership domains can express and enforce the design intent related to object encapsulation and communication using code snippets from the subject system.

4.1.1 Applying the Defaulting Tool

The architectural extractor applied the defaulting tool to the subject system to reduce the effort of adding annotations to the code manually. The defaulting tool annotates local variables, temporary variables of methods, and the formal parameters of methods with `lent`. It also annotates private fields and return types of private methods with `owned`. Variables of type `String` are annotated with `shared`.

6

Figure 6 shows a snippet of the DrawLets code after applying the defaulting tool. The formal parameter of `changedShape()` method, `oldBounds`, is annotated using `lent`. The temporary and local variables `newBounds` and `evt` are also annotated using `lent`.

In figure 7, the protected field, `canvas`, and the return type of the `addButton()` method are annotated with `owned`. The formal parameter `label` is annotated with `shared` since it is of type `String`. The annotations added by the defaulting tool are not necessarily correct and may need to be modified later during the annotation process. Also, while the architectural extractor annotated the system, he found that some already annotated references are not annotated correctly, and thus he modified their annotations and continued the process of adding annotations until all the references were annotated correctly. This reflects the iterative nature of the annotation-based approach.

4.1.2   Selecting the Top-Level Domains

Many applications can be organized into three tiers: User Interface, Logical and Data tiers. Many applications that follow the MVC architecture can be optimized using the Document-View architecture. For simplicity, we organized the core types in DrawLets into two top-level tiers: the `Model` tier and the `UI` tier(Fig. 30). The figure shows that each tier is represented as an ownership domain or group. Initially, the intent was to allow objects in the `UI` tier have permissions to access objects in the `Model` tier by declaring a domain link from the `UI` domain to the `Model` domain, but not the other way around which turned out not be the case in DrawLets were we found some communication violations.

The architectural extractor tried to use the code structure to get a general idea about how to organize the core types in DrawLets into run-time tiers. He noticed from looking at the type hierarchy of `SimpleModelPanel` 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. Figure 30 illustrates how the architectural extractor organized the core types in DrawLets. The `Model` tier consists of `Drawing`, `DrawingCanvas`, `Figure`, etc. A `Drawing` is composed of `Figures`. A `DrawingCanvas` has a list of `Handle`s to allow user interactions.

The `UI` tier consists of `ToolPalette`, `StylePalette`, `ToolBar`, etc. Once we define the two top-level ownership domains, `Model` and `UI`, we pass the corresponding domain parameters `M` and `V` to the various types as discussed in the following sections.

4.1.3   Annotating the Root Class

The architectural extractor started adding annotations to the subject system by annotating the top-level class, `Main` by annotating every object reference inside this class (Fig 8). In line 1, he declared the top-level domain, `UI`. The annotation `@Domain("UI")` in line 4 declares the reference `panel` of type `SimpleModelPanel` to be in the `UI` domain. Notice that he added the `<MODEL>` domain to the annotation since we need to bind a certain value to the formal domain parameter `M` declared on `SimpleModelPanel`. The domain parameter on `SimpleModelPanel` was declared initially on `SimplePanel` to hold `DrawingCanvas` objects. Notice that `SimpleModelPanel` extends `SimplePanel` and this explains the use of the `DomainInherits` annotation. Finally, in the `Main` class we bind the `SimpleModelPanel::M` domain parameter to `Main::MODEL` and this explains why the `DrawingCanvas` objects now appear in the `MODEL` domain on the extracted OOG.

The architectural extractor ran the extraction tool to get the OOG in Figure 9. The figure shows a partial OOG with the dashed box labeled with `lent` to indicate a global domain of type `lent`. This 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`.

4.1.4   Binding Domain Parameters

After declaring domains and placing objects in those domains, the architectural extractor had to add the necessary domain parameters to allow these objects to share other objects across domains. The need for domain parameters propagates the annotation warnings introduced by the type checker which requires the

architectural extractor to bind the correct number of domain parameters, and in the same order of declaration on the declaring type (Figure 10), to resolve these warnings. In general this process of adding annotations and binding domain parameters is iterative. This also means that there are different ways to annotate the system and thats why you can get different extracted OOGs.

The fully annotated class `Main` appears in figure 11. We added the domain `MODEL` in line 1. We know from the top-level class that objects of type `SimpleModelPanel` are in the `UI` domain. The field `model` inside `SimpleModelPanel` class in line 23 is a reference to an object of type `SingleDrawingModel`. We know that `SingleDrawingModel` object should be declared inside `MODEL` domain, so we declared a domain parameter `M` in line 19.

Back to line 23, we annotated the reference `model` with domain parameter `M` to declare it inside `MODEL` domain. We assigned the owner of the reference to be the enclosing domain by putting `<owner>` after the domain parameter. To map the domain parameter `M` to the `MODEL` domain and thus specify the direction of communication between objects in the different tiers, we added the parameter `<MODEL>` to the annotation in line 5. Figure 8 shows the extracted OOG after applying these annotations. We can see that we have two tiers or domains, `UI` and `MODEL`. Our goal is to organize all objects in these two containers. Objects in domain `UI` can refer to objects domain `MODEL`, but not vice versa.

The architectural extractor wanted to add more objects to the diagram, so changed the default annotations on `SimpleModelPanel` and `SimplePanel` types (Figure 13). The figure now shows that `SimpleModelPanel` class has two references to objects. The first one, `model`, is a reference to `SingleDrawingModel` which belongs to the `MODEL` domain. The second reference, `commandPanel`, is a reference to a `Panel` object. Notice that both `model` and `commandPanel` are protected fields witin `SimpleModelPanel` class. However, using the ownership domain annotations the architectural extractor can make instances of `commandPanel` strictly encapsulated within the substructure of `SimpleModelPanel` instances. On the other hand, the `model` instance is not owned by `SimpleModelPanel` instances since it is part of the `MODEL` domain, and thats why you should declare the `M` domain parameter to hold instances of the `MODEL` domain.

`Panel` object is part of the `UI` domain, so the annotation `@Domain(''owned")` puts objects of type `Panel` inside the `private` domain of `SimpleModelPanel` instance. Since `SimpleModelPanel` class extends `SimplePanel` class, any instance of the superclass `SimplePanel` will be part of the structure of `SimplePanel`.

Figure 13 also shows that `SimplePanel` has four fields referencing different types of objects. The first one is a reference to `DrawingCanvas` object. `DrawingCanvas` is part of domain `MODEL`, so we annotated it with domain parameter `M`. The other three references are owned by the `SimplePanel` object, so we annotated them with `@Domain(''owner")`. After applying these modifications to the code, we got the OOG in Figure 14.

4.1.5 Type checking the added annotations

When the architectural extractor encounters an object for the first time, he uses the `Domain` annotation to place this object in an ownership domain. Any other future reference to this object in the code is annotated with the same domain name or any other non-conflicting domain name. For example, when dealing with method declarations the architectural extractor should take into consideration the following: method return types and parameters should be annotated only if they are reference types, modifiers go hand in hand with ownership domain annotations so method parameters and return types should never be marked owned for public methods, in the case of method overriding the architectural extractor cannot change the annotations on return types, parameters, or receivers, and every reference to that object in the program should be annotated consistently. After the architectural extractor adds annotations to the system, he runs the typechecker to check that the annotations are consistent with each other and with the code. The architectural extractor then addresses the resulting annotation warnings.

The architectural extractor fixed the warnings produced by the type checker in the following order: undeclared domains, wrong number of parameters inheritance rules, class instance creation expressions, local variable declarations, assignment rules, storing lent in a field, array parameters container domain parameters, and AliasXML files. For example, the type checker assumes that a variable with any type annotation can be

assigned a `unique` value, while `lent` variables can be assigned a value with any type annotation. This also applies to user declared domains, for example, a `V` domain may not be assigned to `M` domain.

The architectural extractor was able to annotate most of the code with only 56 remaining warnings. The warnings can be classified as follows:

Wrong number of array parameters. Figure 16 illustrates a wrong number of array parameters in the annotation added to `oldHandles[]` which results in the warning:

> "Expected array alias parameters, Wrong number of alias parameters.".

This could be a false positive in the type checker which we need to fix.

Storing a "lent" variable in a field. Figure 15 illustrates that in `FigureTransfer` class, the `figures` field which is annotated with "owned" is being assigned to the `figures` method parameter which obviously has the "lent" annotation cannot be assigned to each other. This results in the following warning:

> "Alias annotation `owned` does not match expected annotation lent at assignment `this.figures=figures`."

Notice that to fix this warning the architectural extractor has to create a copy of lent `figures` argument since it cannot hold on to it.

4.1.6   Annotating External Library Code

To fix some warnings, the architectural extractor annotated external library code using AliasXML files. Figure 17 shows two examples for references that required their annotation to be saved in AliasXML files. We also show two portions from different XML files that contain annotations for Java objects in figure 18.

4.1.7   Visualizing the added Annotations

The purpose of the study was to extract a run-time structure that can be useful for developers doing a code modification on DrawLets. A useful diagram means a diagram that is hierarchical, conveys 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 a diagram that reflects the design intent and show the system as-is, including any implemented design patterns in the code. Often times, the architectural extractor refined the annotations to achieve architectural abstraction either by ownership hierarchy (by pushing implementation details such as data structures underneath application domain objects) or by types.

After completing the annotations process, the architectural extractor was able to extract, what he considered to be a useful diagram of the run-time structure for developers (Fig 19). The architectural extractor considered this diagram to be useful since it expressed the run-time tiers and made the communication between the core objects explicit. However, he wanted to give this diagram to a developer to investigate whether the extracted diagram is indeed useful for developers and whether it needed any further refinement.

```
1  // Main class is used as the entry point
2  @Domains( { "MODEL", "UI" })
3  @DomainLinks({"UI->MODEL"})
4  public class Main {
5      @Domain("UI<MODEL>")SimpleModelPanel panel = new SimpleModelPanel();
6  ...
7  }
8  //SimplePanel
9  @DomainParams({"M"})
10 public class SimplePanel extends Panel {
11 ...
12 }
13 // SimpleModelPanel
14 @Domains({"owned"})
15 @DomainParams({"M"})
16 @DomainInherits({"SimplePanel<M>"})
17 @DomainAssumes({"owner->M"})
18 public class SimpleModelPanel extends SimplePanel implements ActionListener {
19     protected final @Domain("owner<M>") SingleDrawingModel model = new SingleDrawingModel();
20 ...
21 }
22 // SingleDrawingModel
23 @Domains({"owned","SUBS"})
24 @DomainParams({"V"})
25 @DomainAssumes({"owner->V"})
26 @DomainLinks({"SUBS->owned","SUBS->V"})
27 public class SingleDrawingModel extends BasicObservable {
28     @Domain("owner<V>") Drawing drawing = new SimpleDrawing();
29     ...
30     }
31 // SimpleDrawingCanvas
32 @Domains({"owned"})
33 @DomainParams({"V"})
34 @DomainInherits({"DrawingCanvas<V>","ComponentHolder<V>"})
35 public class SimpleDrawingCanvas extends AbstractPaintable implements DrawingCanvas,...{
36  @Domain("lent[owner<V>]") Handle handles[];
37  ...
38  }
```

Figure 1: Examples on the annotation language from `Main`, `SimpleModelPanel`, and `SingleDrawingModel` classes.

```
 1 @DomainReceiver("owner")
 2 public DrawingCanvasComponent(@Domain("M<owner>") DrawingCanvas canvas) {
 3 super();
 4 this.setCanvas(canvas);
 5 if (canvas instanceof MouseListener)
 6 this.addMouseListener((MouseListener)canvas);
 7 if (canvas instanceof MouseMotionListener)
 8 this.addMouseMotionListener((MouseMotionListener)canvas);
 9 if (canvas instanceof KeyListener)
10 this.addKeyListener((KeyListener)canvas);
11 }
12 @DomainReceiver("owner")
13 protected void setCanvas(@Domain("M<owner>") DrawingCanvas canvas) {
14 if (canvas instanceof ComponentHolder)
15 ((ComponentHolder)canvas).setComponent(this);
16 this.canvas = canvas;
17 }
18 }
```

Figure 2: domain receiver is used to declare the receiver of a constructor or a method.

```
 1 //SimpleDrawingCanvas
 2 public class SimpleDrawingCanvas {
 3 protected static @Domain("unique") Clipboard clipboard = new Clipboard( "" );
 4 ...
 5 }
 6 //LocatorConnectionHandle
 7 public class LocatorConnectionHandle extends CanvasHandle{
 8 protected @Domain("unique") Rectangle getConnectionBounds()  {
 9 ...
10 @Domain("unique") Rectangle rectangle = new Rectangle();
11 return rectangle;
12 }
```

Figure 3: Example of using the unique annotation.

```
 1  //Without using owner
 2  @Domains({"owned"})
 3  @DomainParams({"M","V"})
 4  public class SimplePanel extends Panel {
 5  ...
 6  protected @Domain("M") DrawingCanvas canvas;
 7  protected @Domain("V") ToolPalette toolPalette;
 8  protected @Domain("V") StylePalette stylePalette;
 9  protected @Domain("V") ToolBar toolBar;
10  ...
11  }
12  //Using owner
13  @Domains({"owned"})
14  @DomainParams({"M"})
15  public class SimplePanel extends Panel {
16  ...
17  protected @Domain("M") DrawingCanvas canvas;
18  protected @Domain("owner") ToolPalette toolPalette;
19  protected @Domain("owner") StylePalette stylePalette;
20  protected @Domain("owner") ToolBar toolBar;
21  ...
22  }
```

Figure 4: The implicit `owner` domain parameter declared on `SimplePanel` class
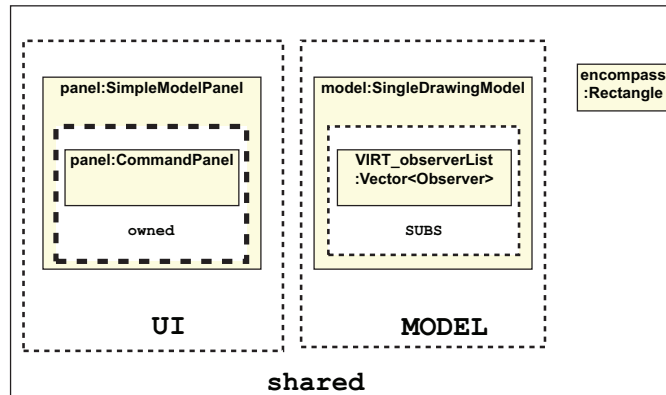
Figure 5: Objects inside the `shared` domain do not show on the extracted OOG.

```
 1 class AbstractFigure{
 2 ...
 3 protected void changedShape(@Domain("lent") Rectangle oldBounds) {
 4    @Domain("lent")
 5    Rectangle newBounds = getBounds();
 6    @Domain("lent")
 7    PropertyChangeEvent evt = new PropertyChangeEvent(this, SHAPE_PROPERTY,
 8                                                      oldBounds, newBounds);
 9    fireShapeChange(evt);
10    firePropertyChange(evt);
11 }
12 ...
13 }
```

Figure 6: Code automatically annotated with `lent` using the defaulting tool

```
 1 @Domains({"owned"})
 2 @DomainParams({})
 3 public abstract class CanvasPalette extends Panel implements ActionListener {
 4      protected @Domain("owned") DrawingCanvas canvas;
 5      ...
 6      protected @Domain("owned") Button addButton(@Domain("shared") String label) {
 7          @Domain("lent")
 8          Button button = new Button(label);
 9          button.addActionListener(this);
10          add(button);
11          return button;
12      }
13      ...
14 }
```

Figure 7: Default annotation for *private* fields is `owned`. *String* variables are annotated with `shared`

```
1  @Domains( { "UI" }) // declare the top level domains
2  public class Main {
3      @Domain("UI") SimpleModelPanel panel = new SimpleModelPanel();
4
5      public Main() {
6      }
7      public void run() {
8          try {
9              @Domain("lent")
10             java.awt.Frame frame = new ExitingFrame( "Simple Model Example" );
11             frame.add("Center", panel);
12             frame.setSize(panel.getSize());
13             frame.setVisible(true);
14         } catch (@Domain("lent") Throwable exception) {
15             System.err.println("Exception occurred in main() of TestCanvas");
16             exception.printStackTrace(System.out);
17         }
18     }
19     public static void main(@Domain("lent[shared]")String args[]) {
20         @Domain("lent")Main system = new Main();
21         system.run();
22     }
23 }
```

Figure 8: First attempt to annotate the root class
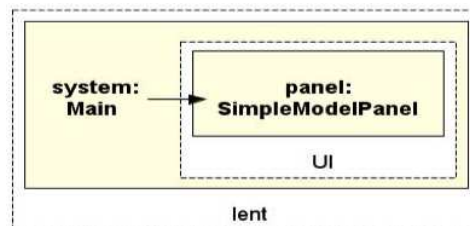
Figure 9: The extracted OOG of DrawLets after applying annotations added in figure 8

```
1  @Domains({"owned"})
2  @DomainParams({"M"})
3  public class SimplePanel extends Panel {
4
5      protected @Domain("M<owner>") DrawingCanvas canvas;
6      protected @Domain("owner<M>") ToolPalette toolPalette;
7      protected @Domain("owner<M>") StylePalette stylePalette;
8      protected @Domain("owner<M>") ToolBar toolBar;
```

Figure 10: Passing the correct number of domain parameters to address warnings.

```
1  @Domains({"MODEL", "UI"}) // Declare top−level domains
2    public class Main {
3      // Declare SimpleModelPanel reference in UI
4      // Bind domain parameter M to actual domain MODEL
5      @Domain("UI<MODEL>") SimpleModelPanel panel = new SimpleModelPanel();
6          public Main() {
7          }
8          ...
9      }
10     public static void main(@Domain("lent[shared]")String args[]) {
11         @Domain("lent") Main system = new Main();
12             system.run();
13     }
14   }
15
16   @Domains({"owned"}) // Declare private domain owned
17   @DomainParams({"M"}) // Declare domain parameter M
18   //@DomainInherits({"SimplePanel<M>"})
19   public class SimpleModelPanel extends SimplePanel implements ActionListener {
20     // Declare SimpleDrawingModel reference in M
21     protected @Domain("M<owner>") SingleDrawingModel model;
22   }
23   //@Domains({"owned"})
24   //@DomainParams({"V"})
25   public class SingleDrawingModel extends BasicObservable {
26     ...
27   }
```

Figure 11: Completing annotation of the top-level class and add domain parameters
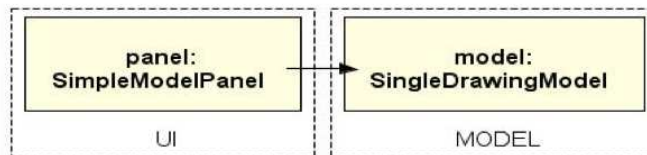
Figure 12: OOG extracted after applying annotations added in figure 11

```
 1 @Domains({"owned"})
 2 @DomainParams({"M"})
 3 @DomainInherits({"SimplePanel<M>"})
 4 public class SimpleModelPanel extends SimplePanel implements ActionListener {
 5     protected @Domain("M<owner>") SingleDrawingModel model;
 6     protected @Domain("owned") Panel commandPanel;
 7 }
 8
 9 @Domains({"owned"})
10 @DomainParams({"M"})
11 public class SimplePanel extends Panel {
12     protected @Domain("M") DrawingCanvas canvas;
13     protected @Domain("owner") ToolPalette toolPalette;
14     protected @Domain("owner") StylePalette stylePalette;
15     protected @Domain("owner") ToolBar toolBar;
16 }
```

Figure 13: Changing the default annotations on objects cause these objects to appear in different domains on the OOG
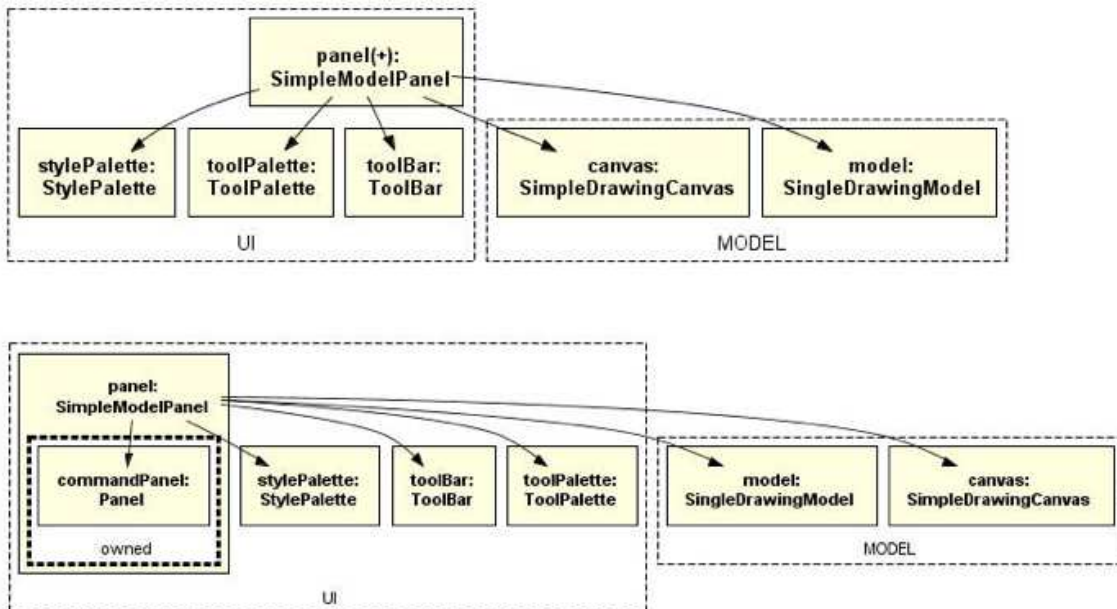

Figure 14: The extracted OOG after applying annotations added in figure 13

```
1  @Domains({"owned"})
2  @DomainParams({"V"})
3  @DomainAssumes({"owner->V"})
4  public class FigureTransfer implements ClipboardOwner, Transferable {
5
6      protected @Domain("owned<V>") Vector<Figure> figures;
7  ...
8  public FigureTransfer(@Domain("lent<owner<V>>") Vector<Figure> figures) {
9          this.figures = figures;
10     }
```

Figure 15: Storing a `lent` variable in a field produces annotation warnings.

```
1  public void addHandles(@Domain("owner<V>") Figure figure,
2                         @Domain("lent[owner<V>]") Handle handles[]) {
3  @Domain("lent[owner<V>]")
4  Handle oldHandles[] = figureHandles.get(figure);
5  ...
6  }
```

Figure 16: Annotating arrays by passing the correct number of array parameters

```
1  class BasicStringRenderer {
2  ...
3  protected @Domain("shared") FontMetrics metrics;
4  ...
5  public void setFont(@Domain("lent") Font font) {
6  if (font != getFont()) {
7      metrics = Toolkit.getDefaultToolkit().getFontMetrics(font);
8      reset();
9      }
10 }
11 }
```

Figure 17: Annotating external library code using AliasXML.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <class id="Ljava/awt/Toolkit;" name="java.awt.Toolkit">
3  ...
4  <method id="..." name="getFontMetrics">
5  <param domain="" id="Ljava/awt/Font;" name="Font" paramActuals="" paramArrays="" />
6  <return domain="shared" id="..." paramActuals="" paramArrays="" type="java.awt.FontMetrics"/>
7  <receiver domain="" paramActuals="" paramArrays="" />
8  </method>
9  ...
10 </class>
```

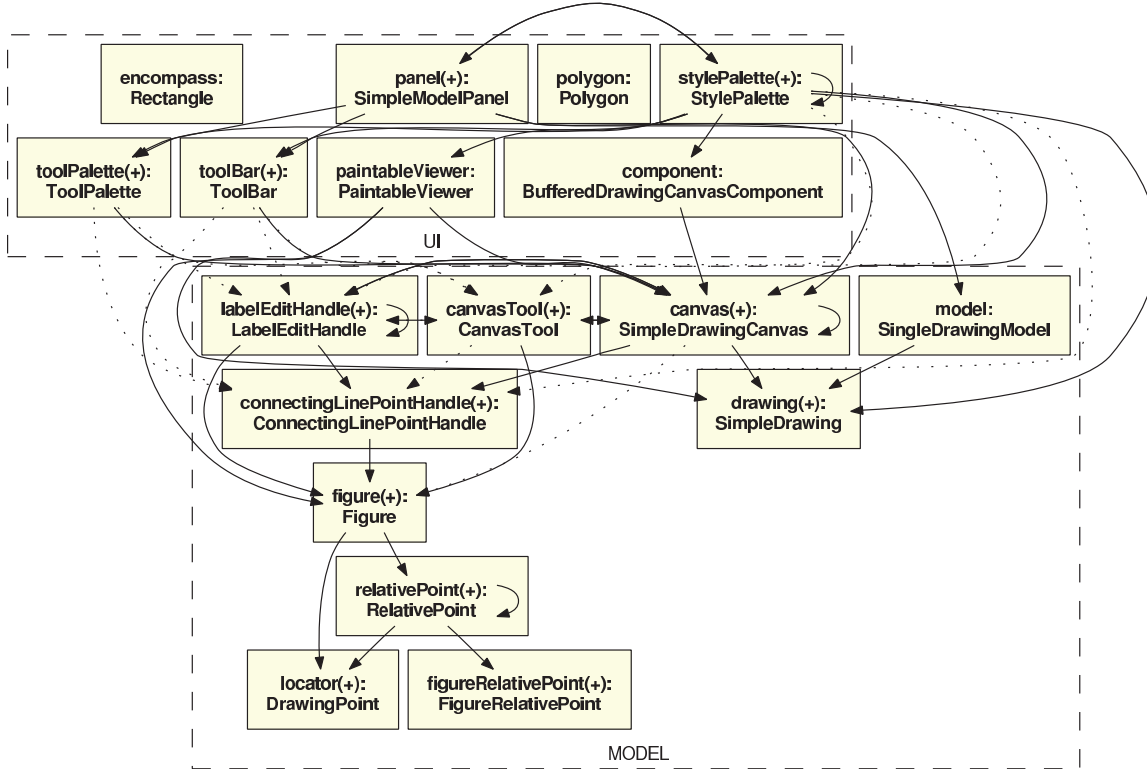Figure 18: Part of the annotations added to java.awt.Toolkit.xml file.

Figure 19: A version of the extracted OOG with non-relevant objects showing in the top-level domains.

4.2 Modifying the annotations to refine the extracted OOG

To confirm that the extracted OOG was useful to developers, the architectural extractor asked a developer to use the diagram and report how much useful she found this diagram to be. In this section we report the developers feedback and the refinement of the OOG.

To reflect the developers mental model of the system, the architectural extractor was manipulating the graph to improve the quality of the visualization either by refining the annotations on some of the already annotated references in the code, or by using certain features in the extraction tool itself. In the following sections we highlight more of the cases where the architectural extractor used the annotations and the extraction tool to get an OOG that reflects the developer's mental model of the system.

4.2.1 Collapsing 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`. Annotating objects that are of the same type and in the same domain with `owner` causes these objects to be merged. Same type can mean same declared type or subtype thereof compatible types. Figure 4 illustrates how the architectural extractor annotated `SimplePanel`, `StylePalette`, and `ToolPalette` instances with `owner` which caused all these objects to be treated as siblings to each other and appear in the same enclosing domain, `UI` (see Figure 12).

The OOG extraction tool has certain features that enable the architectural extractor refine the OOG even more. During this study the architectural extractor used the feature to abstract object by types to control the excessive merging between objects. He also used the feature to add additional labeling types to add the type decoration to an object's label, if it merges at least one object of that type.

```
1 class SimpleDrawing {
2       ...
3      @Domain("shared")
4      public Rectangle getBounds() {
5          if ( isDynamicSize() ) {
6              @Domain("shared")
7              Rectangle encompass = new Rectangle(0,0,0,0);
8              for (@Domain("lent<V>") FigureEnumeration e = figures() ;
9              e.hasMoreElements() ;) {
10                 encompass.add(e.nextElement().getBounds());
11             }
12             return encompass;
13         }
14         @Domain("shared") Rectangle rectangle = new Rectangle( 0, 0, width, height );
15         return rectangle;
16     }
```

Figure 20: Annotating encompass with "shared"

```
1 class AbstractShape{
2 ...
3      public static @Domain("shared") Polygon reshapedPolygon(@Domain("lent") Polygon polygon,
4      int x, int y, int width, int height) {
5          int npoints = polygon.npoints;
6          @Domain("lent")int xpoints[] = new int[npoints];
7          @Domain("lent")int ypoints[] = new int[npoints];
8          @Domain("lent")
9          Rectangle polyBounds = polygon.getBounds();
10         double xScale = (double)width / (double)polyBounds.width;
11         double yScale = (double)height / (double)polyBounds.height;
12         for (int i=0; i < npoints; i++) {
13             xpoints[i] = x + (int)((polygon.xpoints[i] - polyBounds.x) * xScale);
14             ypoints[i] = y + (int)((polygon.ypoints[i] - polyBounds.y) * yScale);
15         }
16         @Domain("shared") Polygon polygon2 = new Polygon(xpoints, ypoints, npoints);
17         return polygon2;
18     }
19 ...
20 }
```

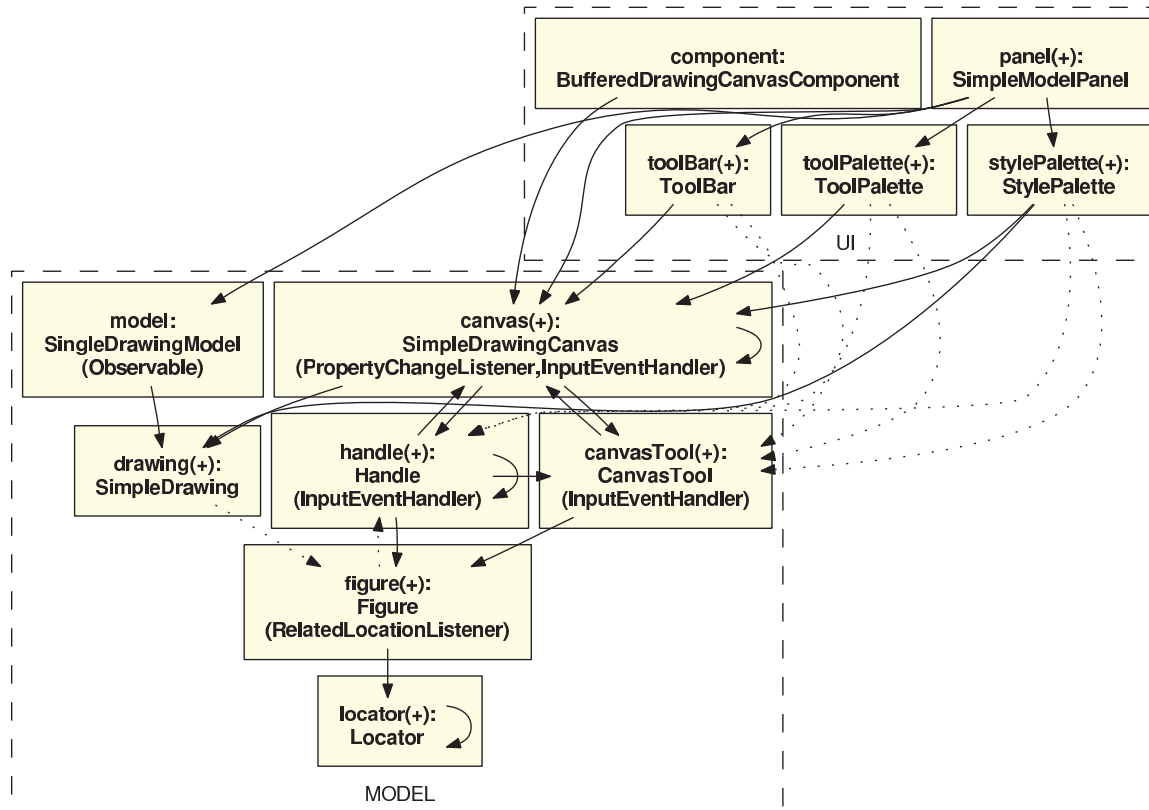Figure 21: Annotating polygon with "shared"

Figure 22: The extracted OOG after applying annotations added in figure 20 and figure 21

To improve the diagram in Figure 12 and get a more abstract diagram, the architectural extractor looked at the type hierarchy for the `Panel` interface, which shows that all these objects has types that implement ultimately the `Panel` interface. The architectural extractor chose the type `Panel` to represent the three objects as one object `panel:Panel` on the extracted OOG. However, the developer 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 reduce the amount of object merging, and have the diagram display those objects as separate boxes.

4.2.2   Manipulating the object hierarchy

The developer thought that the diagram in figure 19 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. The shared domain is a global invisible domain, so any objects within this domain do not appear on the extracted diagram(Fig 5).

4.2.3   Adding labeling types

When the architectural extractor extracts the ownership object graph, the 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. The architectural extractor specifies an optional list of labeling types for labelling objects. The tool then adds the type decoration to an object label if it merges at least one object of that

```
1 @Domains({"owned"})
2 @DomainParams({"M"})
3 @DomainInherits({"CanvasPalette<M>"})
4 public class ToolPalette extends CanvasPalette {
5 protected @Domain("owned<M>") Vector<InputEventHandler> tools = new Vector<InputEventHandler>(5);
6 protected @Domain("owned") Component lastButton;
7 ...
8     }
```

Figure 23: The list of `tools` is strictly encapsulated by instances of `ToolPalette`.

```
1 @Domains({"owned"})
2 @DomainParams({"M"})
3 @DomainInherits({"CanvasPalette<M>"})
4 public class StylePalette extends CanvasPalette {
5
6 protected @Domain("owned<shared>") Vector<Color> colors = new Vector<Color>(5);
7 protected @Domain("shared") Color color = Color.black;
8 protected @Domain("owned<shared>") Vector<Method> setters = new Vector<Method>(5);
```

Figure 24: Color and Method are not are not architecturally significant, so they are moved to the `shared` domain.

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. 29.

4.2.4   Adding design intent types

The architectural extractor studied the core interfaces in the DrawLets framework and used them to add more design intent types, such as `Handle` and `Locator`, to the extracted OOG. To add instances of `Locator` and `Handle`, the architectural extractor added annotations to the different instances in the code which caused these instances to appear on the extracted OOG (Figure 19).

4.2.5   Controlling Object Merging

The architectural extractor wanted to get a more abstract diagram than the one in Figure 19 by reducing the clutter in the top level domains, so he fine tuned the annotations on locator and handle instances (Fig 27) which caused the object types to be merged under their super types. Object merging means that two or more objects of the same type (i.e. same declared type or subtype thereof compatible types) that are in the same domain will be collapsed as one object on the extracted OOG. Figure 26, for example, shows the type hierarchy for the Locator interface, and figure 25 shows the merged objects of these types under Locator. For example, the `relativePoint:RelativePoint`, `figureRelativePoint:FigureRelativePoint`, and `locator:DrawingPoint` were all merged under the `locator:Locator` object (Fig 25).

When object merging occurs, the architectural extractor can then use the abstraction by types feature in the extraction tool to control the amount of merging. The abstraction by types feature enables the architectural extractor to use the notion of subtyping to specify the architecturally relevant types.

The study identified two cases where object merging could be either useful as explained in this section or too abstract and cause confusion as described in section 4.2.1. The architectural extractor can always fine-tune the abstraction by types to reduce the excessive merging of objects, and have the diagram display those objects as either one box (e.g. `Locator` instance) or separate boxes(e.g. different instances of `Panel`).
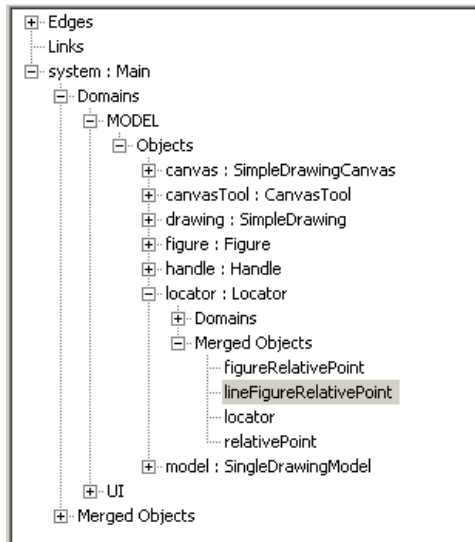
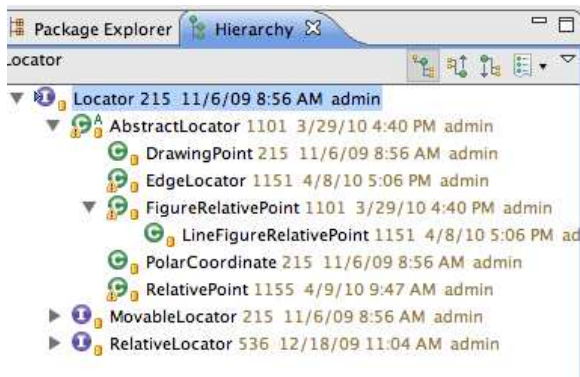Figure 25: Merged Objects inside locator

Figure 26: The Type Hierarchy of the Locator interface as it appears in Eclipse IDE.

4.2.6 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.

Field/Variable	Declared Type	Annotation	EnclosingType	AST Node
target	com.rolemodelsoft.drawlet.Figure	owner	LocatorConnectionHandle	target=canvas.otherFigureAt(figure,x,y)
locator	com.rolemodelsoft.drawlet.basics.DrawingPoint	owner	LocatorConnectionHandle	locator=new DrawingPoint(oldLocator.x(),oldLocator.y())
newLocator	com.rolemodelsoft.drawlet.Locator	owner	LocatorConnectionHandle	newLocator=target.requestConnection(figure,x,y)
relativePoint	com.rolemodelsoft.drawlet.basics.RelativePoint	owner	LocatorConnectionHandle	relativePoint=new RelativePoint(newLocator,oldLocator.x
locator	com.rolemodelsoft.drawlet.basics.DrawingPoint	owner	LocatorConnectionHandle	locator=new DrawingPoint(oldLocator.x(),oldLocator.y())
figure	com.rolemodelsoft.drawlet.Figure	owner	LocatorConnectionHandle	figure
locator	com.rolemodelsoft.drawlet.Locator	owner	LocatorConnectionHandle	locator
connection	com.rolemodelsoft.drawlet.Locator	owner	LocatorConnectionHandle	connection
figureRelativePoint	com.rolemodelsoft.drawlet.basics.FigureRelativePoint	owner	LocatorConnectionHandle	figureRelativePoint=new FigureRelativePoint(figure,0.0,0.
figureRelativePoint	com.rolemodelsoft.drawlet.basics.FigureRelativePoint	owner	AbstractFigure	figureRelativePoint=new FigureRelativePoint(this,relative
locator	com.rolemodelsoft.drawlet.Locator	owner	AbstractFigure	@Domain("owner") Locator locator
locator	com.rolemodelsoft.drawlet.Locator	owner	AbstractFigure	@Domain("owner") Locator locator
locator	com.rolemodelsoft.drawlet.Locator	owner	LineFigure	@Domain("owner") Locator locator
locator	com.rolemodelsoft.drawlet.Locator	owner	LineFigure	@Domain("owner") Locator locator

Figure 27: Annotating instances of the different types that share the type `Locator`.

```
1  //SimpleModelPanel
2  @Domains({"owned"})
3  @DomainParams({"M"})
4  @DomainInherits({"SimplePanel<M>"})
5  public class SimpleModelPanel extends SimplePanel implements ActionListener{
6    protected final @Domain("M<owner>") SingleDrawingModel model = new SingleDrawingModel();
7      ...
8     @Domain("model.SUBS<M<owner>,M<owner>>")
9     ValueAdapter adapter = new ValueAdapter(model, "getDrawing",canvas,"setDrawing");
10     ...
11 }
12 //SimplePanel
13 @Domains({"owned"})
14 @DomainParams({"M"})
15 public class SimplePanel extends Panel {
16     protected @Domain("M<owner>") DrawingCanvas canvas;
17 ...
18 }
19 //SingleDrawingModel
20 @Domains({"owned","SUBS"})
21 @DomainParams({"V"})
22 public class SingleDrawingModel extends BasicObservable {
23 ...
24 }
25 // ValueAdapter
26 @Domains({"owned"})
27 @DomainParams({"M","T"})
28 public class ValueAdapter implements Observer {
29     @Domain("M") Object model;
30     @Domain("T") Object target;
31     ...
32 }
```

Figure 28: Annotations added to `SimpleModelPanel`, `SingleDrawingModel`, and `ValueAdapter` to express the Observer design pattern.
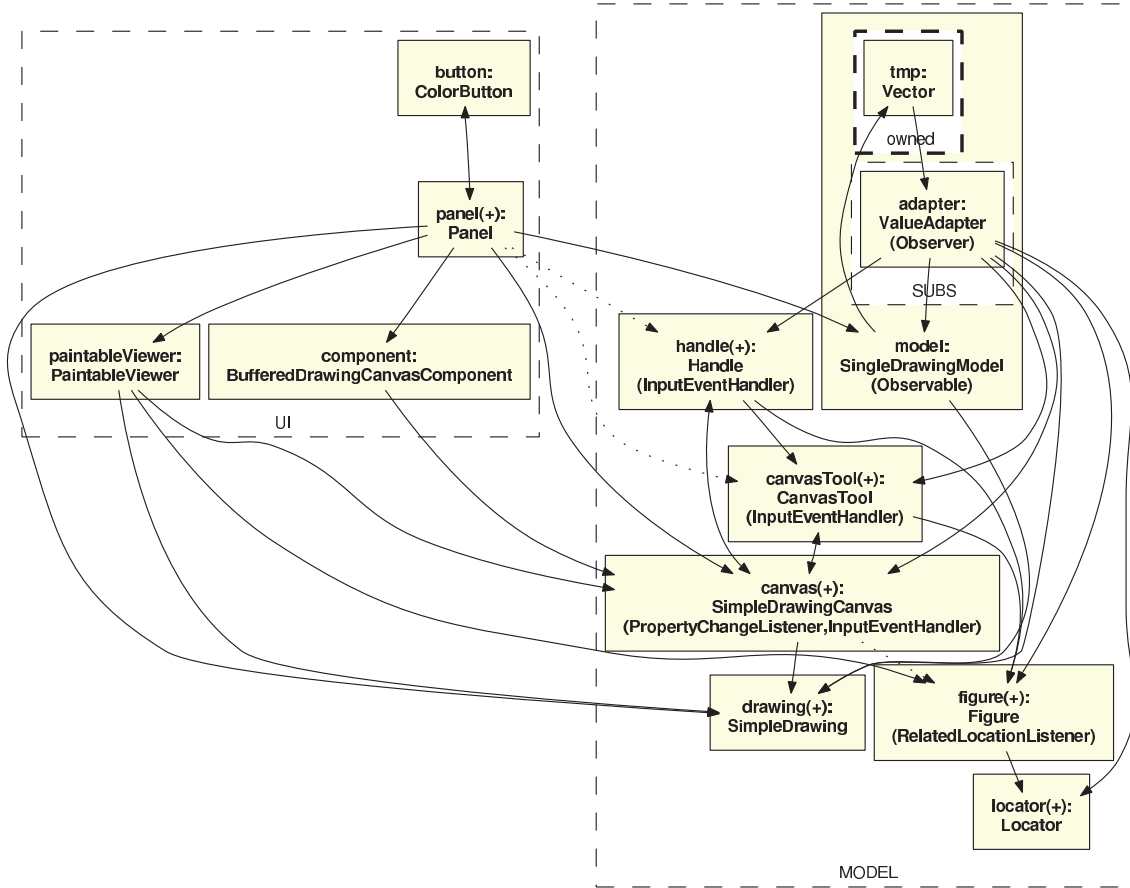
Figure 29: The extracted OOG after applying annotations added in figure 28.

4.3 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 such as the defaulting tool, the typechecker, and the extraction tool.

4.3.1 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 run-time exceptions. Moreover, the use of reflective code poses challenges for the ownership annotations and the static analysis (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). Therefore, through the process of annotating the subject system the architectural extractor had to do some changes to the code such as: extracting local variables, inferring generic types, to fix some problematic code and be able to add annotations easily. In the following sections we discuss some of the architectural extractor's efforts to refactor the code.

The architectural extractor tried to capture the design intent in DrawLets code by reading some informal comments in the code. Also, the using the ownership domain annotations helped him incrementally
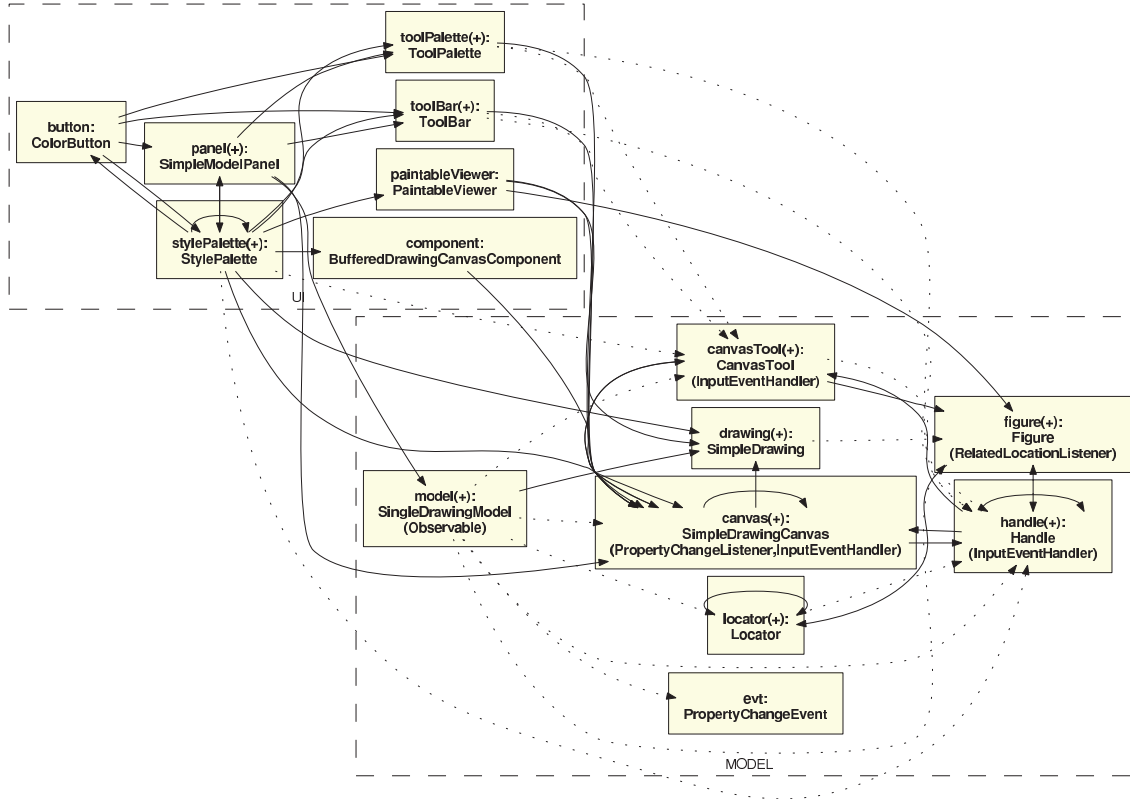
Figure 30: The Final version of the extracted OOG.

understand the code, so he refined the annotations accordingly. The annotations also helped the architectural extractor highlight some design vulnerabilities in the code which he believed needed major or minor refactoring which he would never realize without annotating the code.

This study identified that DrawLets does not strictly follow the Document-View architectural style. Notice that you can easily violate this style by accessing the `canvas` object inside `SimpleModelPanel` class, for example. This means that there is no need to go through `SingleDrawingModel` and that DrawLets does not strictly follow the two tier architectural style where objects in the `VIEW` tier should not have direct references to objects in the `MODEL` tier.

There were cases where the architectural extractor refactored the code to fix some problematic code patterns such as reflective code and loosely typed code. For example, the architectural extractor had to refactor some code to use generics (using Eclipse's refactoring tool). Some reflective code or loosely typed code was not fixed. He also tried to fix a few problematic expressions by extracting a local variable, and adding an annotation to the local variable. Then he added the annotations on the container elements, since he wanted to use the AliasXML files.

Annotating Listeners. The DrawLets code deals with listener interfaces from the Java standard library, such as `java.util.EventListener`. Figures 34 and 35 show annotations added to listener instances inside `AbstractFigure` and `ColorButton`.

There were a few cases where the annotation-based approach failed to reflect the developers mental model. Figure 30 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

```
1    @Domains({"owned"})
2    @DomainParams({"M","T"})
3    public class ValueAdapter implements Observer {
4    @Domain("M") Object model;
5    @Domain("shared") String aspect;
6    @Domain("T") Object target;
7    @Domain("shared") String affect;
8    @DomainReceiver("owner")
9    public ValueAdapter(@Domain("M") Observable model, @Domain("shared") String aspect,
10                       @Domain("T") Object target, @Domain("shared") String affect) {
11   this.model = model;
12   model.addObserver(this);
13   this.aspect = aspect;
14   this.target = target;
15   this.affect = affect;
16   }
17   public ValueAdapter(@Domain("M") Object model, @Domain("shared") String aspect,
18                       @Domain("T") Object target, @Domain("shared") String affect) {
19   this.model = model;
20   this.aspect = aspect;
21   this.target = target;
22   this.affect = affect;
23   }
24   protected @Domain("owned") Method getMessage(@Domain("lent") Class targetClass,
25               @Domain("shared") String message,
26               @Domain("lent[shared]") Class[] targetParameterClasses) {
27   try {
28   return targetClass.getMethod(affect, targetParameterClasses);
29   } catch (@Domain("lent") NoSuchMethodException e) {
30   return null;
31   }    }
32   ...
33   }
```

Figure 31: Example of a loosely typed code using "Object" and "Class"

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 (Figure 35).

Using public domains. When the architectural extractor wanted to annotate the code to express more clearly the observer design pattern, he had to refactor the code since 'model' has to be 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."

Inferring Generic Types. There were times when the architectural extractor refactored some code to use generics using the "infer generic types" refactoring in the Eclipse IDE. Figure 37 shows a snippet of code before and after refactoring. We see from the figure that the annotation of the refactored reference was also updated as required. In the annotation-based approach generic types are handled using nested parameters (e.g., `@Domain(''owned<owner<V>>'')` in figure 37).

4.3.2 Limitations in the current Tools

There are some limitations in the current tools that might add extra overhead on the architectural extractor or might make the run-time structure not as much useful to developers.

```java
 1  public class SimpleDrawingCanvas extends AbstractPaintable implements DrawingCanvas{
 2  ...
 3  // Before
 4      protected @Domain("shared") DrawingStyle defaultStyle () {
 5          return new SimpleDrawingStyle ();
 6      }
 7      ...
 8      @Domain("V")
 9      public Rectangle getBounds () {
10          if ( drawing.isDynamicSize () ) {
11              if ( width < component.getSize ().width ) width = component.getSize ().width;
12              if ( height < component.getSize ().height ) height = component.getSize ().height;
13          }
14          return new Rectangle ( 0, 0, width, height );
15      }
16  // After
17      protected @Domain("shared") DrawingStyle defaultStyle () {
18          @Domain("shared")SimpleDrawingStyle simpleDrawingStyle = new
19          SimpleDrawingStyle ();
20          return simpleDrawingStyle;
21      }
22      ...
23      @Domain("shared")
24      public Rectangle getBounds () {
25          if ( drawing.isDynamicSize () ) {
26              if ( width < component.getSize ().width ) width = component.getSize ().width;
27              if ( height < component.getSize ().height ) height = component.getSize ().height;
28          }
29          @Domain("shared")Rectangle rectangle = new Rectangle ( 0, 0, width, height );
30          return rectangle;
31          ...
32      }
```

Figure 32: Example of problematic code in `SimpleDrawingCanvas` returning a new expression.

Java 1.5 limitations. Java 1.5 allows adding annotations only to declarations. Because of this limitation, the architectural extractor could not annotate certain expressions such as new expressions and cast expressions without modifying the source code. The general way to deal with this limitation, as discussed previously [1], is to declare additional, temporary variables and add the appropriate annotations to the local variables. During the annotation of the subject system, the architectural extractor also faced many of the problematic new and cast expressions. He fixed these problematic expressions by extracting a local variable, and adding an annotation to the local variable. An example on a new expression is shown in figure 39. The code before the change is shown in the upper part of the figure. He re-wrote the code by extracting a local variable from the new expression and adding the appropriate annotation to it.

Figure 41 shows another area of the code where the architectural extractor needed to extract a local variable out of a cast expression. The *static* method `figureFromLocator()` determines whether or not a particular locator is tied to some figure. It takes aLocator as a parameter and returns the figure associated with the locator or `null` if none. The first part of the figure shows the code with two cast expressions. The second part shows the updated code after extracting local variables out of the cast expressions and then annotating these local variables with the appropriate annotations.

Declaring and binding method domain parameters. Java 1.5 annotations cannot be added at method invocation expressions. The architectural extractor had to use block comments to specify the actual domains

for a parameterized method. Figure 47 illustrates how the architectural extractor declared the `elems` domain parameter on `AWTEventMulticaster` class, and then used block comments to bind this parameter inside `ColorButton` class.

Figure 46 illustrates how listener objects inside `AWTEventMulticaster` are held in the `elems` domain parameter indicating that these objects are not owned by the Multicaster and can be referenced by external objects such as `colorButton`.

```
 1 @Domains({"owned"})
 2 @DomainParams({"V"})
 3 @DomainInherits({"DrawingCanvas<V>","ComponentHolder<V>"})
 4 public class SimpleDrawingCanvas extends AbstractPaintable implements DrawingCanvas,...{
 5 // Before
 6 protected @Domain("unique") Vector getFromClipboard(@Domain("lent") Clipboard clipboard){
 7  ...
 8     try {
 9         if (transfer.isDataFlavorSupported(FigureTransfer.figuresFlavor))
10                     return (Vector)transfer.getTransferData(FigureTransfer.figuresFlavor);
11     else
12         if (transfer.isDataFlavorSupported(DataFlavor.stringFlavor)) {
13             @Domain("shared")
14             String string = (String)transfer.getTransferData(DataFlavor.stringFlavor);
15             @Domain("lent<V>")
16             Figure label = getFigureFromString(string);
17             @Domain("unique")
18             Vector container = new Vector(1);
19             container.addElement(label);
20             return container;
21         }
22     ...
23     Toolkit.getDefaultToolkit().beep();
24     @Domain("unique") Vector<Object> vector = new Vector<Object>();
25     return vector;
26 }
27
28 // After
29 @Domains({"owned"})
30 @DomainParams({"V"})
31 @DomainInherits({"DrawingCanvas<V>","ComponentHolder<V>"})
32 public class SimpleDrawingCanvas extends AbstractPaintable implements DrawingCanvas,...{
33 protected @Domain("unique") Vector<Object> getFromClipboard(@Domain("lent") Clipboard clipboard){
34 ...
35 try {
36     if (transfer.isDataFlavorSupported(FigureTransfer.figuresFlavor))
37         return (Vector<Object>)transfer.getTransferData(FigureTransfer.figuresFlavor);
38     else
39      if (transfer.isDataFlavorSupported(DataFlavor.stringFlavor)) {
40         @Domain("shared")
41         String string = (String)transfer.getTransferData(DataFlavor.stringFlavor);
42         @Domain("lent<V>")
43         Figure label = getFigureFromString(string);
44         @Domain("unique")
45         Vector<Object> container = new Vector<Object>(1);
46         container.addElement(label);
47         return container;
48     }
49 ...
50 Toolkit.getDefaultToolkit().beep();
51 @Domain("unique") Vector<Object> vector = new Vector<Object>();
52 return vector;
53     }
```

Figure 33: Example of Problematic polymorphic code dealing with the clipboard.

```java
1  @Domains({"owned"})
2  @DomainParams({"V"})
3  @DomainInherits({"Figure<V>"})
4  public abstract class AbstractFigure extends AbstractPaintable implements Figure {
5  protected transient @Domain("owned<V>") Vector<PropertyChangeListener> listeners;
6  protected @Domain("owned<owner>") Vector<RelatedLocationListener> locationListeners;
7  public synchronized void addPropertyChangeListener(@Domain("lent") PropertyChangeListener listener
8  if (listeners == null) {
9      listeners = new Vector<PropertyChangeListener>();
10 }
11 if ( ! listeners.contains( listener ) ) {
12     listeners.addElement(listener);
13 }
14 }
15 public synchronized void addRelatedLocationListener(@Domain("lent") RelatedLocationListener listen
16 if (locationListeners == null) {
17     locationListeners = new Vector<RelatedLocationListener>();
18 }
19 if ( ! locationListeners.contains( listener ) ) {
20     locationListeners.addElement(listener);
21 }
22 protected void changedSize(@Domain("lent") Dimension oldDimension) {
23 @Domain("lent")
24 Dimension newDimension = getSize();
25 if (oldDimension != null && oldDimension.equals(newDimension)) {
26     return;
27 }
28 @Domain("owner")
29 PropertyChangeEvent evt = new PropertyChangeEvent(this,
30                     SIZE_PROPERTY, oldDimension, newDimension);
31 fireSizeChange(evt);
32 firePropertyChange(evt);
33 }
34 }
```

Figure 34: Annotating listeners in `AbstractFigure` class.

```
 1 @Domains({"owned"})
 2 @DomainParams({"M"})
 3 @DomainInherits({"CanvasPalette<M>"})
 4 public class StylePalette extends CanvasPalette {
 5
 6 @DomainReceiver("owner")
 7 public void addColor(@Domain("shared") Color color,
 8                      @Domain("shared") String label) {
 9 colors.addElement(color);
10 @Domain("owner")
11 ColorButton button = new ColorButton(label, color);
12 button.addActionListener(this);
13 ...
14 if ( ! areButtons ) {
15 currentColorButton = button;
16 button.setHighlighted( true );
17 areButtons = true;
18 }
19 }
20 //ColorButton
21 @Domains({"owned"})
22 public class ColorButton extends Component implements MouseListener {
23 protected @Domain("owner") ActionListener actionListener = null;
24
25 public void addActionListener( @Domain("owner") ActionListener listener ) {
26 actionListener = AWTEventMulticaster.add/*<owner>*/( actionListener, listener );
27 }
28 }
```

Figure 35: `currentColorButton` cannot be annotated with `owned`.
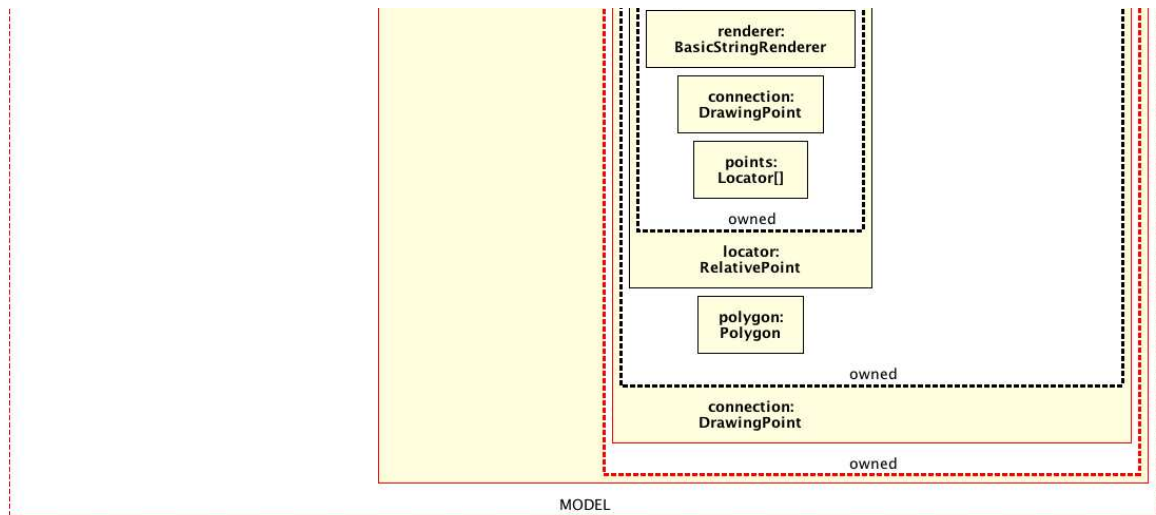

Figure 36: A version of the OOG before inferring generic types.

```
 1 public class SimpleDrawingCanvas extends AbstractPaintable implements DrawingCanvas,...{
 2 // Before
 3 protected @Domain("owned") Vector selections = defaultSelections();
 4 protected @Domain("owned") Vector handles = defaultHandles();
 5 protected @Domain("owned") Hashtable figureHandles = defaultFigureHandles();
 6
 7 protected @Domain("owned<owner>") Vector<Figure> defaultSelections() {
 8       return new Vector<Figure>();
 9        ...
10       }
11 public void moveSelectionsToBack() {
12 for (@Domain("shared") Enumeration e = new ReverseVectorEnumerator(selections); e.hasMoreElements
13      moveFigureToBack((Figure)e.nextElement());
14     }
15     ...
16 protected void paintHandles(@Domain("lent") Graphics g) {
17        ...
18 for (@Domain("shared") Enumeration e = handles.elements() ; e.hasMoreElements() ;) {
19      @Domain("lent<V>") Handle h = (Handle)e.nextElement();
20          if (h.intersects(clip))
21             h.paint(g);
22            } }
23   }
24 // After
25 protected @Domain("owned<owner<V>>") Vector<Figure> selections = defaultSelections();
26 protected @Domain("owned<owner<V>>") Vector<Handle> handles = defaultHandles();
27 protected @Domain("owned") Hashtable<Figure, Handle[]> figureHandles = defaultFigureHandles();
28
29 protected @Domain("owned<owner>") Vector<Figure> defaultSelections() {
30       return new Vector<Figure>();
31     }
32     ...
33 public void moveSelectionsToBack() {
34
35 for (@Domain("lent<owner<V>>") Enumeration<Figure> e = new ReverseVectorEnumerator<Figure>(selecti
36 {   @Domain("owner<V>") Figure figure = (Figure)e.nextElement();
37      moveFigureToBack(figure);
38        }}
39        ...
40 protected void paintHandles(@Domain("lent") Graphics g) {
41        ...
42 for (@Domain("lent<owner<V>>")Enumeration<Handle> e = handles.elements();e.hasMoreElements() ;) {
43    @Domain("lent<V>") Handle handle = (Handle)e.nextElement();
44    if (handle.intersects(clip))
45   handle.paint(g);
46              }}
47        ...
48     }
```

Figure 37: Inferring generic types in SimpleDrawingCanvas using Eclipse refactoring tool.
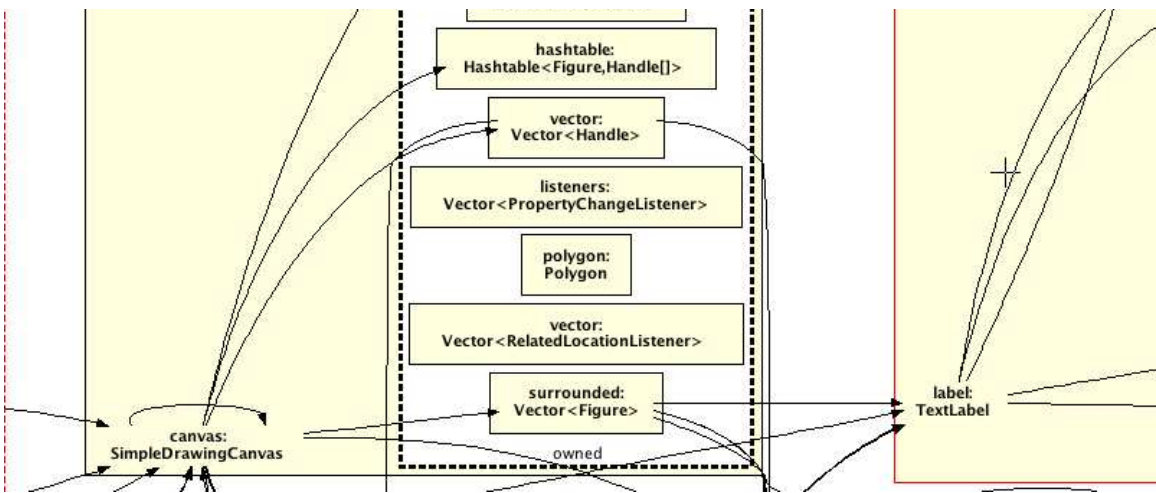
Figure 38: A version of the OOG after inferring generic types.

```
 1 @Domains({"owned"})
 2 @DomainParams({"V"})
 3 @DomainInherits({"CanvasHandle<V>", "FigureHolder<V>"})
 4 public class LocatorConnectionHandle extends CanvasHandle implements FigureHolder{
 5 // Before
 6   protected @Domain("owned") Rectangle getLocatorBounds()  {
 7         int myRadius = getHandleWidth() / 2;
 8         return new Rectangle(locator.x() - myRadius, locator.y() -
 9         myRadius, getHandleWidth(), getHandleHeight());
10     }
11      ...
12     protected @Domain("owned") Rectangle getConnectionBounds()  {
13         @Domain("lent") Locator center;
14         if (connection == null)
15             center = locator;
16         else
17             center = connection;
18         int myRadius = getHandleWidth() / 2;
19         return new Rectangle(center.x() - myRadius, center.y() - myRadius,
20         getHandleWidth(), getHandleHeight());
21     }
22
23 // After
24     protected @Domain("shared") Rectangle getLocatorBounds()  {
25         int myRadius = getHandleWidth() / 2;
26         @Domain("shared")Rectangle rectangle = new Rectangle(locator.x() - myRadius,
27         locator.y() - myRadius, getHandleWidth(), getHandleHeight());
28         return rectangle;
29     }
30      ...
31     protected @Domain("unique") Rectangle getConnectionBounds()  {
32         @Domain("lent") Locator center;
33         if (connection == null)
34             center = locator;
35         else
36             center = connection;
37         int myRadius = getHandleWidth() / 2;
38         @Domain("unique") Rectangle rectangle = new Rectangle(center.x() - myRadius,
39         center.y() - myRadius, getHandleWidth(), getHandleHeight());
40         return rectangle;
41     }
42      ...
43     }
```

Figure 39: Example of extracting local variable to be able to add annotations.

```
1  @Domains({"owned"})
2  @DomainParams({"V"})
3  @DomainInherits({"AbstractFigure<V>"})
4  public class TextLabel extends AbstractFigure implements LabelHolder, RelatedLocationListener{
5      ...
6      // Before
7      protected @Domain("owner") MovableLocator defaultLocator() {
8          return new DrawingPoint(0,0);
9      }
10     // After
11     protected @Domain("owner") MovableLocator defaultLocator() {
12         @Domain("owner") DrawingPoint drawingPoint = new DrawingPoint(0,0);
13         return drawingPoint;
14     }
15     ...
16 }
```

Figure 40: Re-writing a new expression by declaring a local variable.

```
1  @Domains({"owned"})
2  @DomainParams({"V"})
3  @DomainInherits({"Figure<V>"})
4  public abstract class AbstractFigure extends AbstractPaintable implements Figure {
5  ...
6  // Before
7  @Domain("shared<V>")
8  public static Figure figureFromLocator(@Domain("lent") Locator aLocator) {
9      if (aLocator instanceof FigureHolder) {
10         return ((FigureHolder)aLocator).getFigure();
11     }
12     if (aLocator instanceof RelativeLocator) {
13         return figureFromLocator(((RelativeLocator)aLocator).getBase());
14     }
15     return null;
16 }
17
18 // After
19 @Domain("owner<V>")
20 public static Figure figureFromLocator(@Domain("lent") Locator aLocator) {
21 if (aLocator instanceof FigureHolder) {
22     @Domain("owner<V>") FigureHolder figureHolder = (FigureHolder)aLocator;
23     return figureHolder.getFigure();
24     }
25 if (aLocator instanceof RelativeLocator) {
26     @Domain("owner") Locator base = ((RelativeLocator)aLocator).getBase();
27     return figureFromLocator(base);
28     }
29   return null;
30 }
31 ...
32 }
```

Figure 41: Re-writing a casting expression in `AbstractFigure` by declaring a local variable.

Field/Variable	Declared Type	Annotation	EnclosingType	AST Node
prototype	com.rolemodelsoft.drawlet.Figure	owner	PrototypeConstructionTool	prototype
canvas	com.rolemodelsoft.drawlet.DrawingCanvas	owner	PrototypeConstructionTool	@Domain("owner<V>") DrawingCanvas canvas
prototype	com.rolemodelsoft.drawlet.Figure	owner	PrototypeConstructionTool	@Domain("owner<V>") Figure prototype
newFigure	com.rolemodelsoft.drawlet.Figure	owner	PrototypeConstructionTool	newFigure=(Figure)prototype.duplicate()
prototype	com.rolemodelsoft.drawlet.Figure	owner	PrototypeConstructionTool	prototype

Figure 42: Annotating different instances of Figure with OWNER causes these instances to be merged under the owning domain, MODEL, of their supertype .
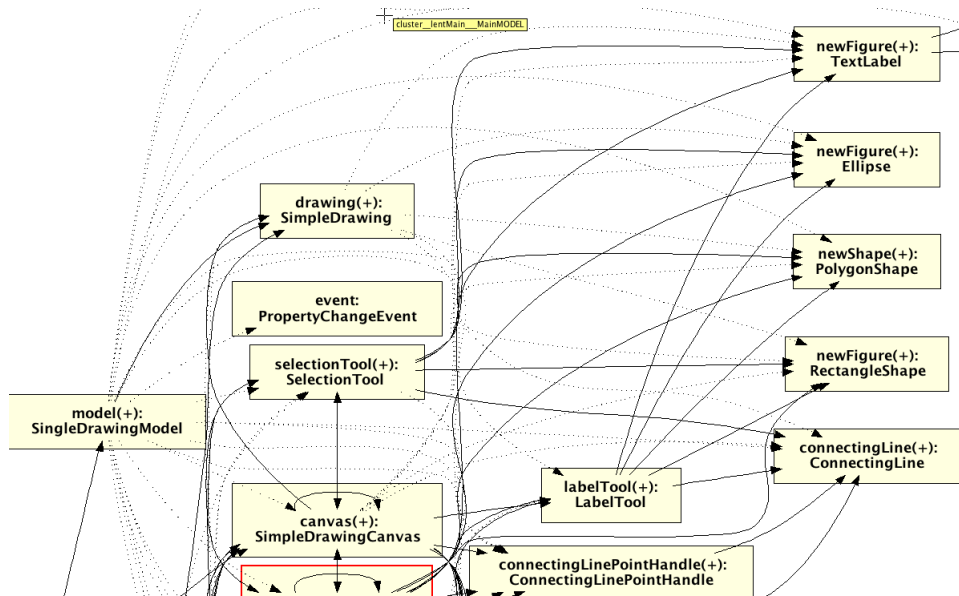

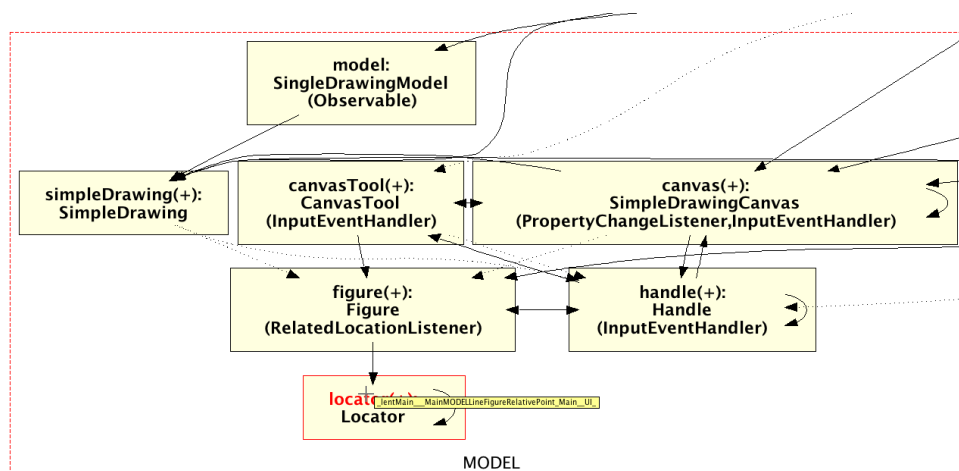Figure 43: The top level view of the OOG shows all different instances of Figure.

Figure 44: Fine tuning "abstraction by types" causes different Figure instances to be merged under their supertype.

36

```
1  @Domains({"owned","SUBS"})
2  public abstract class BasicObservable implements Observable, Serializable {
3
4  protected @Domain("owned") Object observerList;
5  private @Domain("owned<SUBS>") Vector<Observer> VIRT_observerList = new Vector<Observer>();
6      ...
7      }
```
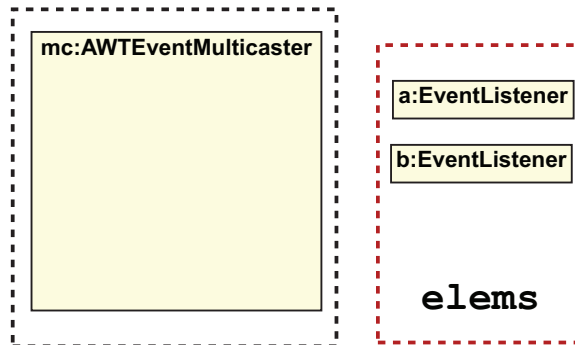
Figure 45: Adding virtual field to Observer

Figure 46: Declare domain parameter `elems` on `AWTEventMulticaster` to hold `EventListener` objects.

Virtual fields. The extracted run-time structure in figure 29 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 diagram 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 (Figure 45 shows an example).

4.3.3 Limitations in the current approach

Extra Overhead. The current approach prohibits an object from placing itself in an ownership domain that it declares. This was a problem especially for the root application object, i.e. `SimpleModelPanel`. The architectural extractor solved this problem with an extra level of indirection by creating a fake top-level class `Main` to declare the `MODEL` and `VIEW` top-level ownership domains. After that he declared the `SimpleModelPanel` object in the `VIEW` domain. The extra over head also includes inferring generic types and extracting local variables as discussed in section 4.3.

Knowledge of the subject system. In this study, the architectural extractor did not have a prior knowledge of the system, and he relied on the type checker to build his knowledge incrementally. The architectural extractor annotated several applications previously including JHotDraw, which is similar to DrawLets, but this knowledge only helped him annotate the system more effectively. Furthermore, in previous studies the architectural extractor relied on design documents and original designers to annotate the system, however, in the case of DrawLets the architectural extractor did not compare the extracted OOG to any as designed architecture; 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.

5 Discussion

The annotation-based approach has two layers of annotations. The second layer of annotations includes domain links and domain link assumptions to enforce architectural constraints between different run-time tiers. During the course of this study, we did not use this second layer of annotations even though 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.

In this study, we annotated the system as is and we did not try to fix any of the design flaws in the code. This implies that the extracted OOGs which we provided to the developer expressed the design of the as-built system. The developer's 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 patters. 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 with other objects of the same type that are in the same domain.

The ownership object graph static analysis currently does not show objects in the global public domain "shared", and this could result in a low quality OOG. Also, many objects were annotated using "lent" and "unique" especially within method body and on method return types, which implies that these objects do not appear on the diagram which could make it unsound. This could be true, especially in the case of lent annotation, but displaying objects that are currently in the shared domain, for instance, can add more clutter to the extracted OOG by displaying uninteresting edges. We often try to mitigate this problem by moving architecturally significant objects to non-shared domains to make them visible on the extracted diagram.

```
1  //AWTEventMulticaster
2  @Domains({"owned"})
3  @DomainParams({"elems"})
4  public class AWTEventMulticaster{
5  protected final @Domain("elems") EventListener a, b;
6
7  protected AWTEventMulticaster(@Domain("elems")EventListener a, @Domain("elems")EventListener b) {
8  this.a = a; this.b = b;
9      }
10
11 protected @Domain("elems") EventListener remove(@Domain("elems") EventListener oldl) {
12 if (oldl == a)   return b;
13 if (oldl == b)   return a;
14 @Domain("elems")
15 EventListener a2 = removeInternal/*<elems>*/(a, oldl);
16 @Domain("elems")
17 EventListener b2 = removeInternal/*<elems>*/(b, oldl);
18 if (a2 == a && b2 == b) {
19     return this;     // it's not here
20 }
21 return addInternal/*<elems>*/(a2, b2);
22 }
23
24 @DomainParams({"Mx"})
25 protected static   @Domain("Mx")EventListener addInternal(@Domain("Mx") EventListener a,
26                                                @Domain("Mx")EventListener b) {
27 if (a == null)   return b;
28 if (b == null)   return a;
29 @Domain("Mx<Mx>")AWTEventMulticaster eventMulticaster = new AWTEventMulticaster(a, b);
30 return eventMulticaster;
31 }
32 ...
33 }
34 //ColorButton
35 @Domains({"owned"})
36 public class ColorButton extends Component implements MouseListener {
37
38 protected @Domain("owner") ActionListener actionListener = null;
39
40 public void addActionListener( @Domain("owner") ActionListener listener ) {
41     actionListener = AWTEventMulticaster.add/*<owner>*/( actionListener, listener );
42 }
43 ...
44     }
```

Figure 47: Binding method domain parameters declared on `AWTEventMulticaster` inside `ColorButton`.

6 Conclusion

We conducted a case study where we added ownership domain annotations to an existing object-oriented system. We extracted a hierarchical object graph from the annotated code. We iterated the process of refining the annotations and the extracted object graph, until we got a diagram of the run-time structure of DrawLets, which we believed to be at an adequate level of abstraction. In future work, we will evaluate if the extracted diagrams convey interesting information to developers performing code modification tasks on DrawLets. Future work also includes replacing the defaulting tool with an ownership inference tool to reduce the annotation burden.

References

[1] M. Abi-Antoun and J. Aldrich. Ownership Domains in the Real World. In *Intl. Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, 2007.

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

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