Adding Ownership Domain Annotations to and Extracting Ownership Object Graphs from MiniDraw

Nariman Ammar **Marwan Abi-Antoun**

December 2011

Department of Computer Science
Wayne State University
Detroit, MI 48202

Abstract

We conducted a case study in adding ownership domain annotations to and extracting hierarchical object graphs from a small pedagogical object-oriented framework written in Java. We describe, using examples from the actual system, how we incrementally annotated the code then used static analysis to extract Ownership Object Graphs (OOGs) depicting the run-time structure of the system. We also discuss a preliminary evaluation of the extracted OOG and how we refined it before providing it to external developers.

Keywords: case study, ownership domains, ownership object graphs, static analysis, reverse engineering

Contents

List of Figures

1 Introduction

The software engineering research community has identified the importance of diagrams for program comprehension. Diagrams can represent the code structure, the runtime structure or the behavior of a software system. Diagrams can help developers to understand and modify the code, or to measure the potential impact of a planned modification [9].

There are multiple kinds of diagrams, and each diagram can answer some of the questions that developers may ask. In this report, we are concerned about diagrams of the run-time structure. For a given system, these diagrams can be either missing, or they are available, can be inconsistent with the code or lack much detail. So, it is often preferred to extract the diagram directly from the code, to ensure that the diagram is consistent with the code, and that it fully reflects the implementation.

Abi-Antoun and Aldrich have recently proposed the Scholia approach to statically extract from object-oriented Java code a hierarchical Ownership Object Graph (OOG), which is a diagram of the runtime structure. Producing an OOG for a system is an iterative process which involves three steps: adding annotations to the code to specify the design intent, extracting initial OOGs, then refining the extracted OOGs to make them convey design intent.

In this report, we present a case study in adding annotations to and extracting OOGs from a medium-sized, pedagogical, object-oriented framework. This case study was conducted in preparation for a controlled experiment that evaluated the usefulness of the OOGs as diagrams of the runtime structure that complement diagrams of the code structure such as class diagrams. Since we gave the OOG to developers and asked them to do code modifications on the system, we briefly report on a preliminary evaluation of the extracted OOGs with the experimenter prior to the experiment.

Outline. This report is organized as follows. Section 2 provides some background on Scholia including the ownership domain annotation syntax and the OOG graphical notation. Sections 3 describes the study setup. Section 4 describes the process of adding annotations (Section 4.1), extracting initial OOGs (Section 4.2) and refining them (Section 4.3). Section 5 evaluates the information content of the OOG, and discusses how the OOG answers key program comprehension questions about the system. We round out the report with a discussion of some issues and limitations (Section 6) and conclude.

```
@Domains( { "owned", "MAPS" })
@DomainParams( {"U","L","D"})
@DomainInherits({"StandardDrawing<U,L,D>", "BoardGameObserver <U,L,D>"})
class BoardDrawing extends StandardDrawing
  implements BoardGameObserver {

  protected @Domain("MAPS<D,MAPS<D>>") Map<Position, List<BoardFigure>> figureMap =
  new HashMap<Position, List<BoardFigure>>();

  protected @Domain("owned<shared, D<U,L,D>>") Map<String, BoardFigure> propMap = null;

  protected @Domain("L<U,L,D>") FigureFactory factory;
  protected @Domain("L") PositioningStrategy adjuster;

  @DomainReceiver("D")
 public BoardDrawing(@Domain("L<U,L,D>") FigureFactory factory,
                     @Domain("L") PositioningStrategy adjuster,
                     @Domain("D<U,L,D>") PropAppearanceStrategy propChanger ) {
  ...
  }
  ...
}

@Domains( { "owned" })
@DomainParams( {"U","L","D"})
@DomainInherits({"ImageFigure <U,L,D>"})
class BoardFigure extends ImageFigure {

  private @Domain("L<U,L,D>") Command command;

  public BoardFigure(@Domain("shared") String image, @Domain("shared") Point origin,
      boolean isMobile, @Domain("L<U,L,D>") Command command) {
          ...
   }
  ...
}
```

Figure 1: Ownership domain annotation syntax illustrated on a small example.

2 Background

2.1 Review of the Scholia Approach

Abi-Antoun and Aldrich [2] recently proposed the Scholia approach to extract a diagram of the runtime structure from an object-oriented system. The approach relies on adding annotations to the code, then running a static analysis to extract a hierarchical Ownership Object Graph (OOG). The extracted OOG can be refined by modifying the annotations or by changing the settings on the static analysis. The annotations implement the Ownership Domains type system, that ensures that the annotations are consistent with each other and with the code.

2.2 Review of Ownership Domain Annotations

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 an object in other domains [3].

There are two types of ownership domains:

Private domains provide strict encapsulation and make an object strictly owned by another. A public method cannot return an alias to an object inside a private domain, although the Java type system allows returning an alias to a field marked as private. The architectural extractor can declare and use only one private domain per class. The name has to be `owned` since the name is hard-coded in the typechecking tool. For example, the class `BoardDrawing` declares the private domain `owned` and stores the field `propMap` of type `Map` in it.

Public domains provide logical containment and make an object conceptually part of another. Having access to an object gives the ability to access objects inside all its public domains. The architectural extractor can declare and use public domains as much as he required. Public domain can have any name other than the private domain name, `owned`. For example, `BoardDrawing` declares a the public domain `MAPS` to hold a `figureMap` object.

2.2.2 The Annotation Syntax

The annotations use existing language support for Java 1.5 annotations. We illustrate the ownership domain annotations using code examples from MiniDraw (Fig. 1). In this section, we briefly explain the annotation syntax.

Domain Declaration `@Domains`. The architectural extractor must declare a domain before using it. For example, the architectural extractor declared the domains `owned` and `MAPS` (Line 1) before using them.

Domain Use `@Domain`. Each object is assigned to a single ownership domain that does not change at runtime. We indicate the domain of an object by annotating each reference to that object in the program. For example, the following statement declares the reference `adjuster` of type `PositioningStrategy` in the domain `L`.

```
protected @Domain( "L" ) PositioningStrategy adjuster;
```

Domain parameters declaration `@DomainParams`. Domain parameters allow objects to share state. Domain parameters can be declared and used as follows: The architectural extractor must declare the

domain parameters at the type declaration level before using them on the instances of that type. For example, the architectural extractor declared three domain parameters `U`, `L` and `D` on the type `BoardDrawing`, corresponding to the three domains `UI`, `LOGIC` and `DATA`, respectively (Fig. 1). After declaring domain parameters at the class level, they can be used to annotate the following:

- **Object declarations.** If at the type declaration there is a domain parameters declaration, then the architectural extractor must add the same number of domain parameters at any object instantiation of that type to match the declared domain parameters. For example, the type `BoardDrawing` uses three domain parameters, so any instance of the type `BoardDrawing` must has three domain parameters at the object instantiation statement. In the following example, the architectural extractor placed `boardDrawing` object in the domain parameter `D` with three domain parameters, `U`, `L` and `D`, that map the domain parameters declared at the class level:

```
@Domain("D<U,L,D>") BoardDrawing boardDrawing;
```

- **Parameterized types.** To specify domains for the parameterized code elements such as `Hashtable`. For example, the following code fragment has a `HashMap`, which has two parameterized elements, `key` and `values`:

```
@Domain("owned<shared,shared>") Map<String,Image> name2Image;
```

  The architectural extractor specified the domain for each one of these elements as follows: he used the `shared` domain for the `keys` of the type `String` and a `shared` domain for the `values` of the type `Image`, where the `HashMap` itself is in the domain `owned`, because it is owned by the class inside which it is declared.

Also, a parameterized type, in itself, can have domain parameters declared at the class level. In that case, the architectural extractor must specify the domain for each element of the parameterized type together with its domain parameters. For example, the parameterized type `Map` in the following statement has elements that map from `String` to `BoardFigure` for keys and values, respectively:

```
protected @Domain("owned<shared, D<U,L,D>>") Map<String, BoardFigure> propMap = null;
```

The architectural extractor placed the `String` elements into `shared` and the `BoardFigure` elements into the domain `D` with a list of domain parameters `<U, L,D>`.

Domain Inheritance `@DomainInherits`. Ownership domain annotations support type hierarchy by passing domain parameters from a sub-type to the corresponding super-types. Architectural extractors use the

`@DomainInherits` statement to pass domain parameters from one type to all of its super-types:

```
@DomainInherits ({Super-Type<Domain-Parameters>,...})
```

For example, the class `BoardDrawing` has two super-types (Fig. 1):

```
...
@DomainInherits({"StandardDrawing<U,L,D>", "BoardGameObserver <U,L,D>"})
class BoardDrawing extends StandardDrawing implements BoardGameObserver { ...
```

2.2.3 External Libraries

Ownership domain annotations use external XML files, AliasXML files, to add annotations to classes from library code that are in use, such as the type `java.util.Map` from the Java Standard Library. The architectural extractor can generate AliasXML files for these types to be able to add annotations to them. AliasXML files are type-specific but not project-specific, so they can be generated once and reused for multiple projects

2.2.4 Special Annotations

There are special annotations that add expressiveness to the ownership type system:

lent. One ownership domain can temporarily lend an object to another and ensure that the second object does not create persistent references to the first by marking it as lent.

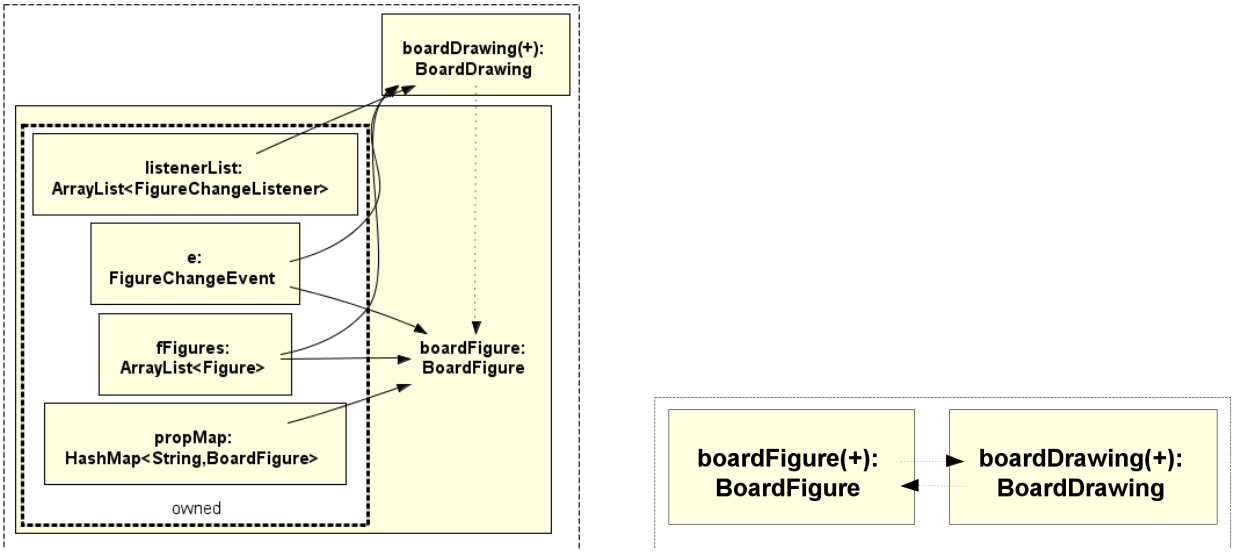
unique. unique indicates an object to which there is only one reference, such as a newly created object, or an object that is passed linearly from one domain to another.

shared. An object that is shared may be aliased globally but may not alias non-shared references, and little reasoning can be done about shared references. Shared objects do not show on the OOG.

The special domains `lent`, `shared` and `unique` are built-in annotations, so the architectural extractor does not have to declare them before using them.

2.3 Ownership Object Graph (OOG)

After adding annotations in the code, we use a static analysis to extract a hierarchical Ownership Object Graph (OOG) from the annotated code (Fig. 2). The extracted visualization uses box nesting to indicate containment of objects inside domains and domains inside objects. We use the following notation on an OOG to represent the different components: a domain represented by a white-filled box with a dashed-border, an object represented by solid-filled box, and a field reference represented by a solid edge. The object label



(a) Expanded OOG.

(b) Collapsed OOG.

Figure 2: Top-level OOG for above example (Fig. 1).

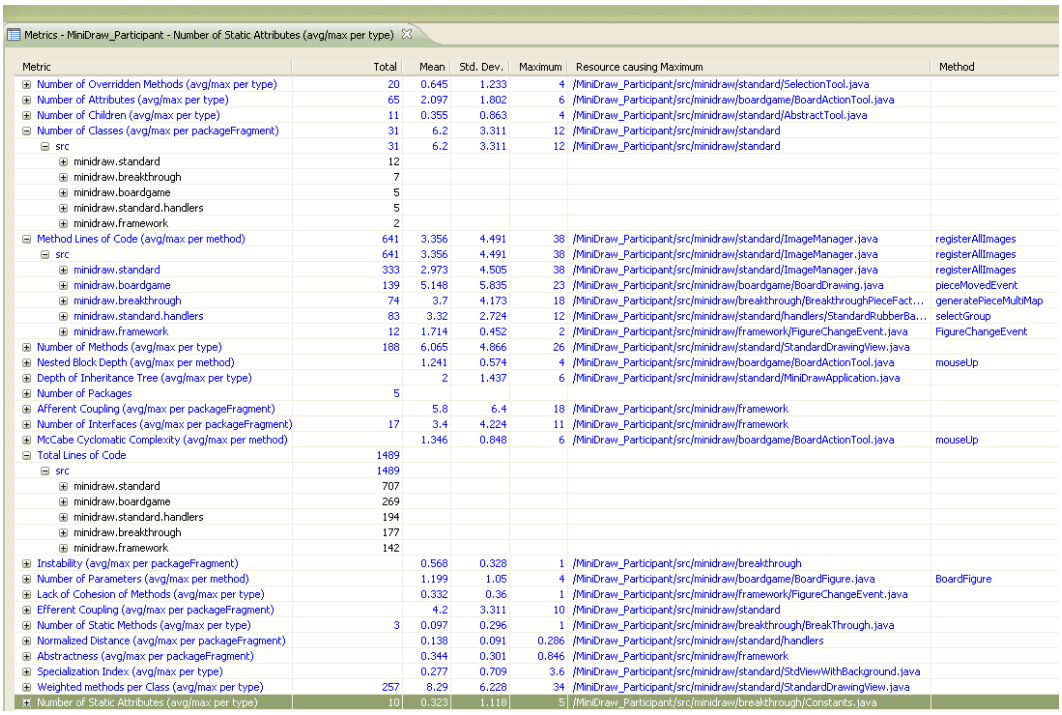
`obj:T` indicates an object reference `obj` of type `T`, which we then refer to either as "object `obj`" or as "`T` object", meaning for brevity, "an instance of the `T` class". The (+) symbol on an object or a domain indicates that it has a collapsed substructure and can be expanded to show the collapsed substructure;

3 Study Setup

3.1 Subject System

For the study, we used MiniDraw, a pedagogical object-oriented framework implemented to support the graphical aspects of board games [6]. MiniDraw comes with several applications, and we chose the Break-Through boardgame application for our study. We chose MiniDraw because it is rich in object-oriented design patterns and comes with several design diagrams including role diagrams, UML class diagrams, and sequence diagrams [6]. We also wanted to use the extracted OOGs for a future controlled experiment which compares OOGs to other diagrams.

According to the Eclipse metrics plugin [7], MiniDraw consists of around 1,500 lines of Java code, divided into 31 classes, 17 interfaces and 5 packages (Fig. 3).



Metrics - MiniDraw_Participant - Number of Static Attributes (avg/max per type)

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Overridden Methods (avg/max per type)	20	0.645	1.233	4	/MiniDraw_Participant/src/minidraw/standard/SelectionTool.java	
Number of Attributes (avg/max per type)	65	2.097	1.802	6	/MiniDraw_Participant/src/minidraw/boardgame/BoardActionTool.java	
Number of Children (avg/max per type)	11	0.355	0.863	4	/MiniDraw_Participant/src/minidraw/standard/AbstractTool.java	
Number of Classes (avg/max per packageFragment)	31	6.2	3.311	12	/MiniDraw_Participant/src/minidraw/standard	
src	31	6.2	3.311	12	/MiniDraw_Participant/src/minidraw/standard	
minidraw.standard	12					
minidraw.breakthrough	7					
minidraw.boardgame	5					
minidraw.standard.handlers	5					
minidraw.framework	2					
Method Lines of Code (avg/max per method)	641	3.356	4.491	38	/MiniDraw_Participant/src/minidraw/standard/ImageManager.java	registerAllImages
src	641	3.356	4.491	38	/MiniDraw_Participant/src/minidraw/standard/ImageManager.java	registerAllImages
minidraw.standard	333	2.973	4.505	38	/MiniDraw_Participant/src/minidraw/standard/ImageManager.java	registerAllImages
minidraw.boardgame	139	5.148	5.835	23	/MiniDraw_Participant/src/minidraw/boardgame/BoardDrawing.java	pieceMovedEvent
minidraw.breakthrough	74	3.7	4.173	18	/MiniDraw_Participant/src/minidraw/breakthrough/BreakthroughPieceFact...	generatePieceMultiMap
minidraw.standard.handlers	83	3.32	2.724	12	/MiniDraw_Participant/src/minidraw/standard/handlers/StandardRubberBa...	selectGroup
minidraw.framework	12	1.714	0.452	2	/MiniDraw_Participant/src/minidraw/framework/FigureChangeEvent.java	FigureChangeEvent
Number of Methods (avg/max per type)	188	6.065	4.866	26	/MiniDraw_Participant/src/minidraw/standard/StandardDrawingView.java	
Nested Block Depth (avg/max per method)		1.241	0.574	4	/MiniDraw_Participant/src/minidraw/boardgame/BoardActionTool.java	mouseUp
Depth of Inheritance Tree (avg/max per type)		2	1.437	6	/MiniDraw_Participant/src/minidraw/standard/MiniDrawApplication.java	
Number of Packages	5					
Afferent Coupling (avg/max per packageFragment)		5.8	6.4	18	/MiniDraw_Participant/src/minidraw/framework	
Number of Interfaces (avg/max per packageFragment)	17	3.4	4.224	11	/MiniDraw_Participant/src/minidraw/framework	
McCabe Cyclomatic Complexity (avg/max per method)		1.346	0.848	6	/MiniDraw_Participant/src/minidraw/boardgame/BoardActionTool.java	mouseUp
Total Lines of Code	1489					
src	1489					
minidraw.standard	707					
minidraw.boardgame	269					
minidraw.standard.handlers	194					
minidraw.breakthrough	177					
minidraw.framework	142					
Instability (avg/max per packageFragment)		0.568	0.328	1	/MiniDraw_Participant/src/minidraw/breakthrough	
Number of Parameters (avg/max per method)		1.199	1.05	4	/MiniDraw_Participant/src/minidraw/boardgame/BoardFigure.java	BoardFigure
Lack of Cohesion of Methods (avg/max per type)		0.332	0.36	1	/MiniDraw_Participant/src/minidraw/framework/FigureChangeEvent.java	
Efferent Coupling (avg/max per packageFragment)		4.2	3.311	10	/MiniDraw_Participant/src/minidraw/standard	
Number of Static Methods (avg/max per type)	3	0.097	0.296	1	/MiniDraw_Participant/src/minidraw/breakthrough/BreakThrough.java	
Normalized Distance (avg/max per packageFragment)		0.138	0.091	0.286	/MiniDraw_Participant/src/minidraw/standard/handlers	
Abstractness (avg/max per packageFragment)		0.344	0.301	0.846	/MiniDraw_Participant/src/minidraw/framework	
Specialization Index (avg/max per type)		0.277	0.709	3.6	/MiniDraw_Participant/src/minidraw/standard/StdViewWithBackground.java	
Weighted methods per Class (avg/max per type)	257	8.29	6.228	34	/MiniDraw_Participant/src/minidraw/standard/StandardDrawingView.java	
Number of Static Attributes (avg/max per type)	10	0.323	1.118	5	/MiniDraw_Participant/src/minidraw/breakthrough/Constants.java	

Figure 3: MiniDraw metrics snapshot.

3.2 Architectural Extractors

We will refer to people who added annotations and extracted OOGs as the architectural extractors. Three architectural extractors participated in the process of adding annotations, extracting OOGs, and refining extracted OOGs: two Ph.D. students from the SEVERE lab at Wayne State University and one of the original developers of SCHOLIA.

3.3 Tools and Instrumentation

We used three Eclipse plugins that support the process of extracting OOGs [1]:

The defaulting tool. This tool aims to reduce the annotation burden by automatically generating a set of initial default annotations. The tool adds annotations to local variables, temporary variables of methods and method's formal parameters with `lent`; private and protected fields and the return value of private or protected methods with `owned`; and `String` objects with `shared`.

The typechecker. After using the defaulting tool, architectural extractors often have to manually fix the added annotations and add missing ones. To achieve that, they run a type checker to check for the following: if there are missing annotations, if the domains are declared and used correctly, and if the annotations are

consistent with each other and with the code.

OOG extraction and viewing tool. The extraction tool extracts an OOG from the annotated code. The architectural extractors can use this tool to view the extracted OOG in both a graph-view and a tree-view. The tool has several features such as searching for an edge or an object either in the tree or on the graph, selecting an object in the tree and tracing to the corresponding field declaration in the code, and collapsing or expanding objects in the hierarchy to control the level of visual detail.

4 Procedure

In this section, we explain the general procedure we followed for adding annotations, extracting OOGs, and refining the extracted OOGs. The process of producing an OOG often requires the following steps from the architectural extractor:

Adding annotations: in this step, the architectural extractors add annotations to the code, run the typechecker to type check the added annotations, and manually fix any annotation warnings. The goal of this step is to minimize the number of annotation warnings. Since these warnings are indicators that the OOG may be unsound, the architectural extractors must attempt to resolve most of these warnings before moving to the next steps.

There are multiple ways of adding annotations to a system. In this study, we followed a strategy for adding annotations that decreases the annotation burden with high quality annotations that reflect the design intent. The architectural extractors follow the following procedure for adding annotations:

1. Adding AliasXML Files;
2. Selecting the Top-Level Domains;
3. Applying Annotation Defaulting Tool;
4. Propagating Domain Parameters;
5. Propagating Domain Inheritance;
6. Fixing Annotation Warnings;

Extracting initial OOGs: in this step, the architectural extractors run the static analysis to extract initial OOGs and tweak the annotations to obtain a less cluttered OOG. The goal here is to reduce the number of objects in the top-level domains.

Refining the extracted OOGs: in this step, the architectural extractors run the extracted OOGs by a developer who is familiar with the code, then refine the OOG to make it reflect the developer's mental model of the system.

4.1 Adding Annotations

4.1.1 Adding AliasXML Files

The first step in adding annotations to MiniDraw was adding AliasXML files to annotate external libraries such as the Java Standard Library. The architectural extractors reused AliasXML files from previously annotated systems. Some components in these files were missing annotations since these components were not used in previously annotated systems, so the architectural extractors had to add the missing annotations. Also, some of the library code was missing AliasXML files, so the architectural extractors had to generate and annotate new XML files for them. The number of AliasXML files used for annotating MiniDraw were 36 files, 14 of them were reused and 22 generated and annotated in this study, and can be used for any future studies that may use the same libraries.

4.1.2 Selecting the Top-Level Domains

The architectural extractors were familiar with the three-tiered architecture, and they knew that MiniDraw was designed following the `MODEL-VIEW-CONTROL` design pattern [6]. Therefore, they organized the objects into three top-level tiers or domains as follows:

- `UI`: contains objects from the user interface tier. For example, objects of types `DrawingEditor`, `StdViewWithBackground`, `DrawingChangeListener`, etc.;
- `LOGIC`: contains objects from the logic tier. For example, objects of types `FigureFactory`, `PositioningStrategy`, `BreakthroughFactory`, etc.;
- `DATA`: contains objects from the data tier. For example, objects of types `Position`, `BardFigure`, `PropAppearanceStrategy`, etc.;

4.1.3 Selecting the Root Class

In order to extract an OOG, the architectural extractors must specify a root class as a starting point for the extraction tool. Since MiniDraw is a framework, the architectural extractors had the option to extract

```
1  @Domains( { "UI", "LOGIC", "DATA" })
2  class BreakThrough {
3          @Domain("LOGIC<UI,LOGIC,DATA>")
4          Game game = new GameStub();
5
6          @Domain("LOGIC<UI,LOGIC,DATA>")
7          BreakthroughFactory factory = new BreakthroughFactory(game);
8
9          @Domain("UI<UI,LOGIC,DATA>")
10         DrawingEditor window = new MiniDrawApplication("Breakthrough Demo", factory);
11
12         public void init() {
13                 window.open();
14
15                 @Domain("DATA<UI,LOGIC,DATA>")
16                 BoardDrawing<Position> drawing = (BoardDrawing<Position>) window.drawing();
17
18                 @Domain("LOGIC<UI,LOGIC,DATA>")
19                 GameStub gameStub = (GameStub) game;
20                 gameStub.addObserver(drawing);
21
22                 @Domain("LOGIC<UI,LOGIC,DATA>")
23                 BoardActionTool boardActionTool = new BoardActionTool(window);
24
25                 window.setTool(boardActionTool);
26         }
27
28         public static void main(@Domain("lent[shared]") String[] args) {
29                 @Domain("lent")
30                 BreakThrough breakThrough = new BreakThrough();
31
32                 breakThrough.init();
33         }
34 }
```

Figure 4: The root class used in MiniDraw to extract OOG.

several OOGs for each of the framework applications. For the study, we chose the BreakThrough board game application, so the root class in our case was `BreakThrough` (Fig. 4).

The architectural extractors declared the three top-level domains on this class declaration (Fig. 4). Then they specified the domains of other objects in the root class. For example, they placed the `BoardDrawing` object in the `DATA` domain (Line 15), the `GameStub` object into the `LOGIC` domain (Line 18), and the `window:MiniDrawApp` object in the `UI` domain (Line 10).

4.1.4 Applying Annotation Defaulting Tool

The architectural extractors then applied the defaulting tool to add default annotations. While the annotation defaulting tool served as a director of where the annotations should be added, some of the annotations added by this tool were not correct, because it did not propagate domains other than `lent`, `shared` and `owned`, for a limited number of cases. Moreover, it did not use domain parameters, public domains, or

domain inheritance. The architectural extractors later refined the added annotations manually to replace some annotations with more precise ones while verifying that the modified annotations are consistent with the old ones.

4.1.5 Propagating Domain Parameters

After deciding on the three top-level domains, the architectural extractors decided to declare three domain parameters corresponding to each top-level domain:

- `U` : used to map the `UI` domain;
- `L` : used to map the `LOGIC`;
- `D` : used to map the `DATA`;

While the existence of unused domain parameters does not affect the extracted OOG, the typechecker complains about the wrong number of domain parameters declared on each type declaration. Therefore, the architectural extractors had to propagate these domain parameters to all type declarations in MiniDraw:

```
@DomainParams( {"U","L","D"})
```

Since they had declared three top-level domains, the architectural extractors declared three domain parameters on each class. After propagating the three domain parameters, they declared and used more domains as needed.

4.1.6 Propagating Domain Inheritance

After propagating domain parameters to all types in the system, the architectural extractors scanned all the types in MiniDraw, and for each type, if the type `implements` or `extends` other types, they added the `@DomainInherits` statement, which contained the super-type together with the list of domain parameters:

```
@DomainParams( {"U","L","D"})
@DomainInherits({"ImageFigure <U,L,D>"})
class BoardFigure extends ImageFigure {
```

4.1.7 Fixing Annotation Warnings

The architectural extractors followed the typechecker which generated warnings due to, but not limited to, one of the following violations:

(1) Missing annotations. For example, in the following statement, the `factory` instance is missing an annotation:

```
protected FigureFactory factory;
```

(2) Mismatched formal domain parameters and actual domains. For example, in the following statement, the number of actual domains declared on a variable of type `FigureFactory` does not match the required number of formal domain parameters declared on the type `FigureFactory`:

```
protected @Domain("L") FigureFactory factory;
```

(3) Violations of the assignment rule. For example, if a variable of type `BoardDrawing` is assigned to the `M` domain in one place, and another variable of type `BoardDrawing` was annotated with `L` in another place, the typechecker will complain about the inconsistent annotations at assignment statements.

```
@Domain("M") BoardDrawing b1;
@Domain("L") BoardDrawing b2;
b1 = b2; // This will generate a warning
```

The architectural extractors classified the warnings, based on the message generated by the type checker, into a prioritized list in hope of reducing the time for adding correct annotations by fixing the warnings in a systematic way.

4.2 Extracting Initial OOGs

The initially extracted OOG was too shallow and cluttered, so the architectural extractors had to modify the existing annotations to refine the extracted OOG and reduce the clutter in the top-level domains. At this stage, the architectural extractors had the OOG in Fig. 5. In the following section, we explain how the architectural extractors refined the initial OOG based on feedback from an external developer.

4.3 Refining Initial OOGs

Since the architectural extractors did not have full knowledge about all objects in the system, there was a risk that the extracted OOG does not reflect the design intent in the code. In order to evaluate the correctness of the extracted OOG, we gave it to a developer who knew more about the system as she had attempted some code modifications on the system. The developer was the person who will be using the extracted OOG in a later controlled experiment to evaluate its usefulness for external developers. Therefore, we refer to this developer as the *experimenter*.

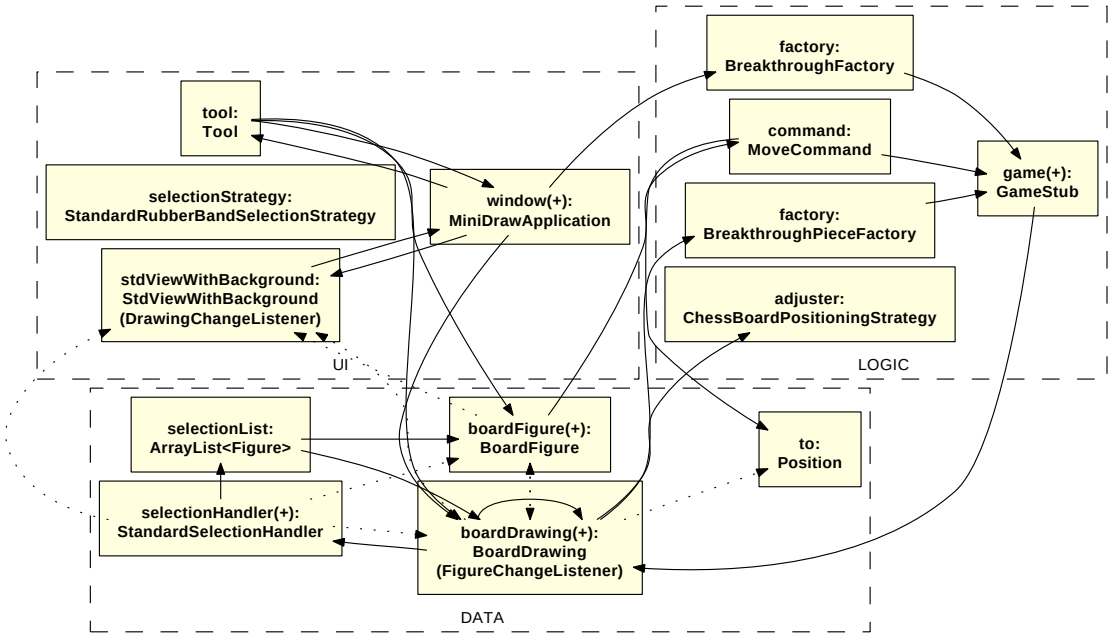


Figure 5: An OOG for BreakThrough.

We first explain the possible OOG refinements then illustrate how the architectural extractors applied some refinements based on feedback from the experimenter, using examples from MiniDraw.

4.3.1 Possible refinements of an OOG

The architectural extractors refined the initial OOGs either by modifying the existing annotations in the code or by fine-tuning some of the static analysis settings as follows:

- Move an object between sibling domains;
- Abstract a low-level object;
- Move an object to a higher-level domain;
- Merge objects that share a common super-type;
- Set the labeling types;

4.3.2 Move an object between sibling domains

There were cases where the architectural extractors assigned some objects to wrong domains, according to feedback from the experimenter, so they changed the annotations on these objects to place them in the

proper domains. For example, objects of type `Position` appeared in the domain `U` on the OOG, in the refinement step they decided to move it to the sibling domain `D`, as in the following example:

```
public @Domain("shared")Point calculateFigureCoordinatesIndexedForLocation
(@Domain("U<U,L,D>") Position location, int index){ ...

public @Domain("shared")Point calculateFigureCoordinatesIndexedForLocation
(@Domain("D<U,L,D>") Position location, int index){ ...
```

4.3.3 Abstract a low-level object

Make an object *part of* another object. In this step, the architectural extractors made an object conceptually part of another, by declaring a public domain inside the parent object and placing the child object inside that domain. For example, an object of type `SelectionHandler` was instantiated inside the class `StandardDrawing`, so, during the annotation process, the architecture extractors placed the `selectionHandler:SelectionHandler` object inside the domain `D`. Later on, the experimenter requested that this object should be pushed inside the `drawing:StandardDrawing` object, so the architectural extractors declared a pubic domain `HANDLERS` inside `drawing:StandardDrawing` and placed the `selectionHandler:SelectionHandler` object in it to make it logically part of `drawing:StandardDrawing`, yet accessible by other objects:

```
protected final @Domain("HANDLERS<U,L,D>")SelectionHandler selectionHandler;
```

As another example, the experimenter reported that the `map:Map<Position, List<Figure>>` is not architecturally significant and that it should not be in a top-level domain. So the architectural extractor declared a public domain `MAP`s inside `drawing:BoardDrawing` and placed `map:Map<Position, List<Figure>>` inside this domain as follows:

```
protected @Domain("MAPS<D,MAPS<D>>") Map<Position, List<BoardFigure>> figureMap =
new HashMap<Position, List<BoardFigure>>();
```

The reason why the architectural extractors declared the `map:Map<Position, List<Figure>>` inside a public domain in this case was because the `map:Map<Position, List<Figure>>` object should be accessible by other objects in the system.

Make an object *owned by* another object. The architectural extractors treated different objects differently based on their usage in the system. For example, the experimenter required the

`l:ArrayList<BoardFigure>` to be also pushed underneath `drawing:BoardDrawing`. However, the architectural extractors used a different annotation in this case since the `l:ArrayList<BoardFigure>` object should not be accessed by other objects, so they placed it inside a private domain, i.e., a domain declared as `owned`, of the parent object. This kind of refinement was mostly applied to data structures. Similarly, the experimenter requested that the `selectionList:ArrayList<SelectionHandler>` object (Fig. 5) should be nested under its owning object instead of showing in the top level domains.

4.3.4 Merge objects that share a common super-type

The architecture extractors found the initial OOG to be cluttered, so they decided to apply the abstraction by types to merge objects, within the same domain, that share a common super-type. For example, the architectural extractors found that the objects of types `minidraw.standard.NullTool`, `minidraw.standard.SelectionTool` and `minidraw.boardgame.BoardActionTool` all share the super-type `minidraw.framework.Tool` and reside in the same domain `CONTROLLER`. So, they added the type `minidraw.framework.Tool` to the list of design intent types. With abstraction by types turned on, the extraction tool merged the above objects into one object, `tool:Tool`.

4.3.5 Set the labeling types

Each object on the OOG can have extra decorating labels to provide more information about these objects. For example, the architectural extractors used the following list of labeling types:

```
minidraw.framework.FigureChangeListener
minidraw.framework.DrawingChangeListener
minidraw.boardgame.BoardGameObserver
java.awt.event.MouseListener
java.awt.event.MouseMotionListener
java.awt.event.KeyListener
```

They added these types to the list of labeling types used by the extraction tool. Using these types caused some objects like `BoardDrawing` to have the label `BoardGameObserver`. Such a label may help the developer to understand that a `Drawing` object is an observer of the `Game` objects (Fig. 6)

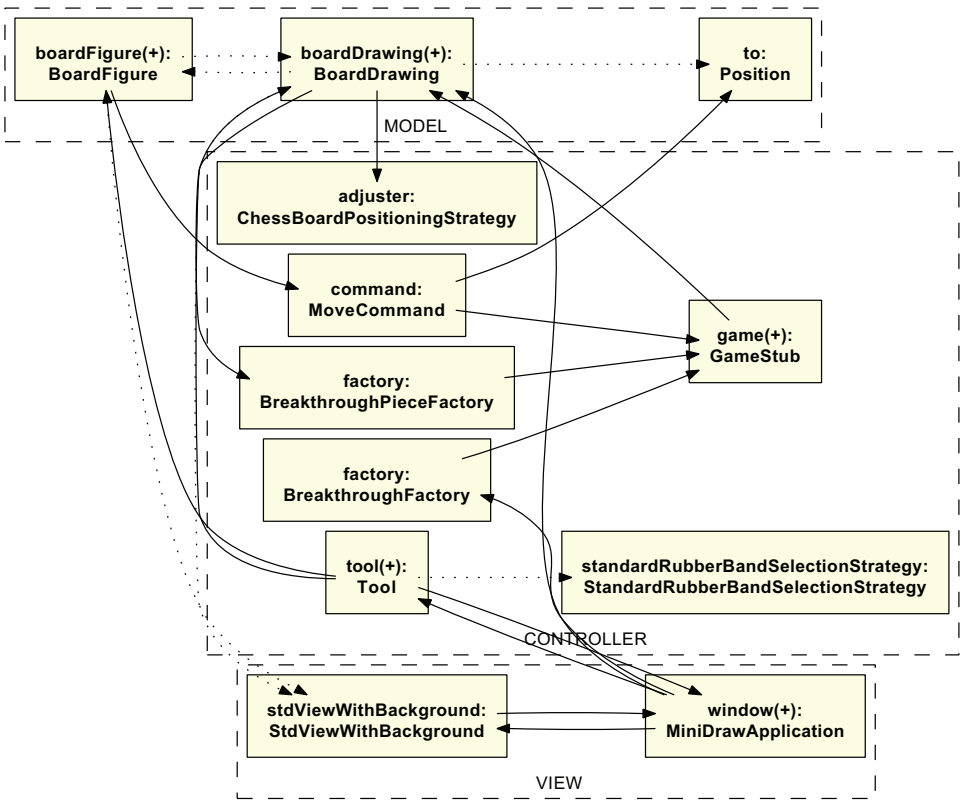


Figure 6: MiniDraw OOG.

5 Evaluating the OOG

5.1 Further Refinements

In addition to the above refinements of the initial OOG, we refined the OOG further. The full set of refinements is discussed elsewhere [4, Chap.2]. Figure 7 shows the final OOG.

5.2 How does an OOG answer key program comprehension questions?

Understanding the object structure is *fundamental* to the program comprehension of object-oriented code. As a diagram of the run-time structure, the OOG highlights some key facts about the system's design that the developers who use the MiniDraw framework need to learn. In this section, we evaluate the extracted OOG as a diagram of the run-time structure by highlighting some key program comprehension questions that we believe an OOG can help developers answer.

Instances matter in object-oriented code. In object-oriented design patterns, much of the functionality is determined by what instances point to what other instances. For instance, in the Observer design

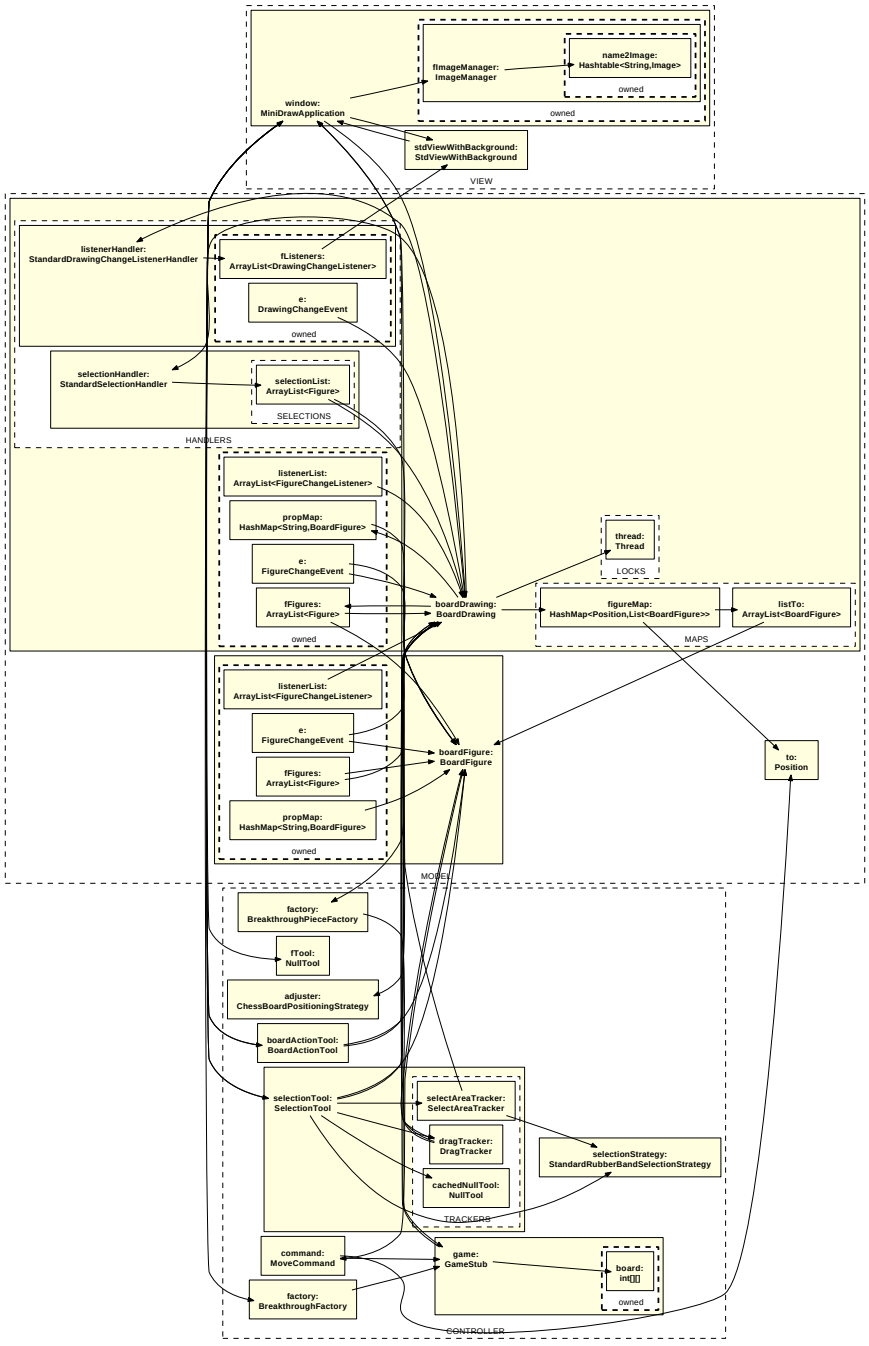


Figure 7: An expanded view of the OOG.

```
class GameStub implements Game {
  private BoardGameObserver observer;

  public void addObserver(BoardGameObserver observer) {
     this.observer = observer;
  }
}

class BoardDrawing extends StandardDrawing implements BoardGameObserver {
  protected Map<Position, List<BoardFigure>> figureMap = new HashMap<Position, List<BoardFigure>>();
}
```

Figure 8: Implementation of the Observer pattern, between BoardDrawing and GameStub.

pattern [8], understanding "what" gets notified during a change notification is crucial for the operation of the system, but "what" does not usually mean a class, "what" means an instance.

Programming to an interface makes the code harder to understand. In object-oriented design, one principle is to program to an interface rather than to an implementation class. MiniDraw follows this recommendation [5]. For example, in MiniDraw, the `BoardDrawing` class implements the `BoardGameObserver` interface, and if a developer sees the relation between `game:GameStub` and `boardDrawing:BoardDrawing` on the OOG, they would assume that they can access the `boardDrawing:BoardDrawing` object from the `game` object (Fig. 6). To investigate further, they can trace to that specific instance in the code. However, that instance will be declared as having the interface type `BoardGameObserver` inside the `GameStub` class instead of the implementation class type `BoardDrawing` (Fig. 8). A feature in the OOG shows the type `BoardGameObserver` as a labeling type on the `boardDrawing` object, which could help developers understand why an object of type `BoardDrawing` would be declared in the code as being of the more general type `BoardGameObserver`. Admittedly, developers can use the Eclipse Type hierarchy to understand the inheritance relation between the interface `BoardGameObserver` and the class `BoardDrawing`.

Do specific instances really matter? An OOG does not pin things down to individual objects. Instead, it abstracts objects by domains and types. For example, it merges several objects of the same or similar types that are in the same domain. If the code creates many instances of the `BoardFigure` class at runtime, and the latter are all in the `MODEL` domain in the `BreakThrough` object, the OOG shows one `boardFigure` object in the `MODEL` domain (Fig. 6). Still, developers can select a canonical object on the OOG and trace to all the lines of code that may create such an object. For example, the tool object on the MiniDraw OOG represents multiple instances of multiple types that implement the `Tool` interface, i.e., `boardActionTool:BoardActionTool`, `ftool:NullTool`, `selectionTool:SelectionTool` (Fig. 7). However, we merge these objects and show them as only one box on the OOG (Fig. 5). Still the developer can trace to the different instances in the code.

If despite merging objects, OOGs hold enough precision and are still useful for program comprehension, an instance may not matter in terms of "the particular object". It seems enough to pin things down just to objects of a type that are within a domain. So this leads us to refine the question:

Does information about *types + ownership + domains* **answer key questions in program comprehension?** We believe that what really matters is the *role* an instance is playing, and information about *types+ownership+domains* give us a richer language for describing that role than type alone. For instance, we can express facts like *"an object of type A* in domain *D* in an object of type *B*", that we show as triplets ≺A,D,B≻. So the question becomes: *how often does the ability to distinguish the role of instances not just by type, but by named groups (domains) or by position in the run-time structure (ownership), matter for code modification tasks?*

We believe an OOG is useful during coding tasks because it conveys that information. We also believe that there are situations where types are not enough, but domain and ownership information give developers exactly what they need for some of the tasks especially those that require searching for a data structure owned by an object. Indeed, the code in the case of MiniDraw uses many `HashMap` instances, and the one that the developer really needs is the `HashMap` in the `MAPS` domain in the `boardDrawing` object (of type `BoardDrawing`). This fact can be represented as the triplet ≺`HashMap, MAPS, BoardDrawing`≻ and is visually obvious on the OOG (Fig. 7).

We also believe that such examples are reasonably common in object-oriented code. For MiniDraw, we were able to count at least 10 such instances (Table 5.2). The OOG (Fig. 7) graphically displays some of these triplets. For example, according to the MiniDraw documentation, some of the responsibilities of the `Drawing` interface are expressed in smaller and more fine-grained interfaces. Namely, `DrawingChangeListenerHandler` defines the management of listeners or observers, and `Selectionhandler` defines the selection handling responsibility. The OOG shows both `selectionHandler:StandardSelectionHandler` and `listnerhandler:DrawingChangeListenerHandler` inside a public domain `HANDLERS` underneath `drawing:BoardDrawing`. Also, the OOG shows `selectionList:ArrayList<Figure>` in a public domain `SELECTIONS` inside `selectionHandler:StandardSelectionHandler`, since each drawing maintains a temporary, possibly empty, subset of all figures called a selection and those are handled by the `selectionHandler:StandardSelectionHandler`. Also, non-architecturally significant objects such as `map:Map<Position, List<Figure>>` and `l:ArrayList<BoardFigure>` are inside `drawing:BoardDrawing` instead of being in the top-level domains. The architectural extractors declared `map:Map<Position,`

Table 1: Examples from MiniDraw to illustrate facts about the runtime structure in terms of a triplet *object of type A owned by domain D in an object of type B* or ≺A,D,B≻.

Object of Type A	...in domain D	...in an Object of Type B
selectionHandler:StandardSelectionHandler	HANDLERS	boardDrawing:BoardDrawing
listenerHandler:StandardDrawingChangeListenerHandler	HANDLERS	boardDrawing
figureMap:HashMap<Position,List<BoardFigure>>	MAPS	boardDrawing:BoardDrawing
selectionList:ArrayList<Figure>	SELECTIONS	selectionHandler:StandardSelectionHandler
thread:Thread	LOCKS	boardDrawing:BoardDrawing
fListeners:ArrayList<DrawingChangeListener>	owned	listenerHandler:StandardDrawingChangeListenerHandler
thread:Thread	LOCKS	boardDrawing:BoardDrawing
selectAreaTracker:SelectAreaTracker	TRACKERS	selectionTool:SelectionTool
dragTracker:DragTracker	TRACKERS	selectionTool:SelectionTool
cachedNullTool:NullTool	TRACKERS	selectionTool:SelectionTool

List<Figure>> inside a public domain MAPS inside drawing:BoardDrawing. Also, trackers in MiniDraw are part of the SelectionTool according to the JavaDoc. As a result, the OOG shows the dragTracker:DragTracker and selectAreaTracker:SelectAreaTracker objects in a public domain, TRACKERS, inside the tool:SelectionTool object.

The special nature of the top-level domains. Domains are not just at the top level of an OOG. They can appear at different levels based on how much information a developer wants to see and the level of detail. However, on an OOG, a top-level domain expresses the design intent related to architectural tiers. MiniDraw follows the Model-View-Controller design pattern, and we express this on the OOG by showing objects in three top-level domains, namely, MODEL, VIEW, and CONTROLLER. This fact helps developers locate where to make changes in the code. For instance, if a developer has to do a code modification that is related to the user interface of the application, and they see the window:MiniDrawApp object inside the VIEW domain, then most probably, the modification will take place in the MiniDrawApp class.

Also, the points-to relations between objects across different domains tells developers how objects communicate across architectural tiers. For example, we show the list of DrawingChangeListenerHandlers (fListeners:List<DrawingChangeListener>) inside a private domain under DrawingChangeListenerHandler which is inside drawing:BoardDrawing (Fig. 7). Moreover, fListeners:List<DrawingChangeListener> points to a StdViewWithBackground object, which indicates that the elements in the list could be of any type. In this particular case, they are of type StdViewWithBackground. This placement of objects can help developers understand better how the drawing:BoardDrawing, which is a MODEL component, and StdViewWithBackground objects, which is a VIEW component, communicate using the observer design pattern.

Distinction between public and private domains.

The visual distinction between public and private domains on the OOG provides developers with hints

```
// Declare private domain "owned"
// Declare public domain "MAPS", "HANDLERS", ...
@Domains( { "owned", "LOCKS", "HANDLERS", "MAPS" })
@DomainParams( { "U", "L", "D" })
@DomainInherits( { "StandardDrawing<U,L,D>", "BoardGameObserver <U,L,D>" })
class BoardDrawing extends StandardDrawing implements BoardGameObserver {
  // Map each location to the set of images positioned on it
  protected @Domain("MAPS<D,MAPS<D>>")
  Map<Position, List<BoardFigure>> figureMap = new ...;

  // Map graphical (x,y) positions to the props of the game
  protected @Domain("owned<shared, D<U,L,D>>")
  Map<String, BoardFigure> propMap = new ...;
...
}
```

Figure 9: Different instances of `Map` are placed in different domains.

```
Domains( { "owned" })
@DomainParams( {"U","L","D"})
@DomainInherits({"DrawingChangeListenerHandler<U,L,D>"})
class StandardDrawingChangeListenerHandler
              implements DrawingChangeListenerHandler {

  /** list over all associated listeners */
  @Domain("owned<U<U,L,D>>")
  protected List<DrawingChangeListener> fListeners;
...
}
```

Figure 10: developers should not get a persistent reference to the `fListeners` list, therefore it should be declared in a private domain.

about which objects are strictly encapsulated, and which objects are only logically contained. MiniDraw has many examples that illustrates this, for example, seeing the `figureMap` object declared inside the `MAPS` public domain under `drawing:BoardDrawing` indicates that developers can get that object from `drawing:BoardDrawing` by calling the public method `getFigureMap()` (Fig. 9).

On the other hand, the fact that `fListeners:List<DrawingChangeListener>` is declared in a private domain inside `DrawingChangeListenerHandler` indicates that developers should not get a reference to this object (Fig. 10). These facts could save developers from introducing hacks in the code such as declaring public getter methods in order to access private objects.

6 Discussion

6.1 Architectural extractors' effort estimates

For this study, we did not accurately record the time spent in annotating and refining the OOGs. The annotation effort is currently estimated at around 1-hour per 1,000 Lines of Code [1], and we do not require

2 Ph.D. students for adding annotations to MiniDraw (1,500 LOC). In fact, MiniDraw was the first system on which the two students practiced adding annotations and extracting OOGs. The third extractor was an expert in adding annotations, so he mentored the first two extractors to ensure that they added good quality annotations.

6.2 Architectural extractors' thought process

One may argue that the architectural extractors need to read the entire code base and understand how the system works prior to adding annotations. Even though the architectural extractors had access to a book about the subject system which explains the design of the code and provides several UML diagrams, they did not read the entire book in order to annotate the entire system. Extracting OOGs requires that the architectural extractors focus on the structure of the system rather than its behavior, so they do not need to have a global knowledge of how the system works in order to annotate the code. The static analysis to extract OOGs, on the other hand, needs a root class, and is a whole program analysis. Therefore, to extract an OOG that reflects the design intent in the code, the architectural extractors provide local hints and add local, modular annotations. The annotations are modular in that they can be checked one class at a time. After that, the architectural extractors use the type checker to check for inconsistencies. Then, they refine the annotations incrementally without repeating the whole process.

6.3 Evaluation of the extracted OOG

We asked the experimenter, the first author of this report, to evaluate the initial OOG before using it in a controlled experiment. One may argue that we have been biased by evaluating the initial OOG by the experimenter, especially since the goal of the experiment was to evaluate the usefulness of this diagram for external developers, and that we should have kept the diagram as-is. As we discussed in this report, OOGs are extracted semi-automatically from the code using static analysis. We make an OOG relevant for developers who are performing code modifications by making the OOG convey design intent and reflect the designer's mental model of the system. Therefore, the only reason why we asked the experimenter to evaluate the diagram is that she was the only person, at the time of the study, who was familiar with the MiniDraw code since she made a few modifications to the code while preparing for the experiment. In fact, the experimenter made more refinements to the extracted OOG, as discussed elsewhere [4]. The goal of the experiment, on the other hand, was to evaluate whether other outside developers can use the OOG to answer some of their key program comprehension questions and understand the system's design.

7 Conclusion

We conducted a case study in adding annotations to and extracting OOGs from a medium-sized, pedagogical, object-oriented framework. The process of adding annotations and extracting OOGs required reasonable effort from the architectural extractors who were able to extract OOGs at an adequate level of abstraction for developers who will be using the diagram for future code modifications. The study was conducted in preparation for a controlled experiment for evaluating the usefulness of the extracted OOGs for developers doing code modifications on the system. Therefore, we provided a preliminary evaluation of the extracted OOG. The refinement process did not require much effort from the architectural extractors, who refined the OOG by incrementally refining the annotations in the code and the settings of the static analysis.

Acknowledgements

The authors thank Zeyad Hailat and Anas Al-Tirawi for their help with adding initial annotations to MiniDraw. In addition, the authors thank other participants in a directed study course during Fall 2010 for providing feedback during the MiniDraw case study.

References

[1] M. Abi-Antoun. *Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure*. PhD thesis, Carnegie Mellon University, 2010. Technical Report CMU-ISR-10-114.

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

[3] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object-Oriented Programming (ECOOP)*, 2004.

[4] N. Ammar. Evaluation of the Usefulness of Diagrams of the Run-Time Structure for Coding Activities. Master's thesis, Wayne State University, 2011. Available at: `www.cs.wayne.edu/~mabianto/`.

[5] H. B. Christensen. Frameworks: Putting Design Patterns into Perspective. In *Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, 2004.

[6] H. B. Christensen. *"Flexible, Reliable Software: Using Patterns and Agile Development"*. Chapman and Hall/CRC, 2010.

[7] Eclipse Metrics Plugin. `http://metrics.sourceforge.net/`, 2010.

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

[9] P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code (Monographs in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.