A Case Study in Extracting the Runtime Architecture of an Object-Oriented System

Marwan Abi-Antoun **Zeyad Hailat**

April 2011

SoftwarE Visualization and Evolution REsearch group (SEVERE)
Wayne State University
Detroit, MI 48202

Abstract

Many legacy systems benefit from having an up-to-date documented runtime architecture. But it is hard to statically extract a sound hierarchal runtime architecture from a legacy object-oriented system that is written in a general-purpose programming language and that follows common design idioms, so many have preferred dynamic analyses. Abi-Antoun and Aldrich recently developed a static analysis to extract runtime architectures from legacy object-oriented programs written in a mainstream language. The approach relies on adding annotations to the code, and statically extracts a hierarchical object graph from the annotated code. The annotations implement a type system, so a typechecker can check that the annotations are consistent with each other and with the code. We present a case study to evaluate the method and the tools on a 36-KLOC system that is in wide use and is actively evolving.
We developed a tool to reduce the annotation burden by generating the initial set of annotations. Then, we used a typechecker to check the annotations, and manually fixed the annotation warnings. We then extracted a hierarchical object graph, which we refined on our own, before showing it to the system maintainer. The maintainer was able to match the extracted object graph to his mental model of the system and gave us suggestions for further refinement. We incorporated the maintainer's feedback by slightly adjusting the annotations and using additional features in the extraction tool, to make the extracted diagram convey the maintainer's design intent.

Keywords: case study, architecture extraction, reverse engineering, runtime architecture, static analysis

Contents

1 Introduction **4**

2 Background **5**
- 2.1 Ownership domain annotations . . . 5
- 2.2 Object graph extraction . . . 8
- 2.3 Subject System . . . 8

3 Extraction of the Architecture **9**
- 3.1 Pre-processing . . . 10
- 3.2 Deciding on the organization of the objects into domains . . . 11
  - 3.2.1 Determining the top-level domains . . . 11
  - 3.2.2 Mapping objects to domains . . . 11
  - 3.2.3 Determining the formal domain parameters . . . 12
- 3.3 Propagating the initial set of annotations . . . 12
  - 3.3.1 Earlier defaulting tool . . . 12
  - 3.3.2 ArchDefault tool . . . 12
  - 3.3.3 Reducing the map size . . . 14
  - 3.3.4 Applying ArchDefault . . . 14
  - 3.3.5 Adding AliasXML files . . . 14
- 3.4 Adding annotations . . . 15
  - 3.4.1 Using ArchCheckJ . . . 15
  - 3.4.2 Refactoring problematic statements . . . 15
  - 3.4.3 Adding missing annotations . . . 16
  - 3.4.4 Adding AliasXML files . . . 16
  - 3.4.5 Fixing array annotation warnings . . . 16
  - 3.4.6 Removing unused domain parameters . . . 16
  - 3.4.7 Remaining warnings . . . 17
- 3.5 Extracting and Refining OOGs . . . 17
  - 3.5.1 Determining the root class . . . 17
  - 3.5.2 Extracting initial OOGs . . . 18
  - 3.5.3 Refining initial OOGs . . . 18

4 Evaluation of the Extracted Architecture **18**
- 4.1 Method . . . 19
  - 4.1.1 Participant. . . . 19
  - 4.1.2 Tools and Instrumentation. . . . 19
  - 4.1.3 Questions. . . . 20
  - 4.1.4 Procedure. . . . 21
- 4.2 Results . . . 21
- 4.3 Additional refinement of the OOG . . . 23
  - 4.3.1 Analysis of maintainer feedback . . . 23
  - 4.3.2 OOG refinements . . . 23
  - 4.3.3 Changing annotations . . . 24
  - 4.3.4 Using abstraction by types . . . 24

5 Discussion and Lessons Learned **24**
- 5.1 Quantitative Data . . . 24
  - 5.1.1 Effort data . . . 24
  - 5.1.2 Metrics. . . . 25
- 5.2 Effectiveness of the ArchDefault tool . . . 26

5.3 Lessons learned from adding annotations 26
5.4 Lessons learned from extracting OOGs 26

6 Related Work **28**

7 Conclusion **30**

List of Figures

1 Ownership domain annotation syntax illustrated on a small example. 6
2 OOG for the above example (Fig. 1). 9
3 The final extracted OOG of P-X. 19
4 Snapshot of ArchRecJ with the P-X OOG. 20
5 Example of adding a warning about representation exposure. 27

1 Introduction

During software evolution, the most reliable and accurate description of a software system is its source code [1]. In addition, high-level architectural diagrams of the system's organization can be useful. Many legacy object-oriented systems are undergoing maintenance, but either lack documented architectures or have out-of-date diagrams. Hence, we need to extract the as-built architecture of the system from the code, for the purposes of redocumentation [2]. For instance, a diagram can help locate the components that must be modified, or indicate the magnitude of the impact of a change based on the dependencies among entities on the diagram.

Many architectural views are needed to describe a software system. The *code architecture* organizes code entities in terms of classes and packages [3], and is useful for studying properties such as maintainability. Another useful view, the *runtime architecture* [3], models runtime entities and their potential interactions. A runtime architecture is important, because it impacts quality attributes such as security, performance, and reliability. In this study, our focus is on the runtime architecture.

Despite receiving much research attention, extracting runtime architectures remains a hard problem. Many have preferred dynamic analyses (or mixed with static analyses) to extract the as-built runtime architecture [4, 5]. A dynamic analysis monitors one or more program runs and shows snapshots of the runtime architecture for those runs. But these descriptions are partial and cover only a few representative interactions between objects, based on particular inputs and exercised use cases. A true architecture is meant to capture a complete description of the system's runtime structure. To meet this goal, a static analysis is preferred. A static analysis must also be *sound* and not fail to reveal entities or relationships that actually exist at runtime. For instance, an architectural-level security analysis requires a complete architectural description to handle the worst, not the typical, case of runtime communication.

At runtime, an object-oriented system can be represented as an *object graph*: nodes correspond to objects, and edges correspond to relations between objects. Taking a snapshot of the heap at runtime reveals the structure at that instant in great detail, but the profusion of objects makes it difficult to get a high-level picture that reveals any design intent. In fact, many reverse engineering tools run the risk of extracting abstractions that the developers do not recognize [6, 7].

To statically extract an object graph which conveys design intent, Abi-Antoun and Aldrich designed the Scholia approach [8]. Scholia is a two-pronged approach that requires first adding annotations in the code to specify some architectural intent. It then uses a static analysis to extract a sound, hierarchical object graph from an annotated program. The hierarchical object graph provides architectural abstraction by ownership hierarchy and by types, where architecturally significant objects are near the top of the hierarchy and data structures are further down. To achieve the hierarchy, the developer picks a top-level object as a starting point, then uses local, modular, ownership annotations in the code [9] to control the abstraction and impose a conceptual hierarchy on the objects in the system.

A few questions remain regarding the effort involved in using the Scholia approach, due to the annotation burden, which is currently a mostly manual process. While automated inference of ownership annotations is an active research area, the current tools cannot be used by the Scholia approach because they neither scale to large programs nor infer precise annotations.

Contributions. This report contributes the following:

- A case study illustrating the approach on a medium-sized, object-oriented system undergoing maintenance;
- A tool to reduce the annotation burden by propagating a large number of the initial annotations;
- A confirmation that the extracted object graphs can be refined to match the maintainer's mental model of the system.

Outline. This paper is organized as follows. Section 2 provides some background on the annotations and the object graph extraction, which are at the core of Scholia. Section 3 describes how we extracted and refined the architecture on our own before showing it to the maintainer. Section 4 discusses the feedback from the maintainer and the additional refinement of the architecture. Section 5 discusses some lessons learned. Finally, we discuss related work (Section 6) and conclude.

2 Background

2.1 Ownership domain annotations

We refer to the person who cleaned-up the system, added annotations and fixed their warnings, extracted and refined the initial OOG, met with the system maintainer, and refined and extracted the final OOG as the architectural extractor. The architectural extractor uses local, modular annotations to specify, in code, strict encapsulation, logical containment and architectural tiers, which are not explicit constructs in general-purpose programming languages.

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 [9]. The annotations use existing language support for Java 1.5 annotations [10] (Fig. 1). The annotations implement a type system and are checked

```
 1 @Domains({ "owned", "PARAGS" })
 2 @DomainParams({ "U", "L", "D" })
 3 public class IniFile {
 4   @Domain("shared") String filename;
 5
 6   @Domain("owned<shared, PARAGS<U,L,D>>")
 7     Hashtable<String, IniParagraph> paragraphs;
 8
 9   @Domain("PARAGS<U,L,D>") IniParagraph para;
10
11   @Domain("L") InputFile f;
12   ...
13 }
14
15 @Domains({ "owned" })
16 @DomainParams({ "U", "L", "D" })
17 public class IniParagraph {
18   @Domain("L") InputFile f;
19   ...
20 }
21
22 // From Java Standard Library
23 @DomainParams({ "DOMKEY", "DOMVALUE" })
24 @DomainInherits({"Map<DOMKEY,DOMVALUE>"})
25 public class Hashtable<TK,TV>  implements Map<TK,TV>{
26   @Domain("DOMKEY") TK key;
27   @Domain("DOMVALUE") TV value;
28   ...
29 }
30 // Root class, used for OOG construction
31 @Domains({"UI","LOGIC","DATA"}) public class System {
32    // Outer LOGIC is the domain of the reference
33    // Class IniFile is parameterized with <U,L,D>
34    // We bind the domain parameters as follows:
35    // U := UI,  L := LOGIC, D := DATA
36   @Domain("LOGIC<UI,LOGIC,DATA>")IniFile iniFile;
37   ...
38 }
```

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

using a typechecking tool, which we refer to as ArchCheckJ. We illustrate the ownership domain annotations using code examples from the subject system (Fig. 1).

Domain declaration. The architectural extractor must declare a domain before using it. For example, the statement `@Domains` declares the domains `owned` and `PARAGS`:

```
@Domains({"owned", "PARAGS"})
```

Domains are declared on a class, but fresh domains are created for each instance of that class. For example, the class `IniFile` declares and uses the private domain `owned` (Line 1). Similarly, the class `IniParagraph` declares its own, different, private domain `owned` (Line 15).

Domain use. Each object is assigned to a single ownership domain that does not change at runtime, The architectural extractor indicates the domain of an object by annotating each reference to that object in the program. For example, both of the types `IniFile` and `IniParagraph` declare a reference of type `InputFile`, namely `f` (Line 11 and Line 18, respectively) in the domain `L`.

Object hierarchy. The annotations define two kinds of object hierarchy:

- **Strict encapsulation:** A private domain provides *strict encapsulation* and makes an object strictly *owned by* another. Then, 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 annotation system currently supports declaring one private domain per class, which by convention is called `owned`. For example, `IniFile` stores a `Hashtable` in its private domain, `owned` (Line 6).
- **Logical containment:** A public domain provides *logical containment* and makes an object conceptually *part of* another. Having access to an object gives the ability to access objects inside all its public domains. The annotation system supports declaring many public domains per class. For example, `IniFile` declares a public domain, `PARAGS`, to hold `para` objects (Line 9).

Domain parameters. Domain parameters on a type allow objects to share state. The architectural extractor declares domain parameters using the `@DomainParams` annotation. When instantiating a type that declares domain parameters, the client code must bind the formal domain parameters to some other domains in scope.

For example, the `Hashtable` class is parametric in two ways. First, `Hashtable` is generic in the types of the keys and values stored in the `Hashtable`, so it takes two generic type parameters, `TK` and `TV` for keys and values, respectively (Line 25). `Hashtable` also takes two formal domain parameters, `DOMKEY` and `DOMVALUE` (Line 23) for the domains of the keys and the values, respectively. Whenever a `Hashtable` is used, the formal domain parameters must be bound to other domains in scope. For example, the architectural extractor binds `DOMKEY` to `shared` and `DOMVALUE` to the `PARAGS` domain inside `IniFile`.

Similarly, the class `IniParagraph` takes the domain parameters `U`, `L` and `D` (Line 16). This way, it can reference objects in the `L` domain, such as `InputFile` (Line 18).

Domain inheritance. Domain parameters on a type are inherited by its sub-types. So the annotations bind a type's formal domain parameters to domain parameters of the super-types. For example, the class `Hashtable` inherits from `Map`. So `Hashtable` binds its domain parameters `DOMKEY` and `DOMVALUE` to `Map`'s domain parameters, using the `@DomainInherits` annotation:

```
@DomainInherits({ "Map<DOMKEY, DOMVALUE>" })
```

Special annotations. There are additional annotations that add expressiveness to the type system [9]: `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. One ownership domain can temporarily lend an object to another and ensure that the second domain does not create persistent references to the object by marking it as `lent`. 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. These special annotations are part of the type system, so they need not be declared.

Library code. The tool supports adding partial annotations to the external libraries as the Java Standard Library, using external XML files called AliasXML [10].

2.2 Object graph extraction

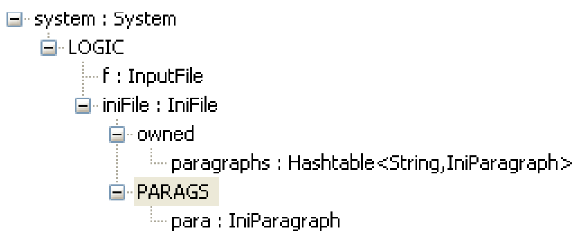
SCHOLIA uses a static analysis to abstractly interpret the program with ownership domain annotations, to extract a hierarchical object graph that provides architectural abstraction by ownership hierarchy and by types, the Ownership Object Graph (OOG). The analysis to construct an OOG requires a root class, namely `System` in the example (Fig. 1). Fig. 2 shows the top-level OOG for the above example (Fig. 1). The OOG shows that both objects `iniFile:IniFile` and `para:IniParagraph` reference the same `f:InputFile` object. The hierarchical representation allows collapsing or expanding objects to control the level of visual detail. In the rest of this paper, we refer to the tool for extracting and viewing OOGs as ArchRecJ.

The visualization uses box nesting to indicate containment of objects inside domains and domains inside objects (Fig. 2). Dashed-border, white-filled boxes represent domains. Solid-filled boxes represent objects. Solid edges represent field references. An object labeled `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". A private domain has a thick, dashed border; a public domain, a thin one. A (+) symbol on an object or a domain indicates that it has a collapsed substructure.

2.3 Subject System

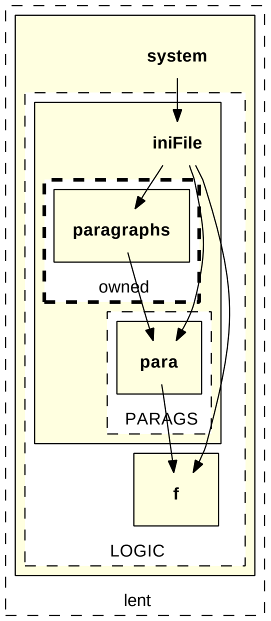
As our subject system, we chose Pathway Express, which is part of the Onto-Tools developed in the Intelligent System and Bioinformatics Laboratory at Wayne State University [11]. Pathway Express finds, builds, and displays a graphical representation of gene interactions, and has more than a thousand users spread across several university research groups who study bio-informatics.

Development on Pathway Express has been ongoing since 2002. The system had been developed by graduate students who had since graduated and were no longer available. The system, however, is still actively maintained by a number of graduate students. We chose this system because we had access to

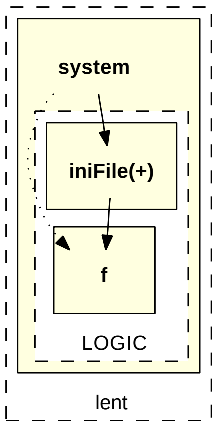


```
system : System
  LOGIC
    f : InputFile
    iniFile : IniFile
      owned
        paragraphs : Hashtable<String,IniParagraph>
      PARAGS
        para : IniParagraph
```

(a) Ownership tree for the OOG, from the root object.



(b) Expanded OOG.



(c) Collapsed OOG.

Figure 2: OOG for the above example (Fig. 1).

one of its maintainers, to help us refine the OOGs and evaluate the usefulness of the OOG during software evolution. In the rest of this paper, we refer to Pathway Express as P-X.

P-X is an object-oriented web application implemented in J2EE that consists of 163 classes, 9 interfaces, and 30 packages, for a total of 36,000 lines of Java code, excluding libraries [12]. The maintainer set up a standalone Java project (with all the associated libraries) in the Eclipse development environment for us to analyze.

3 Extraction of the Architecture

There are multiple ways of annotating a system, where different annotations produce different OOGs. For example, for the same system, the use of different top-level domains will generate different OOGs, and different public domains will express different architectural hierarchies.

The success criteria for the architectural extractor include: having annotations that generate the least number of warnings, making the fewest changes to the code, and expending the least effort to add the annotations and extract the OOGs.

We organized the study into the following phases:

- Pre-processing (cleaning up the code, etc.);
- Deciding on the organization of the objects into domains;
- Propagating the initial set of annotations using a defaulting tool (building the map, etc.);
- Running the typechecker and manually fixing the annotations warnings;
- Extracting OOGs and refining them.

3.1 Pre-processing

Much of the P-X code was written prior to Java 1.5, and did not use generics. So we cleaned up the code, as a first step to increase the precision of the extracted OOG. As part of the code cleanup, we refactored the code to use generics. Non-generic code tends to suffer from various problems. For example, reading an object from a non-generic collection requires a cast, which may cause a runtime exception if objects of the wrong type are inserted into the collection.

There were many instances of containers from the Java Standard Library which used "raw types". The Eclipse compiler generates warnings when the code uses raw types, which made it easy to find where we needed to change the code. The architectural extractor inferred generic types using Eclipse's refactoring tool. In some cases, Eclipse failed to infer a type, so the architectural extractor did the changes manually. As a benefit of adding generics, he was able to remove unnecessary type casts, using the Eclipse code cleanup tool.

In some cases, the architectural extractor introduced generic type parameters [13]. Although there is tool support for this, he did this mostly manually. For example, the class `MultiValuedSetHashtable` implemented the raw type `Map`. In fact, `Map`, from the Java Standard Library, is parameterized as `Map<K,V>`, where `K` and `V` are the type parameters for the keys and values, respectively. So, we genericized the super-type of `MultiValuedSetHashtable` as `Map<K, List<V>>`, because `MultiValuedSetHashtable` maps key objects to a list of values. Once we did that, we changed the signatures of the implementation of the methods in the interface, as well as some of the corresponding fields, to no longer use raw types.

In the process of adding generic types, we looked for method signatures that used `Object`, and tried to convert those occurrences to a more specific type. In some cases, some methods were implementing methods from the Java Standard Library, but did not have the `@Override` annotation. In a few cases, we inadvertently changed their signature, which meant that these methods were no longer overriding methods, thus changing the meaning of the program. We then realized we can use the Eclipse clean-up tool to add any missing `@Override` annotations. As a result, if a method has the `@Override` annotation and compiled correctly before our changes, the Eclipse compiler will generate an error if our changes made the signatures to no longer match. Also, the `@Override` annotation will be useful for program comprehension.

3.2 Deciding on the organization of the objects into domains

In this step, the architectural extractor determined the top-level domains of the system, and the mapping of objects to domains.

3.2.1 Determining the top-level domains

From an initial meeting with the system maintainer, the architectural extractor decided to organize the application into three tiers: User Interface, Logic and Data. He represented each of the top-level tiers using the top-level domains, `UI`, `LOGIC`, and `DATA`, respectively.

3.2.2 Mapping objects to domains

Ideally, the system maintainers provide the mapping of objects to domains or refer the architectural extractors to available documentation, such as a description of the system structure or any available diagrams.

The architectural extractor aimed to reduce the burden on the maintainer, so in this case, he did not ask the maintainer to provide the mapping. Furthermore, the architectural extractor did not have documentation available for P-X. The architectural extractor determined the mapping himself, by relying on the package or the type names, to determine to which domain the instances of the type belong. For example, he used the package name `oe.standalone.client.pe.gui.oe` as a hint that the instances of the types declared in that package, such as `DefaultFunctionBar`, are in the `UI` domain. Similarly, based on the type name, he placed instances of the type `DBConnectionManager` in the `LOGIC` domain, and instances of `OntoToolData` in the `DATA` domain.

3.2.3 Determining the formal domain parameters

Once the architectural extractor determined the three top-level domains, `UI`, `LOGIC`, and `DATA`, he introduced the corresponding domain parameters `U`, `L` and `D`, respectively, on the various types, to allow objects to share their state with other objects.

3.3 Propagating the initial set of annotations

3.3.1 Earlier defaulting tool

Abi-Antoun [10] had previously developed a defaulting tool to reduce the annotation burden. The defaulting tool added mostly boilerplate annotations. For example, it added default domain declarations with the private domain, `owned`, `@Domains("owned")` and a blank list of domain parameters `@DomainParams({" "})` for all types. It annotated private fields and return types of private methods with `owned`, and variables of type `String` with `shared`. It also annotated local variables and method formal parameters with `lent`.

There were a few issues with the earlier defaulting tool. For example, it did not generate domain parameters. So the architectural extractor had to provide these parameters. Moreover, it did not add the `@DomainInherits` annotation. As a result, the architectural extractor had to manually determine the super-types and add them to the `@DomainInherits` clause.

Fully automated inference of ownership annotations is a hard problem. Many of the static inference techniques adopt a restrictive notion of ownership, infer only strictly encapsulated objects and unaliased objects, do not map their results back to a type system, do not infer domain parameters or infer imprecise long lists of domain parameters, and finally, do not scale. So we decided to build a heuristics-based tool to add more of the annotations, and reduce the annotation burden.

3.3.2 ArchDefault tool

We enhanced the previous defaulting tool, hereafter called ArchDefault [14], to add more of the annotations automatically. ArchDefault takes two inputs. The first input is a list of formal domain parameters that will be propagated to type declarations. For P-X, it was the three parameters `U`, `L` and `D`. The second input is a list of Type-Domain-HasParameters tuples. Each tuple has the following fields:

- `Type`: the fully qualified name of a type, e.g., “x.y.z.Foo”;
- `Domain`: the name of the domain parameter in which to place all instances of this type, e.g., “D”;

- `HasParameters`: a boolean flag. If true, it means that ArchDefault should provide the domain parameters.

For example, the Type-Domain-HasParameters tuple (type="oe.standalone.util.IniParagraph", domain="L", hasParameters="true") causes ArchDefault to do two things. First, it adds `@DomainParams` to the type declaration of `IniParagraph` (Fig. 1). Second, when ArchDefault encounters a field, local variable, or formal method parameter of type `IniParagraph`, it generates the following `@Domain` annotation:

```
@Domain("L<U,L,D>") IniParagraph iniParagraph;
```

With the ArchDefault map, we are implicitly mapping, as a first approximation of the annotations, from types to domains (Table 1). For example, the type `IniParagraph` is mapped to `Context::L`. In this map, `Context` refers to the enclosing type that holds references of type `IniParagraph`. The reason this map is a first-level approximation is that it maps *types* to domains. To extract an OOG, we need to map *objects* to domains. As a result, we exclude from the map instances of the types that may often occur in multiple domains, such as `Vector`, `ArrayList`, `Hashtable`.

Table 1: Mapping of types to domains for the above example (Fig. 1).

Declared Type	**Domain**
String	::shared
IniFile	Context::L
IniParagraph	Context::L
InputFile	Context::L
System	Context::L

This works because ArchDefault propagates domain parameters to almost all of the types in the system, by adding to them the `@DomainParams` annotation, with entries for all the domain parameters declared in the map (`U`, `L` and `D` in the above example). So, when the domain parameters occur in the `@Domain` annotation on a field or variable declaration (`iniParagraph` in the above example), they will be in the scope of the enclosing class, and the annotations will typecheck. Also, there is no harm in declaring domain parameters on a class and not using them, except for the extra clutter in the code. The static analysis that extracts OOGs substitutes actual domains for formal domain parameters, and ensures that a domain in the OOG will show all the objects that could possibly exist in that domain, in any program run.

The ArchDefault map can also place objects in the `shared` domain, such as instances of `String`, `Font`, or `Color`. To do so, one adds to the map a tuple Type-Domain-HasParameters such as (type="java.awt.String", domain="shared", hasParameters="false"). ArchDefault will then generate the following annotation in the code, when it encounters a field, local variable, or method parameter of type `String` :

```
@Domain("shared") String filename;
```

In addition, ArchDefault populates the values in the `@DomainInherits` annotation as

```
@DomainInherits({"Supertype1<DomainParameters>, ... "})
```

From the type declaration, ArchDefault determines the set of all the direct super-types of the current type. Then, for each super-type, the tool determines if the super-type takes domain parameters from the map, using the same heuristics it used for finding domain parameters. ArchDefault then adds the super-types and their corresponding domain parameters to the `@DomainInherits` annotation.

3.3.3 Reducing the map size

To reduce the burden on the architectural extractor, it will be ideal if the map had fewer entries than the total number of types used in the program. To do so, we rely on the notion of sub-typing. Where applicable, it is enough to add to the map an entry for just the super-types. Then, ArchDefault applies the same annotation to all the sub-types of the super-type.

For each type T, ArchDefault tries to find the Type-Domain entry for T. If the entry for T exists, ArchDefault uses it. If the entry for T does not exist, ArchDefault looks for an entry for type T', where T is a sub-type of T'.

3.3.4 Applying ArchDefault

The architectural extractor built the P-X map, which consists of 133 Type-Domain entries divided as follow: 58 types in `D`, 20 types in `L`, 32 types in `U`, and 23 types in `shared`. Because P-X does not have a rich inheritance hierarchy (the maximum depth of the inheritance tree was 3), the map contains 110 entries for the custom types, which is not much less than the total number of classes (163 classes). The architectural extractor spent around 5 hours between creating the map and determining the domain for each types in the map to annotate the system objects later. Then, he used ArchDefault with the custom map to propagate most of the initial annotations to the code automatically.

3.3.5 Adding AliasXML files

Because P-X uses the Java Standard Library, the architectural extractor reused existing AliasXML files, from previously annotated systems, to speed up the annotation process. For example, he reused the AliasXML files for `StringBuffer`, `Hashtable`, `Iterator`, etc.

In some cases, the architectural extractor had to modify existing AliasXML files and add annotations for any code that is actually in use. For example, in the AliasXML file for the type `StringBuffer`, the method `toString()` was not previously used, so it was not annotated. The architectural extractor added the `shared` domain to the return type of the method. He also generated additional AliasXML files for `LinkedList`, `SortedSet`, `StringTokenizer`, etc. For P-X, we reused 36 files from previous studies and generated 49 new files, for a total of 85 AliasXML files.

3.4 Adding annotations

3.4.1 Using ArchCheckJ

After the architectural extractor used ArchDefault to propagate the initial set of annotations and added the AliasXML files, and since ArchDefault is not a smart inference tool, the architectural extractor had to fix the annotations manually. He relied mostly on ArchCheckJ to guide the annotation process. ArchCheckJ validates the added annotations, finds missing annotations, and warns about inconsistent ones.

He modified the annotations manually to fix the annotation warnings, in a systematic and iterative way by:

1. Running ArchCheckJ to check the annotations;
2. Addressing warnings of specific type, starting with the high-priority ones;
3. Going back to the first step, until the warnings could no longer be addressed.

The architectural extractor addressed the annotation warnings in the following order:

3.4.2 Refactoring problematic statements

If a type declares formal domain parameters, the code must supply actual domains when creating an object of that type or when casting references of different types. However, Java 1.5 does not allow annotations on object allocation or cast expressions. So the annotation system requires extracting local variables, and adding annotations to them. For example, the architectural extractor used the Eclipse refactoring tool to extract a local variable from the expression in:

```
return new PEOntoExpressGraphCP(odt, gLCP);
```

and annotating the local variable as follows:

```
@Domain("U<U,L,D>")PEOntoExpressGraphCP gLCP = new ...;
return gLCP;
```

3.4.3 Adding missing annotations

ArchDefault failed to add annotations to some references, because their types or super-types were missing from the map. Also, the map does not include the Java Standard Library containers like `List` and `Hashtable`, because different instances of these types are often in different domains. For example, you can find a `List` object in `U`, and another `List` in `L`. So, the architectural extractor had to supply those annotations manually.

3.4.4 Adding AliasXML files

Without AliasXML files, the typechecker does not know that containers like `List`, `Hashtable`, etc., take domain parameters, so it does not complain about missing domain parameters. Once we add the AliasXML files, the typechecker raises additional warnings, because all the formal domain parameters on containers must now be bound, such as:

```
@Domain("owned") Hashtable<String, IniParagraph> paragraphs;
```

The architectural extractor had to provide the actual domains for the keys and the values domain parameters (Fig. 1).

3.4.5 Fixing array annotation warnings

References of an array type require two annotations. First, the architectural extractor must specify the domain for the array object itself. Second, he must specify the domain of the array elements. For example, ArchDefault annotated an array of String objects as follows:

```
@Domain("lent") String[] chipVec;
```

the architectural extractor fixed it by providing the `shared` domain for the array elements:

```
@Domain("lent[shared]") String[] chipVec;
```

3.4.6 Removing unused domain parameters

The architectural extractor found that some classes had domain parameters which were not really needed, and in fact, caused annotations warnings when the class was used by static methods. So, he simply removed the domain parameter declarations and the domain parameters from the corresponding objects instantiation. For example, he removed the formal domain parameters from the type declaration for `OutputFile` and the actual domains from any instantiations of that type. As a consequence of removing domain parameters from some types, the architectural extractor removed unnecessary `@DomainInherits` annotations.

3.4.7 Remaining warnings

The architectural extractor fixed most of the annotation warnings. The 469 remaining warnings are either due to inherent expressiveness challenges in the type system, bugs in the tools that we are currently fixing or design issues in P-X.

3.5 Extracting and Refining OOGs

3.5.1 Determining the root class

The architectural extractor had to determine the root class, from which ArchRecJ starts analyzing the annotated code. There were around 20 `static void main` methods scattered in different classes of the system. Most of these methods were used as test harnesses for unit testing. We found two likely classes, `PathwayExpress` and `OntoToolsApp`.

We first chose `OntoToolsApp` as the root class. However, we realized that the extracted OOG was missing one of the top-level domains and many objects in that top-level domain.

The architectural extractor was not a J2EE web application expert, but he knew enough about web development, namely that each client-server application has two main components. The server-side component runs always on the host server. In J2EE, this component is usually an instance of the type `Servlet` or any sub-type thereof, such as `HttpServlet`, that responds to the client's requests. The architectural extractor found the class `PathwayExpressServlet` which sub-classes `HttpServlet`, as a server-side type.

The client-side component is typically a Java applet which requests services from the server and displays the results in the client's web browser. For P-X, the architectural extractor found the class `PEInputApplet`, which represents the client-side component and instantiates `OntoToolsApp`, which in turn initializes several of the core objects in the system.

P-X is a web-based system. The instantiation of objects on the server is controlled by various configuration files, which are interpreted by the server infrastructure. But our static analysis does not know about these files. So, the architectural extractor had to simulate the initialization of the system and provide this information explicitly to the static analysis.

The goal was to extract an architecture of the entire, complete system, so we wanted to include both client- and server-side objects. So, the architectural extractor created a top-level class, `Main`, which instantiates both `PathwayExpressServlet` and `PEInputApplet` in order to include all of the system's objects.

3.5.2 Extracting initial OOGs

After the architectural extractor fixed most of the annotation warnings, he used ArchRecJ to extract initial OOGs. The initial OOG displayed many objects in the top-level domains. Some of these objects were low-level, implementation details. So, the architectural extractor needed to refine the OOGs, by slightly adjusting the annotations. The initial OOGs were not at an acceptable level of abstraction that we could show them to the maintainer to get his feedback.

3.5.3 Refining initial OOGs

The architectural extractor controls the abstraction by ownership hierarchy in the OOG using the annotations. Namely, he can push low-level objects underneath more architecturally significant objects.

Make an object *owned by* another. He pushed uninteresting objects below more interesting ones. For example, he moved the objects of type `PEGraphClass` from the top-level domain `D` to the domain `owned` inside the class `PEGraphTableData`.

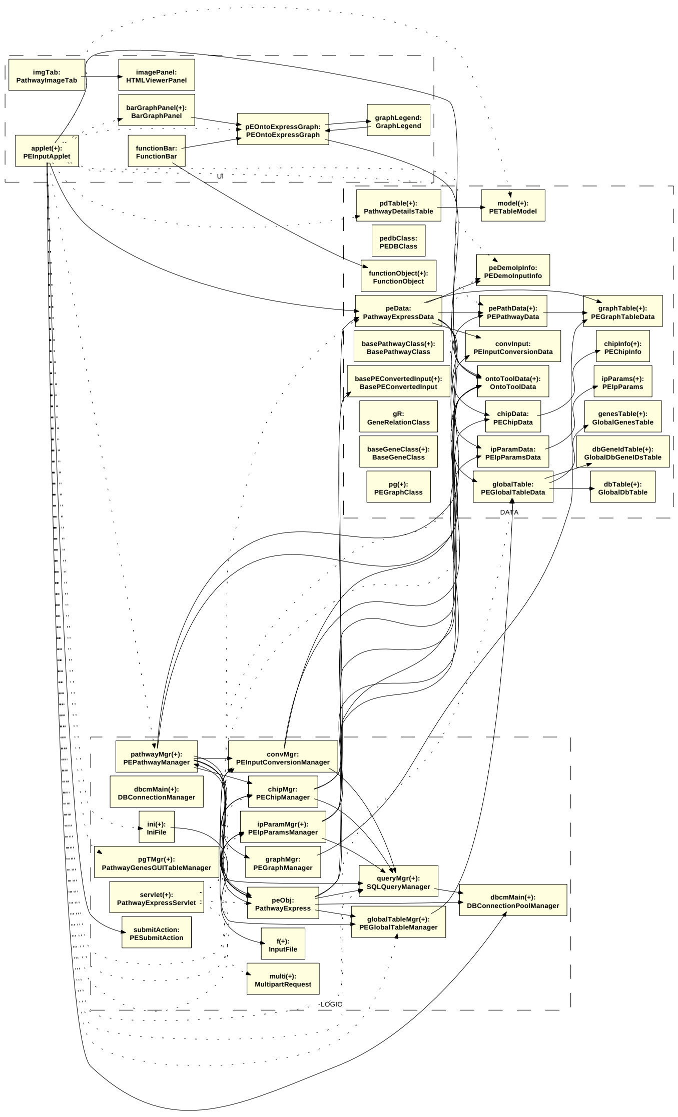
Make an object *part of* another. The architectural extractor made an object part of another, by declaring a public domain, and placing the object in it. For example, he created the public domain `PARAGS` on class `IniFile`, then moved the object `para:IniParagraph` from the top-level domain `DATA` into the `PARAGS` domain (Fig. 1).

Objects in public domains are accessible to the outside. For example, the architectural extractor declared the public domain `DB` in the type `PETableModel` to hold objects of type `Row`. It was not possible to place these objects in the `owned` domain, because several public methods on `PETableModel` returned objects of type `Row`.

4 Evaluation of the Extracted Architecture

The goal of showing the OOG to the maintainer was to determine if he understood the abstractions we generated. In particular, we wanted him to confirm the placement of objects into domains since the maintainer did not verify the ArchDefault map. Where the extracted OOG did not reflect the maintainer's mental model of the system, we wanted a list of refinements of the OOG, in order to refine it further.

The success criteria for the maintainers include having an OOG that is sufficiently abstract and not too cluttered, and one that reflects their mental model of the system. Ideally, they will be able to use the extracted OOGs as a form of documentation for new developers learning the system.


imgTab: PathwayImageTab
imagePanel: HTMLViewerPanel
barGraphPanel(+): BarGraphPanel
pEOntoExpressGraph: PEOntoExpressGraph
graphLegend: GraphLegend
applet(+): PEInputApplet
functionBar: FunctionBar
UI
pdTable(+): PathwayDetailsTable
model(+): PETableModel
pedbClass: PEDBClass
functionObject(+): FunctionObject
peDemoIpInfo: PEDemoInputInfo
peData: PathwayExpressData
pePathData(+): PEPathwayData
graphTable(+): PEGraphTableData
basePathwayClass(+): BasePathwayClass
convInput: PEInputConversionData
chipInfo(+): PEChipInfo
basePEConvertedInput(+): BasePEConvertedInput
ontoToolData(+): OntoToolData
ipParams(+): PEIpParams
gR: GeneRelationClass
chipData: PEChipData
genesTable(+): GlobalGenesTable
baseGeneClass(+): BaseGeneClass
ipParamData: PEIpParamsData
dbGeneIdTable(+): GlobalDbGeneIDsTable
pg(+): PEGraphClass
globalTable: PEGlobalTableData
dbTable(+): GlobalDbTable
DATA
pathwayMgr(+): PEPathwayManager
convMgr: PEInputConversionManager
dbcmMain(+): DBConnectionManager
chipMgr: PEChipManager
ini(+): IniFile
ipParamMgr(+): PEIpParamsManager
pgTMgr(+): PathwayGenesGUITableManager
graphMgr: PEGraphManager
queryMgr(+): SQLQueryManager
servlet(+): PathwayExpressServlet
peObj: PathwayExpress
dbcmMain(+): DBConnectionPoolManager
globalTableMgr(+): PEGlobalTableManager
submitAction: PESubmitAction
f(+): InputFile
multi(+): MultipartRequest
LOGIC
Figure 3: The final extracted OOG of P-X.

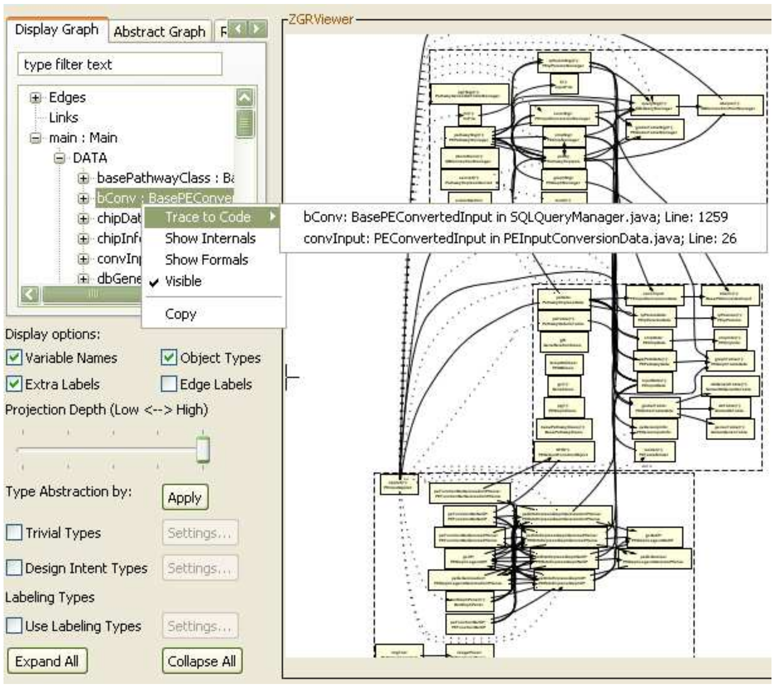
4.1 Method

4.1.1 Participant.

The participant in the evaluation of the OOG was one of the current maintainers of the P-X system, and a graduate student at Wayne State University. He was already familiar with the Eclipse environment.

4.1.2 Tools and Instrumentation.

We gave the participant ArchRecJ, which allows for interactive navigation of the OOG (Fig. 4). We used a video screen capture software to track the maintainer's navigation of the OOG using ArchRecJ. We also



bConv: BasePEConvertedInput in SQLQueryManager.java; Line: 1259
convInput: PEConvertedInput in PEInputConversionData.java; Line: 26

Display options:
✓ Variable Names ✓ Object Types
✓ Extra Labels ☐ Edge Labels
Projection Depth (Low <--> High)
Type Abstraction by:
☐ Trivial Types
☐ Design Intent Types
Labeling Types
☐ Use Labeling Types

Figure 4: Snapshot of ArchRecJ with the P-X OOG.

recorded audio of the discussion.

The OOG extraction and viewing tool (Fig. 4) has the following features:

- A graph-based view of the OOG, showing the objects and the edges between them.
- The ability to expand or collapse objects to see the nested domains and objects inside them.
- A tree-view of the ownership hierarchy of the system, which also allows viewing the sub-structure of an object.
- A trace to code feature: he can select object or edge, either in the tree view or the plain view, and go to the corresponding lines of code;
- A feature to search for an object or an edge: he can search for an object, by name or by type, as an exact or partial match. The matching elements are highlighted in the tree.

4.1.3 Questions.

The architectural extractor asked the maintainer to answer the following questions:

- Q1–Does object X of type T belong to tier A? And if not, to which tier does it belong?

- Q2–Which objects, do you think, are useful and helpful for code modifications to see at the top-level of the OOG?
- Q3–Are there any missing objects from the top-level of the OOG? or from the rest of the OOG?

4.1.4 Procedure.

The architectural extractor launched ArchRecJ and displayed the extracted OOG. Then, he gave the maintainer a quick tutorial explaining the tool and its features. The architectural extractor asked the maintainer all the questions at the beginning of the meeting, to set his expectations for the meeting. For each of the above questions, the maintainer went through each tier, and each object in the tier.

4.2 Results

Feedback on the tools. At the beginning of the interview, the maintainer told the architectural extractor that he cannot give accurate information about whether an object X is in tier A and that he did not know all the objects in the system.

The maintainer used several ArchRecJ features. For example, he used the traceability to code (Fig. 4) to trace from objects or edges in the OOG to the corresponding lines of code. He also found the tree view to be a useful way of navigating all the objects of the OOG. ArchRecJ is an Eclipse plugin. So, the maintainer was also able to use the features available in Eclipse to navigate the code. For example, the maintainer used the search feature to find specific types. He used the Outline view to see the structure of some type declarations, and the Type Hierarchy view to traverse the inheritance hierarchy.

The meeting with the maintainer was divided into three parts. In each part, the maintainer answered one of the architectural extractor's questions, starting with one tier at a time, then going through all objects of that tier, one at a time.

Q1– Does object X of type T belong to tier A? And if not, to which tier does it belong?

The maintainer navigated all objects in the tree and agreed on the placement of most of the objects into tiers. In a few cases, the maintainer suggested moving some objects from one domain to another. For example, he suggested moving the object `model:PETableModel` from `UI` to `DATA`. In other cases, the maintainer was unsure in which domain an object should be. For example, `f:InputFile` was in `DATA`, but it could also be in `LOGIC`. For some cases, he was not able to determine the domain of an object, so he traced to the code and found the type of that object and navigated the type outline to decide in which tier it should be.

Q2– Which objects, do you think, are useful and helpful for code modifications to see at the top-level of the OOG?

For this question, the maintainer navigated all the top-level objects in the OOG carefully, and other objects quickly, because at this point, he knew most of the objects and had become familiar with the tool. In a few cases, the maintainer pointed to objects that were not useful to show in the top-level domains. For example, he suggested merging all objects that have `Table` in their declared types, like `pdTable: PathwayDetailsTable`, `ipIdGenesTable: InputIdGenesTable` and `model: PETableModel`. Surprisingly, he could not find a common super-type for all the types of these objects. As a result, the architectural extractor could not use abstraction by types to accomplish this merging.

Q3– Are there any missing objects from the top-level domains of the OOG? or from the rest of the OOG?

The maintainer noticed that the OOG was missing important objects that display the results of operations in the system. He thought the type was called `ResultFrame`, but he did not find such a type in the code. Finally, he found a method called `displayGUI()` in the class `PEGUIManager` which is used to display results. Then he decided that the objects of type `PEGUIManager` must be in `UI` and `LOGIC`.

The maintainer thought that some objects could be made conceptually part of others. For example, he wondered why the object `lf:LoginFrame` was not inside a domain of the object `inputFrame:PEInputFrame`.

While the maintainer navigated the OOG, he asked questions about some edges between some objects. For example, he wondered about an edge between `lf:LoginFrame` and the object `otApp:OntoToolsApp` and what it meant. The architectural extractor selected the edge and traced to the code to understand the meaning of this edge, which turned out to be a field declaration.

The maintainer was surprised by the existence of some edges in the system and wondered about the absence of others. For example, he asked why there was no edge between `lf:LoginFrame` and `inputFrame:PEInputFrame`. So, he selected `inputFrame:PEInputFrame` in the OOG, traced to the code and found that the edge was missing because there was no field declaration in the code.

Table 2: Classification of OOG refinements requested by maintainer.

OOG Refinement	Requested	Completed
R1 Move object between sibling domains	19	18
R2 Abstract low-level object	40	21
R3 Move object to higher-level domain	0	0
R4 Collapse related instances of subtypes	20	14

4.3 Additional refinement of the OOG

4.3.1 Analysis of maintainer feedback

We transcribed the thinkaloud recordings and screen capture video into action logs, consisting of a total of 319 lines. Next, we used qualitative protocol analysis. From a previous case study [15], we had classified the list of possible refinements of an OOG. We coded what the maintainer asked for using this model (Table 2).

4.3.2 OOG refinements

One can refine an OOG as follows:

1. Move an object between sibling domains, e.g., between the top-level domains `UI` and `LOGIC`;
2. Abstract a low-level object from a top-level domain by pushing it underneath a more architectural object, i.e.:
   (a) Make an object conceptually *part of* another object;
   (b) Make an object *owned by* another object;

   However, the maintainer typically does not know whether to use (2a) or (2b). It is often the architectural extractor who decides whether to use a private or a public domain;
3. Move an object that was previously abstracted from a lower-level to a higher-level domain;
4. Collapse related instances of subtypes, using abstraction by types;
5. Other, such as edit an object's label, or make an object `shared`.

Interestingly, the maintainer requested that many objects be moved around the top-level domains (1). This is not surprising since the architectural extractor did not ask the maintainer to validate the map before proceeding with the annotation process. The maintainer did not recommend any objects be moved up to the top-level domains (3). This is perhaps understandable, since the top-level domains were still relatively too cluttered.

Refining an extracted OOG involves changing the annotations, incrementally and in a localized manner. When the extracted OOG does not match the maintainer's mental model, the architectural extractor must identify the cases where the cause of the discrepancy is an incorrect ownership relationship, change the annotations in the code consistently to reflect the corrected ownership relationship, then re-extract the OOG.

4.3.3 Changing annotations

The architectural extractor modified the annotations in the code to reflect the changes that the maintainer had asked for. He moved objects from one domain to another, pushed objects underneath others, etc. In a few cases, he could not refine the OOG because the approach does not support the required change.

4.3.4 Using abstraction by types

Abstraction by design intent types can further collapse objects in a given domain based on their declared types [10]. ArchRecJ provides the option to select and add the types to be used for abstraction by types. To implement a change request from the maintainer, the architectural extractor added the type `FunctionBar` to the list of design intent types. As a result, ArchRecJ grouped the objects of types `FunctionBar`, `PEFunctionBar`, and `PEFunctionBarGammaPValue`, in the top-level domain `UI`.

We show the number of objects in the top-level domains, at the meeting with the maintainer and after we refined the OOG (Table 3). We were able to reduce the number of visible objects in the top-level domains (from 68 to 46). A rule of thumb in architectural documentation is to have 5 to 7 components per tier [16]. We still, however, have a high number of objects in the `DATA`, due to our inability to apply additional abstraction by types (See Section 5.4).

5 Discussion and Lessons Learned

5.1 Quantitative Data

5.1.1 Effort data

Table 4 shows the time effort spent in this study. The architectural extractor spent 36 hours adding annotations and fixing warnings, which is consistent with our previous estimate of the annotation rate of

Table 3: Number of objects in the top-level domains.

Domain	At meeting	After meeting
`UI`	29	7
`LOGIC`	12	16
`DATA`	27	23
Total	68	46

Table 4: Time to extract the OOG.

Phase	**Effort(Hours)**	**Percent**
Adding annotations and extracting OOGs	31	69%
Building the ArchDefault map	5	11%
Refining the OOG on our own	5	11%
Meeting with maintainer	1	2%
Refining the OOG after meeting	3	7%
Total	45	100%

1-hour/KLOC [10]. The architectural extractor spent around 5 hours on his own refining the OOG. He then spent another 3 hours refining the OOG after the meeting with the maintainer. The total time of the meeting with the maintainer lasted about an hour and a quarter. This is 2.8% of the total time, excluding an initial meeting to launch the architectural extraction project. So the architectural extraction approach does not require significant and continuous involvement from the part of the maintainers.

It seems that the typechecker provided much of the guidance to the architectural extractor, who did not have to constantly ask the maintainer. Also, it is relatively easy to change the annotations after the fact. Indeed, after meeting with the maintainer, the architectural extractor spent much less time refining the OOG than before the meeting.

5.1.2 Metrics.

We built a simple program to analyze the annotations we added (Table 5). We used these metrics to judge the quality of the annotations and the quality of the extracted OOGs. For example, a high proportion of `shared` annotations meant we were not reasoning about many objects in the system.

In addition, we typically do not show the `shared` domain in an OOG. In P-X, there are 6,320 fields or variables of a reference type which have annotations. 17% of the objects are `shared`, which is a high but not unreasonable proportion that is due to the excessive use of `String` objects in the P-X code.

Table 5: Domains Statistics.

Annotation	**Frequency**	**Percent**
`U`	196	3%
`L`	122	2%
`D`	846	14%
`owned`	564	9%
`shared`	1,140	17%
`unique`	284	5%
`lent`	3,146	49%
Other public domains	22	1%
Total	6,320	100%

5.2 Effectiveness of the ArchDefault tool

Using ArchDefault in the beginning helped the architectural extractor start with fewer warnings to resolve, and reach an acceptable level of warnings more quickly in order to refine the OOG with the maintainer. Thus, it was possible to use a tool to propagate many of the annotations automatically. ArchDefault helped the most in propagating domain parameters and domain parameter inheritance. The architectural extractor still had to manually fix annotation warnings.

During the study, we thought of a few improvements to ArchDefault. For instance, it would be nice to map all instances of all the types defined in a package to one domain. For example, the types `Font`, `Image` and `Point` are in the `java.awt` package. Instead of listing many fully qualified type names, the map will have a Type-Domain-HasParameters tuple with a regular expression, (type="java.awt.*", domain="shared", HasParameters="false").

In addition, since legacy Java code uses heavily the various collection types from the Java Standard Library, it would be nice to make ArchDefault insert the inner annotation and any nested domain parameters for the actual domain for the elements of the collection (the `PARAGS<U,L,D>` annotation on `Hashtable` in Fig. 1). This requires having the generic types on the containers, so ArchDefault can lookup in the map the domain of the contained elements (`IniParagraph` in Fig. 1).

5.3 Lessons learned from adding annotations

We learned that code that cannot be easily genericized cannot be easily annotated with ownership domains. This should come as no surprise since some ownership type systems combine ownership types and generic types [17]. Similarly, it is not a good sign if the only generic types that can be added are fairly imprecise ones, such as `Object`, `Serializable` or `Cloneable`. In fact, the abstraction by types feature in the OOG uses these types as trivial types. Having these types in the code can also lead to very imprecise edges, i.e., more edges in the OOG than are needed for soundness.

The annotations highlighted many instances of "representation exposure". We added warnings and TODOs in the code as reminders for the maintainers to investigate and fix them (Fig. 5).

5.4 Lessons learned from extracting OOGs

Extracting OOGs helped us identify several design smells in the P-X system.

Lack of rich inheritance hierarchy. We could not apply the abstraction by types as much as the

```
@Domains({ "owned" })
@DomainParams({ "U", "L", "D" })
abstract class AbstractStandaloneInput ... {
  protected @Domain("owned<D>") List<Serializable> references;
  ...
  public @Domain("unique<D>") List<Serializable> getReferences() {
     //TODO: Return a copy of the owned field.
     return references;
  }
  ...
}
```

Figure 5: Example of adding a warning about representation exposure.

Table 6: Mapping of references to OOG for the above example (Fig. 1).

Enclosing Type	**Reference**	**Domain**	**Declared Type**	**Path in OOG**, i.e., (domain.object)*
IniFile	filename	::shared	String	shared
IniFile	paragraphs	IniFile::owned	Hashtable	lent.system.LOGIC.iniFile.owned.paragraphs
IniFile	para	IniFile::PARAGS	IniParagraph	lent.system.LOGIC.iniFile.PARAGS.para
IniFile	f	IniFile::L	InputFile	lent.system.LOGIC.f
IniParagraph	f	IniParagraph::L	InputFile	lent.system.LOGIC.f
System	iniFile	System::LOGIC	IniFile	lent.system.LOGIC.iniFile

maintainer wanted us to. Upon further investigation, we found that many related classes do not share a common superclass. This is a design smell that could also indicate possible code duplication. Ideally, the common code will be refactored into a common super-class.

Loosely typed containers. We also found several loosely typed containers which store objects which do not share a common type that is more specific than `Object` or `Serializable`.

Lack of user-defined types. Similarly, the code uses `String`s very heavily, instead of declaring domain-specific, user-defined types. In our approach, we treat `String`s as `shared`, do not reason about them, or show them in the OOGs. From a design standpoint, it is better to declare and use a user-defined type, e.g., a type `GeneUniqueId` instead of `String`.

Overall, these design issues point to the lack of a rich type structure and precise, declared types in the code. Not only can the OOG enable us to identify these design smells, it can also guide us toward fixing them. For instance, we can recommend to the maintainers that they create a common supertype for a number of related types that contain `Table` in their name. Also, the maintainer can point to any architecturally relevant object in the `shared` domain, create a custom type for it if needed, and have the architectural extractor update the annotations to move that object to a top-level domain.

6 Related Work

Relation to our previous case study. Abi-Antoun and Aldrich [18] previously conducted a field study to extract the architecture of a 30-KLOC module, called LbGrid. For the P-X study, we developed and used ArchDefault to generate the initial annotations. For LbGrid, we used the previous defaulting tool. At the end of the first phase with LbGrid, we had 4,000 warnings, whereas P-X has fewer than 500 warnings. This demonstrates that ArchDefault was effective in reducing the manual annotation burden. For the P-X system, we incorporated the maintainer's refinements by changing the annotations to control the abstraction by ownership hierarchy. We also used abstraction by types to declutter the OOG. Unlike the LbGrid case study, the P-X maintainer interacted directly with the extraction tool.

Similarities with Reflexion Models. Murphy's Reflexion Models (RM) [7] inspired the map idea. In RM, a developer maps source-level entities to components in a high-level model. For instance, he maps an interface "Foo" to a "manager" component. In our approach, the architectural extractor maps objects to domains instead of individual components in those domains. The architectural analogue of a domain is the notion of architectural tier. As a first approximation only, ArchDefault takes a map which maps types to domains. After using ArchDefault, the architectural maintainer completes the mapping of objects to domains manually, by fixing the annotation warnings. Using RM to build correctly by hand an entire map of objects to higher-level components would be very challenging since the map will have to somehow deal with aliasing, inheritance and polymorphism [10, §6.6.4].

General architectural extraction process. There are several published case studies in architectural extraction, some of which tackled big legacy systems written in procedural languages. Tzerpos and Holt [19] used a "hybrid" process that combines facts extracted from the code and information derived from interviewing developers. These steps include: collecting "back of the envelope" designs from project personnel; extracting raw facts from the source code; collecting naming conventions for files; clustering code artifacts based on naming conventions; creating tentative structural diagrams, and collecting the reactions of the developers to these tentative diagrams; and so on, until they converged to a code architecture. They concluded that there is a reasonably well-defined sequence of steps to go through to extract a code architecture. Indeed, we followed similar steps to extract the runtime architecture, but did not use clustering and relied on the system maintainer to a much smaller extent.

Many architectural extraction studies use various sources of information extrinsic to the code, with no clear exit criteria. It is also fairly common for different architectural extraction teams to produce very different architectures. In Scholia, the annotations have to be consistent with each other and with the code, and are checked using a tool. During the P-X study, we followed a principled approach: we added annotations, typechecked them, extracted OOGs, refined the OOGs on our own, then involved the system maintainer to a very limited extent (Table 4). Our exit criteria are to minimize the number of objects in the top-level domains, and the number of annotation warnings.

Architecture extraction of procedural systems. Gröne et al. [20] manually extracted the architecture of Apache (written in C) and represented the architecture using agents. The architectural extraction involved ad-hoc manual techniques and many people—many students enrolled in a class. The only tool used for the analysis of the source code transformed the C source code into a set of syntax highlighted and hyperlinked HTML file. The study justified not using more advanced tools by saying that "an important amount of information needed for the conceptual architecture is not existent in the code and therefore cannot be extracted by a tool" [20]. This is precisely why we use annotations, to clarify some of the architectural intent in the code, and extract abstractions that match the mental model of the system maintainer.

Extraction of code architecture of object-oriented systems. Several researchers extracted architectural views of large object-oriented systems, e.g., Jigsaw system (300 classes), but focused on the code architecture [21, 22]. Medvidovic and Jacobak [21] point out that tools are often unable to discover a relationship that is implemented indirectly, e.g., by using instances of container classes such as `Vector`, `Map`, `List`, to store objects of some other application class. User-defined container classes complicate matters further. On the other hand, Scholia readily handles those container classes. Indeed, we spent a lot of effort adding generic types first, then ownership domain annotations to the containers. In particular, when adding annotations to a container, one has to specify separately the annotation on the container object itself and the annotation on the contained elements.

Dynamic extraction of runtime architecture. Many have preferred dynamic analyses (or mixed with static analysis) to extract the as-built runtime architecture [4, 5]. Our approach is entirely static. We did not run the P-X system. Doing so will have significantly complicated the process for the architectural extractors, since it will have required setting up a database server, a web server, configuration files, etc.

Evaluation of extracted architectures. Evaluating the quality of an extracted architecture is subject to debate, with no generally accepted evaluation criteria. More generally, this appears to be a common issue in the empirical evaluation of reverse engineering tools. Tonella et al. [23] state: "the same piece of information recovered from the code may be immensely useful or completely unusable depending on the end user who is performing the current software engineering task and depending on the amount of knowledge the user already has about the system".

One approach to measure the "goodness" of an extracted architecture is to compute various structural metrics. Indeed, clustering methods use this approach to evaluate the quality of the result. For example, a clustering is "good" if the clusters are reasonably sized and exhibit low coupling and high cohesion [24]. In our case, we measured percentages on the annotations we added, and ensured that we were not treating an excessive proportion of the objects as `shared` (Table 5). Another measurable success criteria was to minimize the number of objects in the top-level domains (Table 3).

Scalability of approach. The scale of the system we analyzed may pale in comparison to previous case studies that analyzed the code architecture of large systems. Indeed, the tools that analyze the runtime architecture are much less mature than those that analyze the code architecture. Moreover, SCHOLIA is a design-intent-based technique that requires specifying the design intent using annotations. This makes using SCHOLIA to analyze millions of lines of code prohibitively costly, without automated annotation inference. For comparison, the closest prior work that used annotations to extract object models, by Lam and Rinard [25], was evaluated on one 1700-line system. This paper confirmed the empirical estimate of effort required for adding annotations at the rate of 1-hour/KLOC, which is not unreasonable for studying some of the core sub-systems of business-critical applications.

7 Conclusion

In this paper, we presented the results of a case study in extracting the runtime architecture of an actively maintained object-oriented system. We confirmed that the system maintainers understood a global, hierarchical object graph of the entire system, and that it was possible to refine the OOG incrementally to convey their design intent.

There were a few cases where the as-built OOG clashed with the maintainer's expectations. Addressing these clashes would likely require refactoring or re-engineering the code.

Acknowledgements

We thank Professor Sorin Draghici for granting us permission to study the Pathway Express code base and publish details of the case study.

References

[1] P. Tonella and A. Potrich, *Reverse Engineering of Object Oriented Code*. Springer-Verlag, 2004.

[2] S. Ducasse and D. Pollet, "Software Architecture Reconstruction: A Process-Oriented Taxonomy," *TSE*, vol. 35, no. 4, 2009.

[3] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architectures: View and Beyond*. Addison-Wesley, 2003.

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

[5] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, "Discovering Architectures from Running Systems," *TSE*, vol. 32, no. 7, 2006.

[6] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey, "Structural Redocumentation: a Case Study," *IEEE Software*, vol. 12, no. 1, 1995.

[7] G. Murphy, D. Notkin, and K. J. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation," *TSE*, vol. 27, no. 4, 2001.

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

[9] J. Aldrich and C. Chambers, "Ownership Domains: Separating Aliasing Policy from Mechanism," in *ECOOP*, 2004.

[10] M. Abi-Antoun, "Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure," Ph.D. dissertation, Carnegie Mellon University, 2009, Technical Report CMU-ISR-09-119.

[11] "Intelligent Systems and Bioinformatics Laboratory (ISBL)," http://vortex.cs.wayne.edu/projects.htm/, 2003.

[12] "Eclipse Metrics Plugin," http://metrics.sourceforge.net/, 2010.

[13] A. Kieżun, M. D. Ernst, F. Tip, and R. M. Fuhrer, "Refactoring for Parameterizing Java Classes," in *ICSE*, 2007.

[14] Z. Hailat and M. Abi-Antoun, "ArchDefault: a Tool for Propagating Default Ownership Domain Annotations," www.cs.wayne.edu/~mabianto/px/, Tech. Rep., 2011.

[15] M. Abi-Antoun, T. Selitsky, and T. LaToza, "Developer Refinement of Runtime Architectural Structure," in *Workshop on SHAring and Reusing architectural Knowledge (SHARK)*, 2010.

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

[17] W. Dietl, S. Drossopoulou, and P. Müller, "Generic Universe Types," in *ECOOP*, 2007.

[18] M. Abi-Antoun and J. Aldrich, "A Field Study in Static Extraction of Runtime Architectures," in *PASTE*, 2008.

[19] V. Tzerpos and R. C. Holt, "A Hybrid Process for Recovering Software Architecture," in *CASCON*, 1996.

[20] B. Gröne, A. Knöpfel, and R. Kugel, "Architecture Recovery of Apache 1.3 – a Case Study," in *International Conference on Software Engineering Research and Practice*, 2002.

[21] N. Medvidovic and V. Jakobac, "Using Software Evolution to Focus Architectural Recovery," *ASE*, vol. 13, no. 2, 2006.

[22] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit, "Extraction of Component-Based Architecture from Object-Oriented Systems," in *WICSA*, 2008.

[23] P. Tonella, M. Torchiano, B. Du Bois, and T. Systä, "Empirical Studies in Reverse Engineering: State of the Art and Future Trends," *Empirical Software Engineering*, vol. 12, no. 5, 2007.

[24] T. Systä, P. Yu, and H. Müller, "Analyzing Java software by combining metrics and program visualization," in *CSMR*, 2000.

[25] P. Lam and M. Rinard, "A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information," in *ECOOP*, 2003.