Ownership Domains in the Real World

Marwan Abi-Antoun
School of Computer Science
Carnegie Mellon University
marwan.abi-antoun@cs.cmu.edu

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
jonathan.aldrich@cs.cmu.edu

Abstract

The Ownership Domains type system has had publicly available tool support for a few years. However, the previous implementation used non-backwards compatible language extensions to Java and ran on a research infrastructure, which made it difficult to conduct substantial case studies on interesting systems.

We first present a re-implementation of ownership domains using Java 1.5 annotations and the Eclipse infrastructure. We then use the improved tool to annotate two real 15,000-line Java programs while using refactoring tool support, generics and external libraries.

Ownership domains, as most other ownership type systems, provide useful encapsulation properties. We illustrate using actual examples from the subject systems how ownership domains also express and enforce design intent regarding object encapsulation and communication and help identify tight coupling. Finally, we mention some expressiveness gaps that we encountered.

1. Introduction

Researchers have proposed many ownership type systems, e.g., [15, 10, 3, 17, 41], but have not reported significant experience with most of them on real code. Only a few systems, notably Ownership Domains [6, 3], Universes [17] and Generic Ownership [41], have released tool support [46, 20, 40], and even fewer systems have been evaluated in substantial case studies [6, 25, 2, 38].

The previous implementation of ownership domains [3] used non-backwards compatible extensions of Java [46]. As a result, none of the rich tool support for Java programs was available to programs with ownership domain annotations[1].

In a previous case study [2], we identified that adding ownership domain annotations to existing code often highlights refactoring opportunities. For instance, a lengthy domain parameter list is often an indication of tightly coupled code that could benefit from refactoring — such as extracting an interface and programming to that interface. It is unrealistic to assume that it is possible to refactor all such code prior to annotating it. In our experience, having access to refactoring tool support during the annotation process was invaluable. Using language extensions also makes it harder to partially and incrementally annotate existing code and thus conduct case studies on interesting systems. Finally, the previous tool used a modified research infrastructure [8] that is no longer actively maintained and does not support Java generics — as of this writing.

To address these adoptability challenges, we re-implemented the Ownership Domains type system using the annotation facility in Java 1.5 [27], so that Java programs with ownership annotations remain legal Java 1.5 programs. We also implemented the tool as a plugin to the Eclipse open source development environment that has become popular with researchers and practitioners [24, 37].

[1] The Universes tools built on the Java Modeling Language (JML) infrastructure support both language extensions and stylized comments [20].

We believe this improved tool support promotes the adoptability of the ownership domains technique by Java developers as follows. First, all the Eclipse tool support such as syntax highlighting, refactoring, etc., remains available to annotated programs. Second, using annotations makes it easier to support in a non-breaking way additional annotations such as external uniqueness [14] or `readonly` [17]. Third, using annotations gives the ability to incrementally and partially specify annotations on large code bases. Fourth, using annotations will make it possible to study the evolution of programs with ownership annotations, an area that has not received much attention — since no one will maintain a program with limited tool support. Finally, annotating existing code is difficult and time-consuming and tools are being developed to add annotations semi-automatically [6, 16]. One of the benefits of using annotations over language extensions is that an inference algorithm cannot break an existing program by inserting potentially incorrect annotations.

We made the following design choices for the annotation system. First, we worked within the limits of Java 1.5 annotations [27], even though annotations may be more verbose than an elegantly designed language. Moreover, Java 1.5 annotations impose several restrictions, e.g., no annotations on generic type arguments. Other researchers have tried to eliminate some of these restrictions by proposing revisions of the language [19], but until such proposals are officially adopted, their prototype implementations are not Eclipse compatible, an important factor for adoptability. Second, to work around the Java 1.5 limitation of allowing annotations only on declarations, we consistently declare additional temporary variables and add annotations to them. This has worked well for new expressions, cast expressions (both implicit and explicit) and arguments for method and constructors. Third, checking ownership domain annotations only generates informational messages, i.e., no errors or warnings, and does not stop a developer from running the program. Fourth, we hard-code a minimal number of implicit defaults and provide a separate tool to supply explicit reasonable defaults to reduce the annotation burden. In the future, this tool can be replaced with a smarter annotation inference tool. Finally, the annotations are non-executable and do not affect the program's behavior [2]; unlike the earlier implementation, the current system does not include runtime checks. As a result, the annotation-based system is unsound at casts — but could be made sound using bytecode rewriting to add necessary dynamic checks.

The rest of the paper is organized as follows: we review ownership domains in Section 2, describe the annotation language in Section 3 and the salient tool features in Section 4. We discuss two case studies in Section 5 and show how ownership domains express and enforce design intent related to object communication and encapsulation. We discuss some expressiveness gaps that we encountered in Section 6 and conclude with related work in Section 7.

[2] Annotations may increase the memory footprint and slow down class loading as a result, but no empirical data has been reported to date.

2. Review of Ownership Domains

Ownership domains are conceptual groups of objects with explicit domain names and explicit policies that govern references between them. Each object belongs to a single ownership domain, and a top-level domain is assumed.

Public and Private Domains. Each object can declare one or more *public* or *private* domains to hold its internal objects, thus supporting hierarchy. A public domain is accessed using a syntax similar to field access. Domain declarations are added to a class, but for each instance of that class, fresh instances of these domains are created for that object, i.e., `obj1.DomainA` and `obj2.DomainA` are distinct if `obj1` and `obj2` are instances of the same class `T` and do not alias each other.

Explicit Domain Links. Each object can declare a policy describing the permitted aliasing among objects in its public domains, and between its private domains and public domains. Ownership domains support two kinds of policy specifications: a) a link from one domain to another allows objects in the first domain to access objects in the second domain; and b) permission to access an object implies permission to access its public domains. In addition to explicit domain links, the following implicit policy specifications are included: a) an object has permission to access other objects in the same domain; and b) an object has permission to access objects in any domain that it declares. Any reference not explicitly permitted by these rules is prohibited, and link permissions are not transitive.

Ownership Domain Parameters. Two objects can access objects in the same domain, as long as implicit or explicit permissions allow that access, by declaring a formal domain parameter on one object, and binding that formal domain parameter to another domain. Method domain parameters are also supported and are often needed for static methods.

Alias Types. In addition, the following special annotations are defined for increased expressiveness [3]:

- `unique`: indicates an object to which there is only one reference such as a newly created object. Unique objects can be passed linearly from one object to another, by destroying the old reference to the object when the new reference is created;
- `lent`: one ownership domain can temporarily lend an object to another ownership domain, with the constraint that the second ownership domain will not create any persistent references to that object: e.g., a method formal parameter is often annotated with `lent` to indicate that it is a temporary alias;
- `shared`: indicates that an object may be aliased globally. `shared` references may not alias non-`shared` references.

Unlike *owner-as-dominator* type systems [15], public domains in the ownership domains type system can express constructs such as iterators [3] (See Figure 1) or an instance of the Composite design pattern [22, p. 163] that does not encapsulate its subcomponents and gives clients the ability to add components to any composite of the hierarchy and not only to the root composite [30]. Developers can still express owner-as-dominator in ownership domains by: a) never declaring a public domain; and b) never linking a domain parameter to an internal domain [3].

3. Annotation Design

In this section, we describe the concrete annotation syntax. For maximum flexibility and to work around some of the limitations of Java 1.5 annotations, all annotation values are strings. Annotations that are plural take values that are arrays of strings.

The annotations are illustrated using snippets from a canonical `Sequence` abstract data type, a common benchmark for ownership type systems. Within the `Sequence`, the `iters` ownership domain is used to hold `Iterator` objects that clients use to traverse the `Sequence`, and the default *private* `owned` ownership domain is used to hold the `Cons` cells in the linked list that is used to represent the `Sequence`. The full example is shown in Figure 1.

@Domains: declare public or private domains on a type.

- **Format**: $identifier$
- **Applies to**: type (class or interface).
- **Examples**: the following declares a private `owned` domain (`owned` is private by naming convention), and a public domain `iters` to store the `Iterator` objects of the `Sequence`.

```
@Domains({"owned","iters"})
class Sequence<T> {
...
}
```

@DomainParams: declare ordered domain parameters on a type or method domain parameters on a method.

- **Format**: $identifier$
- **Applies to**: type or method.
- **Examples**: `Sequence` declares a domain parameter `Towner` to hold its elements.

```
@DomainParams({"Towner"})
class Sequence<T> {
...
}
```

@DomainInherits: pass parameters to superclass or implemented interfaces.

- **Format**: $typename < parameter, \ldots >$
- **Applies to**: type (class or interface).
- **Examples**: the `Iterator` interface is also parameterized by the `Towner` domain parameter. Class `SeqIterator` inherits domain parameter `Towner` from interface `Iterator`, and adds the `list` parameter to access the `Cons` cells.

```
@DomainParams({"list", "Towner"})
@DomainInherits({"Iterator <Towner>"})
class SeqIterator<T> implements Iterator<T> {
...
}
```

@DomainLinks: declare domain links.

- **Format**: $fromDomainId$ -> $toDomainId$
- **Applies to**: type (class or interface).
- **Examples**: the `Sequence` gives `Iterator` objects in the `iters` domain permission to access objects in the `owned` domain, including the `Cons` cells.

```
@DomainLinks({..., "iters -> owned", ...})
class Sequence<T> {
...
}
```

@DomainAssumes: declare domain link assumptions.

- **Format**: $fromDomainId$ -> $toDomainIds$
- **Applies to**: type (class or interface).
- **Examples**: the `Sequence` assumes that the `owner` of the `Sequence` has access to the `Towner` domain containing the sequence elements.

```
@DomainAssumes("owner -> Towner") /* default */
class Sequence<T> {
...
}
```

@Domain: declare the domain, actual parameters and actual array parameters.

- **Format**: `annotation<domParams,...>[arrayParams,...]`
  - `annotation`: indicate a domain name (e.g., `owned`), one of the special alias types (e.g., `unique`), or a public domain of an object using a field access syntax (e.g., `seq.iters`);
  - `<domParams,...>`: specify actual domain parameters by order of formal domain parameters, at object creation and access sites;

- `[arrayParams,...]`: in ownership domains, arrays have two ownership modifiers, one for the array object itself and one for the objects stored in the array. For variables of array type, this argument specifies the actual array parameters by order of array dimension (for multi-dimensional arrays).
- **Applies to**: local variable declaration, field declaration, method formal parameter and method return value.
- **Examples**: the following declares a `unique Iterator` object and binds the `list` domain parameter on `SeqIterator` to `owned` domain on `Sequence`, and the `Towner` domain parameter on `SeqIterator` to the parameter by the same name on `Sequence`.

```
@Domain("unique<owned,Towner>")
SeqIterator<T> it = new SeqIterator<T>(head);
```

- **Examples**: a `lent` array of `shared` `Strings`:

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

@DomainReceiver: declare the domain of the receiver of a constructor or a method.
- **Format**: *identifier*
- **Applies to**: constructor or method.
- **Examples**:

```
@DomainReceiver("state")
void run() { ... }
```

4. Tool Design and Implementation

Ownership domain annotations are typechecked using two visitors on the Eclipse Abstract Syntax Tree (AST).

4.1 Ownership Domains Typechecking

A first-pass visitor performs the following:
- **Identify Problematic Patterns:** these will need to be replaced with equivalent constructs, e.g., by declaring a local variable and adding the appropriate annotations to it;[3]
- **Read Annotations from AST:** the Java 1.5 annotations added to a program are part of the AST. The visitor locates the annotations nodes in the AST and parses their contents using a JavaCC [26] parser. The visitor also locates special block comments on method invocation expressions as described later. In addition, the visitor infers default annotations for some AST nodes that cannot be annotated, e.g., it implicitly defaults the `NullLiteral` AST node to `unique`. The visitor maps each AST node to an annotation structure in preparation for the second pass visitor which will typecheck the annotations;
- **Propagate Local Annotations:** the visitor propagates the explicit annotations from the AST nodes (for types, variables, and methods) to all the expression nodes in the AST, including translating formals to actuals.

A second-pass visitor checks the annotations on each expression based on the static semantics of Ownership Domains. Checking the assignment rule requires a value flow analysis. A Live Variables Analysis (LVA) from a lightweight data flow analysis framework [5] — that also uses the Eclipse AST, is invoked intra-procedurally at each method boundary using a separate visitor. The LVA analysis verifies that a `unique` pointer only has one non-`lent` read.

4.2 Additional Features

The tool offers the following additional features:

[3] Using the Eclipse built-in refactoring (“Extract Local Variable”), this operation can be performed with very little effort.

```
@Domains({"owned","iters"})
@DomainParams({"Towner"})
@DomainAssumes("owner -> Towner")
@DomainLinks({"owned->Towner", "iters->Towner",
              "iters->owned"})
class Sequence<T> {
  @Domain("owned<Towner>") Cons<T> head;
  void add(@Domain("Towner")T o) {
    @Domain("owned<Towner>")
    Cons<T> cons = new Cons<T>(o,head);
    head = cons;
  }
  @Domain("iters<Towner>") Iterator<T> getIter() {
    @Domain("iters<owned, Towner>")
    SeqIterator<T> it = new SeqIterator<T>(head);
    return it;
  }
}

@DomainParams({"Towner"})
@DomainAssumes("owner -> Towner")
class Cons<T> {
 @Domain("Towner") T obj;
 @Domain("owner<Towner>")Cons<T>  next;

 Cons(@Domain("Towner")T obj,
      @Domain("owner<Towner>")Cons<T> next) {
   this.obj=obj;
   this.next=next;
 }
}

@DomainParams({"Towner"})
interface Iterator<T> {
  @Domain("Towner")T next();
  boolean hasNext();
}

@DomainParams({"list", "Towner"})
@DomainAssumes({"list -> Towner"})
@DomainInherits({"Iterator<Towner>"})
class SeqIterator<T> implements Iterator<T> {
  @Domain("list<Towner>")Cons<T> current;
  ...
  SeqIterator(@Domain("list<Towner>")Cons<T> head) {
    current = head;
  }
  public @Domain("Towner") T next() {
    @Domain("Towner")T obj2 = current.obj;
     current = current.next;
     return obj2;
   }
}

@Domains({"owned","state"})
class SequenceClient {
  final @Domain("owned<state>")
        Sequence<Integer> seq = new Sequence<Integer >();

  void run() {
    @Domain("state")Integer int5 = new Integer(5);
    seq.add(int5);
    @Domain("seq.iters<state>")
    Iterator<Integer> it = this.seq.getIter();
    while (it.hasNext()) {
      @Domain("state")Integer cur = it.next();
      ...
    }
  }
  ...
}
```

Figure 1. A `Sequence` Abstract Data Type with ownership domain annotations.

```
@DomainParams({"state"})
class Student {
...
}
@DomainParams({"state"})
class Data ... {
  final @Domain("state<state<state>>")
  Sequence<Student> vStudent;

  @Domain("state<state>")Student
  getStudentRecord(@Domain("shared")String sSID) {
    @Domain("vStudent.iters<state<state>>")
    Iterator<Student> i = vStudent.getIter();
    while (i.hasNext()) {
      @Domain("state<state>")
      Student objStudent = i.next();
      ...
    }
    ...
  }
}
```

Figure 2. Adding annotations to generic code.

External Libraries. There are two approaches to support adding annotations to the standard Java libraries and other third-party libraries, one that involves annotating the library and pointing the tool to the annotated library and one that involves placing the annotations in external files. The earlier tool used the former approach [46], but we adopted the latter approach this time since it does not require changing library or third-party code — which may not be available and when it is, tends to evolve separately. Other annotation based systems adopted the same strategy [42]. The tool supports associating ownership domain annotations with any Java bytecode `.class` file using an external XML file, following the same annotation constructs described in Section 3.

Generics. Our annotation system currently treat generic types as orthogonal to ownership domain parameters, so generic type parameters and arguments are added separately from ownership domain annotations — except that nested actual domains may need to be provided where applicable. Proponents of Generic Ownership [41] argue that this leads to awkward syntax, which may be true. However, in our case studies annotating two 15,000-line Java programs including using generic types, we did not observe this to be a serious problem. Figure 2 illustrates the interaction between generics and ownership domains: the `Student` class is parameterized by the `state` domain parameter. The `Data` class maintains a `Sequence` of `Student` objects and is also parameterized by `state`.

Method Domain Parameters. Java 1.5 annotations cannot be added at method invocation expressions. So we used block comments to specify the actual domains for a parameterized method (See Figure 3 for an example). Unfortunately, even proposals to improve the Java 1.5 annotation facilities [19] do not yet address adding annotations to such expressions.

Defaulting Tool. To reduce the annotation burden, we implemented a separate tool to add default annotations such as marking private fields as `owned`, method parameters as `lent`, and `Strings` as `shared`. However, an annotation added by the defaulting tool (e.g., `owned`) may need to be modified manually to supply actual domains for domain parameters (e.g., `owned<owned>`).

Annotation 'owner'. We also added the special `owner` annotation, similar to `peer` in Universes [17]. Using `owner` can often eliminate a domain parameter: e.g., in Figure 1, `Cons.owner` is `Sequence`, `SeqIterator.owner` is `Sequence`.

4.3 Tool Limitations and Future Work

Java 1.5 annotations suffer from the following limitations: (1) A declaration cannot have multiple annotations of the same annotation type; (2) Annotation types cannot have members of the their

```
class Sequence<T> {
...
  @DomainParams("TTowner") /* Method domain parameter */
  @Domain("shared") /* Domain for return value */
  static <TT> String
    toString(@Domain("lent<TTowner>")Sequence<TT> seq) {
    ...
  }
  void dump() {
    @Domain("owned<shared>")
    Sequence<String> seq = ...;

    @Domain("shared")
    /* Provide <actuals...> using block comment */
    String str = Sequence.toString/*<state>*/(seq);
  }
}
```

Figure 3. Declaring and binding method domain parameters.

```
while (objCourseFile.ready()) {
  this.vCourse.add(new Course(courseFile.readLine()));
}
/*  ABOVE MUST BE REWRITTEN AS  ....  */
while (objCourseFile.ready()) {
 @Domain("shared")String line = courseFile.readLine();
 @Domain("state<state>")Course crs = new Course(line);
 this.vCourse.add(crs);
}
```

Figure 4. Re-writing a new expression using local variables.

own type; (3) It is only legal to use single-member annotations for annotation types with multiple members, as long as one member is named `value`, and all other members have default values. Otherwise, the more verbose syntax is required, e.g., `@Name(first = "Joe", last = "Hacker")`; (4) Annotation types cannot extend any entity (class, interface or annotation); and (5) Annotations are allowed on type, field, variable and method declarations and not allowed on type parameters or method invocations.

The fist restriction prevented us from using the `@Domain` annotation to specify both the annotation on the receiver and on the return type of a method. The second restriction prevented us from having shorthand constant annotations for the special alias types, e.g., `@owned` instead of `@Domain("owned")`: such constants cannot be used inside other annotations as in `@Domain(annotation = @owned, parameters = {@owned})`.

To avoid having multiple ways of indicating the same meaning, we use strings for all the annotations and require annotations of the form `@Domain("owned<owned>")`. Although developers may be more likely to introduce spelling mistakes in string annotations, the typechecker will catch these problems early enough. The third restriction, i.e., the lack of positional arguments, required the use of the verbose syntax `@Domains(publicDomains = {"d1", "d2"}, privateDomains = {"pda", "pdb"})`.

The final restriction and the current lack of annotation inference require converting some expressions to more verbose constructs by declaring local variables and annotating them. The most common such expressions were new expressions (See Figure 4) and cast expressions (See Figure 5).

We plan to address some of the following limitations:

- **Infer method domains:** just as actual type arguments do not have to be passed to a generic method in Java, it may be possible to infer, in most cases, the actuals for method domain parameters based on the types of the actual arguments;
- **Allow suppressing messages:** since reflective code cannot be annotated successfully using ownership domains [6];

```
ArrayList vCourse = student.getRegisteredCourses();
for (int i=0; i<vCourse.size(); i++) {
 if (((Course) vCourse.get(i)).conflicts(course)) {
    ...
 }
}
/* ABOVE MUST BE REWRITTEN AS ... */
@Domain("lent<state>")
ArrayList vCourse = student.getRegisteredCourses();
for (int i=0; i<vCourse.size(); i++) {
  @Domain("lent<state>")
  Course crs = (Course) vCourse.get(i);
  if (crs.conflicts(course)) {
     ...
  }
}
```

Figure 5. Re-writing a cast expression using local variables.

- **Display annotations more elegantly:** an Eclipse plug-in by Eisenberg and Kiczales [18] can beautify Java 1.5 annotations for interactive editing while the analysis uses the same AST.

5. Ownership Domains Case Studies

The annotation-based system is mostly complete — the domain link checks are still being implemented as of this writing. We used the tools to add and check ownership domain annotations on two real Java programs with around 15,000 lines of code each.

JHotDraw. The subject system for the first case study is JHotDraw [23]. Version 5.3 has around 200 classes and 15,000 lines of Java. JHotDraw is rich with design patterns [22], uses both composition and inheritance heavily and has evolved through several versions. We first used the defaulting tool then manually modified the annotations as needed. Adding annotations was iterative. For instance, over several iterations, we made more use of the `owned` annotation. JHotDraw was annotated without making any structural refactoring such as extracting interfaces, etc. Some code changes were needed however to use our annotation system, e.g., extract a local variable from a new expression and add an annotation on the local variable, convert an anonymous class to a nested class to add domain parameters to it, etc. JHotDraw Version 5.3 did not use generic types, so we used Eclipse refactorings [21] to infer generic types of containers.

HillClimber. By many accounts, JHotDraw is considered the brainchild of experts in object-oriented design and programming. In comparison, the subject system for this case study, HillClimber, is another 15,000 line application that was mainly developed and maintained by undergraduates [2]. In previous work, we re-engineered the original Java program to an ArchJava [4] implementation with ownership domain annotations, but using language extensions instead of Java 1.5 annotations [2]. The re-engineering case study also produced a version that refactored the original code by making most class fields as `private` [2]. For this case study, we started from the refactored Java code and added ownership domain annotations to it.

Unlike JHotDraw, adding annotations to HillClimber involved refactoring to decouple the code as discussed below. We also refactored the code to use generics, mostly automatically using Eclipse. However, Eclipse cannot infer the generic type of a variable of type `Vector` storing arrays of `Node` objects. Such code was manually refactored to use `Vector<Vector<Node>>`.

Compared to the earlier case study with language extensions [2], the annotation-based system allowed using Eclipse refactoring tools to extract interfaces and infer generic types while adding the ownership domain annotations. Comparing the number of hours would not be meaningful since the annotation-based system was still under development while the case study was under way, and that would not account for the learning effect of annotating the same program twice. Anecdotally, we were more productive with the annotation-based system than with the earlier tool using language extensions. The overall process changed around 40% of the lines of code in HillClimber. The 40% code changes included boilerplate `imports` to use our Java 1.5 annotations, and code changes to support adding annotations to some expressions. To more accurately gauge the manual annotation overhead, an AST-visitor was used to count the instances where the current annotation is the same as the one generated by the defaulting tool: over 40% of the annotations were exactly the same as the default ones for HillClimber; that number was around 30% for JHotDraw. There are 60 type errors remaining in JHotDraw and 42 errors remaining in HillClimber.

In this following discussion, we illustrate using actual examples from JHotDraw and HillClimber, how ownership domains can express and enforce design intent related to object encapsulation and communication, using code snippets from the subject systems. The code was slightly edited for presentation by removing trivial modifiers. Some code is highlighted using underlining.

5.1 Ownership domains enforce instance encapsulation

All ownership type systems can express and enforce instance encapsulation which is stronger than the module visibility mechanism of making a field `private`. In ownership domains, placing a field in the private `owned` domain means that the object can be reached only by going through its owner; as a result, no aliases to that object can leak to the outside. Consider `CompositeFigure` in JHotDraw:

```
@Domains({"owned"})
@DomainParams({"M"})...
abstract class CompositeFigure ... {
 // The figures that comprise this figure
 @Domain("owned<M<M>>")Vector<Figure> fFigures;

 /**
  * Adds a vector of figures.
  */
 void addAll(@Domain("M<M<M>>")Vector<Figure> figs) {
  // Cannot assign object in "M<M>" to "owned<M>"
  // this.fFigures = figs;

  // This is correct however
  fFigures.addAll(figs);
 }
...
}
```

Annotating field `fFigures` with `owned` encapsulates the list of composite `Figures` (`fFigures`) to prevent objects that only have access to the composite object from modifying the list directly. If a developer tries to subvert the language visibility mechanisms by exposing a `private` or `protected` field using a `public` accessor method, the ownership domains type system prohibits a `public` method from having an `owned` parameter or return value. Letting Eclipse generate a setter for the `fFigures` field produces the following code — without annotations:

```
 void setFFigures(Vector<Figure> figs) {
   this.fFigures = figs;
 }
```

Upon adding the annotations, a developer can realize that the setter is overwriting the existing field since the parameter `figs` cannot be marked as `owned` and any other annotation would fail the assignment check when overwriting the `fFigures` field.

When manually adding annotations, it is possible to miss many opportunities for making objects `owned`. Indeed, we initially annotated `fFigures` with the domain parameter `M` instead of `owned`. In some cases, objects should be `owned` but are not, and making them `owned` may require code changes, e.g., to return a copy of an object instead of an alias to a private field.

Visualizing the annotations encouraged us to make more use of the `owned` annotation since `owned` avoids cluttering the top-level domains [1]. Perhaps better tool support can prompt a developer to encapsulate a field that could be annotated with `owned` but is not, e.g., a lightweight compile-time ownership inference algorithm [33] could suggest possible Eclipse "quickfixes".

5.2 Ownership domains specify architectural tiers

A tiered architecture is often used to organize an application into a User Interface tier, a Business Logic tier, and a Data tier. Ownership domains can express and enforce such a tiered runtime architecture by representing a tier as an ownership domain [3], and a permission between tiers as a domain link to allow objects in the User Interface tier to refer to objects in the Business Logic tier but not vice versa. Such an architectural structure and constraints cannot be easily expressed in plain Java code.

We organized the core JHotDraw types in Figure 6 according to the Model-View-Controller design pattern as follows:

- `Model:` consists of `Drawing`, `Figure`, `Connector`, etc. A `Drawing` is composed of `Figures` which know their containing `Drawing`. A `Figure` has a list of `Handles` to allow user interactions. A `Drawing` also extends `FigureChangeListener` (not shown) to listen to changes to its `Figures`.
- `View:` consists of `DrawingEditor`, `DrawingView` and associated types. `DrawingView` extends `DrawingChangeListener` (not shown) to listen to changes to `Drawing` objects.
- `Controller:` includes interfaces such as `Handle`, `Tool` and `Command`. A `Tool` is used by a `DrawingView` to manipulate a `Drawing`. A `Command` encapsulates an action to be executed — a simple instance of the Command design pattern [22, p. 233] without undo support.

Once we defined the three top-level ownership domains, `Model`, `View` and `Controller`, we passed the corresponding domain parameters `M`, `V` and `C` to various types as discussed below. A visualization of the JHotDraw execution structure based on these ownership domain annotations is available [1].

In HillClimber, the application *window* uses a *canvas* to display *nodes* and *edges* of a *graph* in order to demonstrate algorithms for constraint satisfaction problems provided by the *engine*. So we organized the HillClimber types in Figure 12 as follows. The `data` ownership domain stores the graph objects (instances of `Graph`, `Node`, etc., and those of their subclasses, `HillGraph`, `HillNode`, etc.). The `ui` domain holds user interface objects. The `logic` domain holds instances of `HillEngine`, `Search` (and subclasses thereof) objects, and associated objects. A visualization of the HillClimber execution structure based on these ownership domain annotations is available [1].

5.3 Ownership domains expose implicit communication

Design patterns — such as Observer [22, p. 293], used to decouple object-oriented code also tend to make the communication between objects implicit. Adding ownership domain annotations helps make that communication more explicit.

We initially wanted to parameterize `Drawing` (See Figure 7) with only the `M` domain parameter, but `DrawingChangeListener` is implemented by `DrawingView`. So `DrawingChangeListener` needed to be annotated with the `V` domain parameter corresponding to the `View`. By making implicit communication explicit, annotations seem to prematurely constrain `DrawingChangeListener` objects to be in the `View` domain. Since `Drawing` was a core interface referenced by other interfaces in the core `framework` package, this led to passing all three domain parameters to many additional interfaces and classes.

It is true that if `Drawing` had to be parameterized by domain parameter `V` for some other reason, the implicit communication in

```
/**
 * Drawing is a container for Figures. Drawing sends
 * out DrawingChanged events to DrawingChangeListeners
 * whenever a part of its area was invalidated.
 * The Observer pattern is used to decouple the Drawing
 * from its views and to enable multiple views.
 */
@DomainParams({"M", "V"})
@DomainInherits({"FigureChangeListener<M>",...})
interface Drawing extends FigureChangeListener... {
  // Adds a listener for this drawing.
  void addDrawingChangeListener(
     @Domain("V<M,V>")DrawingChangeListener l);

  // Adds a figure and sets its container
  // to refer to this drawing.
  @Domain("M<M>")
  Figure add(@Domain("M<M>") Figure figure);
...
}
```

Figure 7. Adding annotations to `Drawing`.

```
@DomainParams({"M","V","C"})
interface Handle {
 void invokeStart(@Domain("V<M,V,C>")DrawingView v);
 ...
 @Domain("M<M,V,C>")Undoable getUndoActivity();
}
```

Figure 8. `Handle` with `M`, `V` and `C` domain parameters.

the observer would not have been discovered this way. Ownership domain annotations help make implicit communication explicit when a reference requires permission to access a new part of the program for the first time.

In HillClimber, adding ownership domain annotations exposed covert object communication through base classes from two parallel inheritance hierarchies. During an early iteration, we parameterized the base class `GraphCanvas` by the `ui` and `data` domain parameters. We then realized that `Graph`, the base class for `HillGraph`, required the `ui` domain parameter (See Figure 12). Class `Graph` only needed the `ui` domain parameter to properly annotate a `GraphCanvas` field reference that we did not expect. This in turn revealed that `HillGraph` and `HillCanvas` were communicating through their base classes `Graph` and `GraphCanvas`. In the end, the reference to `GraphCanvas` was moved from `Graph` to `HillGraph` and generalized as an `IHillCanvas` reference by extracting an interface `IHillGraph` from `HillGraph`.

5.4 Ownership domains expose tight coupling

Let us temporarily ignore the earlier limitation with adding annotations to the listeners and assume that `Drawing` could be parameterized by only the `M` domain parameter. Let us consider whether it would be possible to parameterize interface `Handle` (See Figure 8) with domains `M` and `C`. A `Handle` would be in the `C` domain and would access objects in that domain and in `M` domain, i.e., it should not access objects in the `V` domain parameter. Note that even if the explicit parameter `C` was not provided, that domain would still be accessible to `Handle` using the `owner` annotation.

A comment in the code indicated that Version 4.1 deprecated the original `invokeStart` method which took a `Drawing` object as one of its parameters, in favor of an `invokeStart` method that takes instead a formal parameter `DrawingView` parameterized by `M`,`V`, and `C`. This required passing to `Handle` the additional domain parameter `V`. Since `Handle` is a core interface referenced by other interfaces in the core `framework` package, this also led to passing all three domain parameters to many additional types.

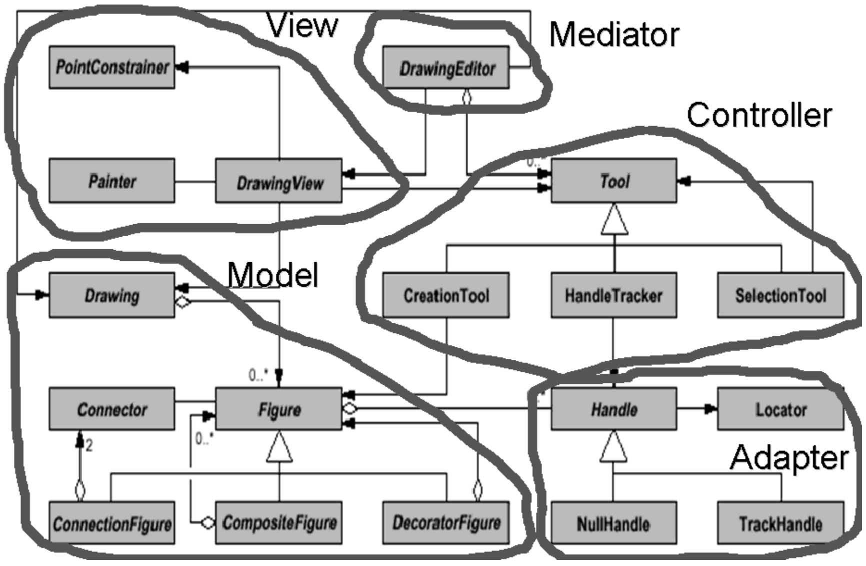

Figure 6. Simplified class diagram for JHotDraw (Adapted from manual class diagram by Riehle [43, 12]).

```
@DomainParams({"M","C"})
interface Handle {
 @DomainParams({"V"})
 void invokeStart(@Domain("V<M,V,C>")DrawingView v);
 ...
 @Domain("M<M>")Undoable getUndoActivity();
}
```

Figure 9. `Handle` with only `M` and `C` domain parameters.

```
@DomainParams({"M","C"})
@DomainInherits({"Handle<M,C>"})
abstract class AbstractHandle implements Handle {
 // Will not typecheck since 'V' unbound
 @Domain("V<M,V,C>")DrawingView view;
 ...
 @DomainParams({"V"})
 void invokeStart(@Domain("V<M,V,C>")DrawingView v) {
   // Cannot store argument in field 'this.view'
 }
}
```

Figure 10. Method domain parameters can enforce lifetime.

5.5 Ownership domains expose and enforce object lifetime

Let us assume in this section that the refactoring which introduced the tighter coupling was never performed, i.e., `Handle` still needed a `Drawing` instead of a `DrawingView`. Undo support was added to JHotDraw for the first time in Version 5.3. In particular, `Handle` now had a reference to `Undoable` — which in turn required domain parameters `M`,`V` and `C` because `Undoable`'s `getDrawingView()` method returned a `DrawingView`.

Now, let us see if it would be possible to annotate `Undoable` and `Handle` with only the domain parameters `M` and `C` (See Figure 9) — the domain parameter `V` can then be supplied to `invokeStart()` as a method domain parameter.

Using a method domain parameter to annotate the formal parameter `v` could enforce the constraint that a developer should not store in a field the `DrawingView` object passed as an argument to `invokeStart()`, as in Figure 10. Of course, a developer could store the `DrawingView` object in a field of type `Object`, but that field would have to be cast to a `DrawingView` to be of any use.

Instead of a method domain parameter, the `lent` annotation could also be used to allow a temporary alias to an object within a method boundary. We found a few such examples in JHotDraw. Method `setAffectedFigures` in Figure 11 makes a copy of the `lent` argument so it cannot just hold on to it.

In fact, `lent` can be formally modeled as a method domain parameter. However, the type system does not allow a method to return a `lent` value but it allows a method to return an object in a method domain parameter. In the case of `DrawingView`, `lent` cannot be used because implementations of `invokeStart()` construct `Undoable` objects that maintain aliases to the `DrawingView` and thus require the `V` domain parameter.

For that same reason, the `Undoable` interface requires the `V` domain parameter because `Undoable` stores the `DrawingView` where the activity to be undone was performed in order to undo the changes to that view only. This may slightly violate the Model-View-Controller design, where model objects should not hold on to view objects, because there might be multiple views that need to be updated in response to changes in the model. At the same time, it would be counter-intuitive for a user to undo a change in one view and observe changes in some other view. Thus, ownership domain annotations expose the tighter coupling that the Undo feature introduced. Figure 11 shows in more detail the interaction between `Handle`, `Undoable` and `DrawingView`.

An earlier empirical study of JHotDraw mentioned that "a common architectural mistake [. . .] was to provide `Figures` with a reference to the `Drawing` or the `DrawingView`. `Figures` do not by default have any access to either the `Drawing` or the `DrawingView` in which they are contained. This prevents them from accessing information such as the size of the `Drawing`. However, it is possible to overcome this problem by passing the view into the constructor of a figure, which can then store and access this as required" [28]. Starting with Version 5.3, one could get to the `Figure`'s `Handles` through its `handles()` method then get a `DrawingView` through a `Handle`'s `UndoActivity` objects.

5.6 Ownership domains promote decoupling code

Ownership domain annotations highlight tight coupling and promote programming practices that decouple code.

Programming to an Interface. It is recommended to "refer to objects by their interfaces" [7, Item #34] since interfaces can reduce

```
@DomainParams({"M", "V", "C"})
@DomainInherits({"LocatorHandle<M,V,C>"})
class ResizeHandle extends LocatorHandle {
  @Override
  void invokeStart(int x, int y,
  @Domain("V<M,V,C>") DrawingView view) {
        setUndoActivity(createUndoActivity(view));
      ...
  }
  /**
   * Factory method for undo activity.
   * To be overriden by subclasses.
   */
  protected @Domain("M<M,V,C>")Undoable
  createUndoActivity(
                     @Domain("V<M,V,C>")DrawingView view) {
    @Domain("unique<M,V,C>")
    ResizeHandle.UndoActivity
    undoable = new ResizeHandle.UndoActivity(view);
    return undoable;
   }

    @DomainParams({"M", "V", "C"})
    @DomainInherits("UndoableAdapter<M,V,C>")
    static class UndoActivity extends UndoableAdapter {
    ...
    }
}
/**
 * Basic implementation for an Undoable activity
 */
@DomainParams({"M", "V", "C"})
@DomainInherits("Undoable<M,V,C>")
public class UndoableAdapter implements Undoable {
  @Domain("V<M,V,C>")DrawingView myDrawingView;

  UndoableAdapter(@Domain("V<M,V,C>")DrawingView dv) {
    setDrawingView(dv);
  }
 @Domain("V<M,V,C>") DrawingView getDrawingView() {
   return myDrawingView;
 }
 void setDrawingView(@Domain("V<M,V,C>")DrawingView dv) {
   myDrawingView = dv;
 }
 void setAffectedFigures(@Domain("lent<M>")FigureEnumeration fe) {
   // the enumeration is not reusable therefore a copy is made
   // to be able to undo-redo the command several time
   rememberFigures(fe);
 }
}
```

Figure 11. Concrete implementation class of `Handle`.

coupling between classes by splitting intent from implementation. When fewer domain parameters are needed to annotate an interface (as compared to the corresponding class), ownership domain annotations can enforce this idiom.

In particular, an implementation class can require a private ownership domain to be passed as an actual value for one its parameters. Since a private ownership domain cannot be named by an outside client, the client is then forced to use the interface which does not require these parameters.

For instance, in the earlier `Sequence` example (Figure 1), the `SeqIterator` class receives the `Sequence`'s private domain `owned` and hides the extra parameterization behind the `Iterator` interface. This forces a client of the `Sequence` to access the iterator objects only through the `Iterator` interface. A client may not even cast the `Iterator` reference to a `SeqIterator` class.

We used a similar technique to decouple the code in HillClimber (See Figure 12 for the inheritance hierarchy). The original implementation for class `HillNode` had a field reference of type `HillGraph`. However, `HillGraph` took the three domain parameters `ui`, `logic` and `data`, which required passing all those parameters to `HillNode`.

```
@DomainParams({"ui","logic","data"})
@DomainInherits({"Node<data>"})
class HillNode extends Node {
  @Domain("data<ui,logic,data>")HillGraph graph;
...
}
```

When adding annotations, an unexpected domain parameter often indicates unnecessary coupling, e.g., why should `HillNode` have access to the `ui` domain? Thus a lengthy domain parameter list can be an objective measure of a code smell [2]. Furthermore, ownership domain annotations can help a developer lower the coupling by suggesting which specific type declarations need to be generalized to shorten the list of domain parameters on the enclosing type.

In HillClimber, one solution was to extract an `IHillGraph` interface from class `HillGraph` that only requires the `data` domain parameter and make a `HillNode` object reference the `HillGraph` object through the `IHillGraph` interface. We decided against carrying this refactoring further and eliminating the `ui` and `logic` domain parameters on `HillGraph` itself.

Since the `HillGraph`, `HillNode`, etc., form a parallel inheritance hierarchy to `Graph`, `Node`, etc., a similar refactoring was performed on `Graph` by extracting a `IGraph` interface – even though `Graph` and `IGraph` are both parameterized by `data`.

```
@DomainParams({"ui","logic","data"})
@DomainInherits({"Graph<data>",
                 "IHillGraph<data>"})
class HillGraph extends Graph
                implements IHillGraph {
 ...
}
@DomainParams({"data"})
@DomainInherits({"IGraph<data>"})
interface IHillGraph extends IGraph {
...
}
@DomainParams({"data"})
@DomainInherits({"Node<data>"})
class HillNode extends Node {
  @Domain("data<data>")IHillGraph graph;
}
```

Tightly coupled code was observed throughout HillClimber. Similarly, we were surprised that a dialog class `FontDialog` required the `data` domain parameter. It turned out that `FontDialog` had a field reference declared with its most specific type `GraphCanvas`. In some cases, it is possible to generalize the type of the reference, e.g., use `java.awt.Frame` to eliminate the need for the domain parameter. However, `FontDialog` needed access to some of the `GraphCanvas` functionality, so a different solution was needed.

Mediator Pattern. Defining an interface is sometimes insufficient to decouple code since referring to an object through its interface still requires access to the domain the object is in. One solution is to use the Mediator design pattern [22, p. 273], as shown here.

In the original HillClimber implementation, `Node` obtained a reference to `GraphCanvas`, which violates the Law of Demeter [32], i.e., objects should only talk to their immediate neighbors:

```
@DomainParams({"data"})
abstract class Entity {
  @Domain("data<data>") Graph graph; // parent graph
...
}
@DomainParams({"data"})
@DomainInherits({"Entity<data>"})
class Node extends Entity
 ...
 int getHeight() {
   return graph.getCanvas().getFontMetrics()...;
  }
}
```

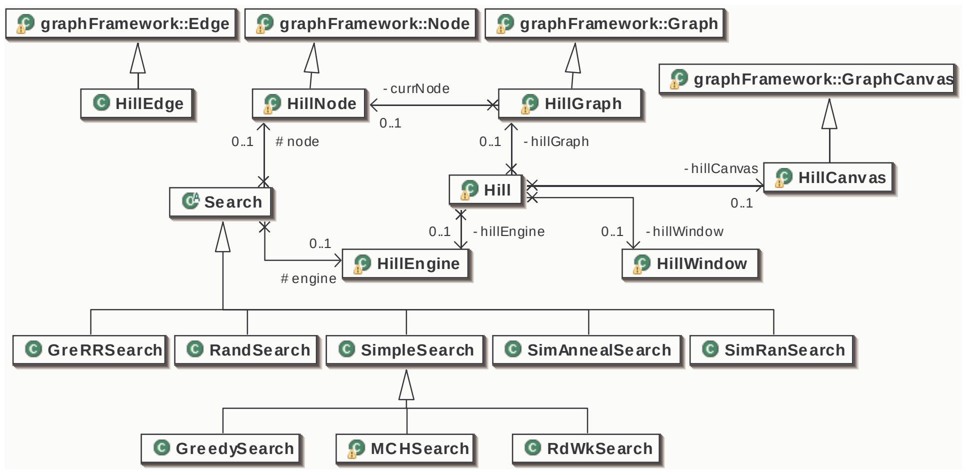

Figure 12. Partial UML Class Diagram for HillClimber obtained from the original implementation using Eclipse UML [39]. This diagram does not reflect some of the types introduced during refactoring, such as `IGraph`, `IHillGraph` and `ICanvasMediator`.

Extracting an interface from `GraphCanvas` would not work, as that reference would still need the `ui` domain parameter. Moreover, the implementation of `getFontMetrics()` could not be moved to `Graph` as it required access to objects in the `ui` domain.

```
@DomainParams({"data"})
abstract class Entity {
  @Domain("ui")IGraphCanvas canvas; // 'ui' unbound
...
}
```

A mediator was defined as follows:

```
/**
 * Mediator interface
 */
interface ICanvasMediator {
 @Domain("shared")FontMetrics getFontMetrics();
...
}
/**
 * Mediator implementation class
 */
@DomainParams({"ui","data"})
class MediatorImpl implements ICanvasMediator {
 @Domain("ui<ui,data>")GraphCanvas canvas;

 MediatorImpl(@Domain("ui<ui,data>")GraphCanvas c) {
  this.canvas = c;
 }
 @Domain("shared")FontMetrics getFontMetrics() {
   return canvas.getFontMetrics();
 }
...
}
```

`GraphCanvas` initializes the mediator:

```
@DomainParams({"ui","data"})
class GraphCanvas extends ... {
 @Domain("data<ui,data>")MediatorImpl mediator;
 ...
 @Domain("data")ICanvasMediator getMediator() {
    return mediator;
 }
}
```

`Entity` and `Node` can then use the mediator as follows:

```
@DomainParams({"data"})
abstract class Entity {
  @Domain("data") ICanvasMediator mediator;
...
}
```

```
/**
 * DrawApplication defines a standard presentation
 * for standalone drawing editors
 */
@DomainParams({"M", "V", "C"})
@DomainInherits({"DrawingEditor<M,V,C>", ...)
class DrawApplication implements DrawingEditor... {
  // Opens a new window with a drawing view.
  @DomainReceiver("unique")
  protected void open(...) {
    fIconkit = new Iconkit(this);
    ...
  }
}
class Iconkit {
  static @Domain("unique")Iconkit fgIconkit = null;

  // Constructs an Iconkit that uses the given editor
  // to resolve image path names.
  @DomainReceiver("unique")
  public Iconkit(@Domain("unique")Component comp){
    fgIconkit = this;
    ...
  }
}
```

Figure 13. Annotating a singleton using `unique`.

```
@DomainParams({"data"})
@DomainInherits({"Entity<data>"})
class Node extends Entity {
  int getHeight() {
    return getMediator().getFontMetrics()...;
  }
```

5.7 Ownership domains can help identify singletons

While adding ownership domain annotations, we discovered a curious instance of the Singleton design pattern: `IconKit`'s constructor was not private, although it had a static `instance()` method. Indeed, there is a `unique` instance of `DrawingEditor` (the application itself) and a `unique` `IconKit` (See Figure 13) at runtime.

6. Expressiveness Challenges

In this section, we discuss some of the expressiveness gaps that we encountered, some of which had been previously mentioned.

```
class DrawApplication implements DrawingEditor ... {
...
class MDI_DrawApplication extends DrawApplication ...{
...
@DomainParams({"M", "V", "C"})
@DomainInherits({"MDI_DrawApplication<M,V,C>"})
class JavaDrawApp extends MDI_DrawApplication {
...
@Domains({"Model", "View", "Controller"})
class Main {
  @Domain("View<Model,View,Controller>")
  JavaDrawApp app = new JavaDrawApp();

  public static void main(
       @Domain("lent[shared]")String args[]) {
       @Domain("lent")Main system = new Main();
  }
}
```

Figure 14. Defining the top-level domains in a separate class.

6.1 An object cannot be in more than one ownership domain

Ownership domains, as most other ownership type systems, support only *single ownership*, i.e., an object cannot be part of more than one ownership hierarchy. Proposals for *multiple ownership* [11] lift this restriction in other type systems. Ownership domains do not support *ownership transfer* [31] either, i.e., an object's owner does not change — only unique objects can flow between any two domains. As a result, many fine-grained ownership domains cannot be defined to represent multiple roles in design patterns: e.g., if an object is both a mediator in the Mediator pattern and a view in the Model-View-Controller pattern, it cannot be in both a Mediator ownership domain and a View ownership domain at the same time.

For instance, creating top-level ownership domains to correspond to the design in Figure 6 would have been more challenging than creating the three top-level domains for Model, View and Controller: placing a DrawingEditor object in a Mediator domain would have prohibited it from also being in the View domain.

6.2 An object cannot place itself in a domain it declares

An object cannot place itself in an ownership domain that it declares. This is problematic for the root application object, i.e., the JavaDrawApp instance (JavaDrawApp extends DrawApplication which in turn extends DrawingEditor). True to form, we solve this problem with an extra level of indirection by creating a fake top-level class Main to declare the Model, View and Controller top-level ownership domains and declare the JavaDrawApp object in the View domain (See Figure 14).

6.3 Public domains are hard to use

Public domains make the ownership domains type system more flexible than *owner-as-dominator* type systems [15]. Also, public domains are ideal for visualization because placing an object inside a public domain of another object relates these objects without cluttering the top-level domains [1]. However, public domains are typically hard to use without refactoring the code. We started using them in a few cases but quickly abandoned those attempts.

Since the Observer design pattern tends to make communication between objects implicit, we attempted to represent listeners more explicitly using ownership domain annotations. For instance, it might make sense to create a public domain LISTENERS as a domain to hold the Listener objects that an Observer will notify — a Listener often needs special access to the Observer, but usually does not need special access to the Subject.

JHotDraw uses a delegation-based event model: for instance, a DrawingView calls method figureSelectionChanged to notify a FigureSelectionListener observer of selection changes. So it might make sense to declare a FIGURESELECTIONLISTENERS

```
/**
 * DrawingView renders a Drawing and listens to its
 * changes. It receives user input and delegates
 * it to the current Tool.
 */
@DomainParams({"M", "V", "C"})
@DomainInherits({"DrawingChangeListener<M,V>"})
interface DrawingView extends DrawingChangeListener... {
  // Add a listener for selection changes
  void addFigureSelectionListener(
   @Domain("?<M,V,C>")FigureSelectionListener fsl);
...
}
@Domains({"owned"})
@DomainParams({"M", "V", "C"})
@DomainInherits({"DrawingView<M,V,C>"})
class StandardDrawingView implements DrawingView... {
  // Registered list of listeners for selection changes
  private @Domain("owned<?<M,V,C>>")
  Vector<FigureSelectionListener> fSelectionListeners;

  StandardDrawingView(
    @Domain("V<M,V,C>")DrawingEditor editor, ...) {
    // editor is in 'V' domain parameter, not 'C'!
    addFigureSelectionListener(editor);
    ...
  }
  // Add a listener for selection changes.
  // Command implements FigureSelectionListener
  // but Command is in the 'C' domain parameter!
  void addFigureSelectionListener(
    @Domain("?<M,V,C>")FigureSelectionListener fsl) {
    fSelectionListeners.add(fsl);
  }
```

Figure 15. How to annotate addFigureSelectionListener?

public domain on Command to hold the FigureSelectionListener objects. But Command implements FigureSelectionListener, so a Command *is-a* FigureSelectionListener. Thus a Command object cannot split a part of itself and place it in the public domain FIGURESELECTIONLISTENERS that it declares.

6.4 Listener objects are particularly challenging

There were additional complications when trying to highlight the event subsystem in JHotDraw using ownership domain annotations. Command, which is in the Controller domain, implements FigureSelectionListener, and so does DrawingEditor, which is in the View domain.

Consider method addFigureSelectionListener in (See Figure 15). How would one annotate the formal parameter FigureSelectionList
The parameter should support both annotations C<M,V,C> and V<M,V,C>. Existential ownership [13, 29, 34] may be the answer to increase the expressiveness, e.g., by annotating the parameter with "any" [34]. Other problems of adding ownership domains annotations to listeners had been previously identified [44].

6.5 Static code can be challenging

Even in such a well-designed program as JHotDraw, we found a few instances where ownership annotations cannot be made to typecheck. In particular, in Figure 16, the static Hashtable cannot have the M, V, and C domain parameters because the domain parameters declared on the class NullDrawingView are not in scope for static members. Static members can only be annotated with shared or unique, and these values cannot flow to the Mx, Vx or Cx method domain parameters.

Annotating the generic Hashtable also requires nested parameters: Hashtable has three domain parameters for its keys, values and entries. Both DrawingView and DrawingEditor take M, V, and C as parameters. Although the number of annotations seems excessive and maybe argues in favor of generic ownership [41], the

```
@DomainParams({"M", "V", "C"})
@DomainInherits({"DrawingView<M,V,C>"})
class NullDrawingView implements DrawingView ... {
  static @Domain("unique<?<?,?,?>,?<?,?,?>,?>")
  Hashtable<DrawingEditor,DrawingView> dvMgr = ...;

  @DomainParams({"Mx","Vx","Cx"})
  public synchronized static @Domain("Vx<Mx,Vx,Cx>")
  DrawingView getManagedDrawingView(
        @Domain("Vx<Mx,Vx,Cx>")DrawingEditor ed) {
    if (dvMgr.containsKey(ed)) {
      @Domain("Vx<Mx,Vx,Cx>")
      DrawingView dv = dvMgr.get(ed);
      return dv;
    }
    ...
  }
```

Figure 16. How to annotate objects that are stored in static fields?

ownership domains for the Hashtable key, value and entries need not correspond to the M, V and C ownership domains.

A solution that is not type-safe would be to store the Hashtable as Object, then cast down to a Hashtable upon use — the equivalent of raw types but without re-implementing them in the ownership domains type system. Another solution would be to refactor the program to eliminate this static field since it gives any object access to all the DrawingView and DrawingEditor objects. Since it is often unrealistic to perform such a significant refactoring, maybe the best solution would be to support package-level static ownership domains, similar to confined types [9].

6.6 Annotations may be unnecessarily verbose

Ownership domain annotations tend to be verbose: e.g., formal method parameters need to be fully annotated even if they are not used in the method body or used in a restricted way. This produces particularly unwieldy annotations for containers of generic types.

In Figure 17, method clearStackVerbose indicates the current level of annotations needed. It should be possible to leave out domain parameters when they are not really needed. This may involve using implicit existential ownership types as in clearStackAny: i.e., there exists some domain parameters d1, d2, d3, d4, such that the formal method parameter s could be annotated with lent<d1<d2,d3,d4>>. Using appropriate defaults, the annotations could probably be reduced to the level needed to annotate a raw type, as shown in clearStack.

6.7 Manifest ownership can reduce the annotation burden

The current defaulting tool only adds the shared annotation to String objects. However, during the annotation process, we found ourselves adding the shared annotation to many other types such as Font, FontMetrics, Color, etc. Specifying a per-type default globally and not for every instance, as in *manifest ownership* [13], would have reduced the annotation burden.

6.8 Reflective code cannot be annotated

JHotDraw uses reflective code to serialize and deserialize its state and such code cannot be annotated using ownership domains [6].

6.9 Annotate Exceptions as lent

We annotated exceptions with lent since we were not particularly interested in reasoning about them. However, richer annotations are possible [45].

7. Related Work

Case studies applying ownership type systems on real code are few and far between. Hächler [25] documented a case study applying

```
@Domains({"owned"})
@DomainParams({"M","V","C"})
public class UndoManager {
  /**
   * Collection of undo activities
   */
  @Domain("owned<M<M,V,C>>")Vector<Undoable> undoStack;

  void clearStackVerbose(
    @Domain("lent<M<M,V,C>>")Vector<Undoable> s) {
    s.removeAllElements();
  }

  void clearStackAny(
    @Domain("lent<?<?,?,?>>")Vector<Undoable> s) {
    s.removeAllElements();
  }

  void clearStack(
    @Domain("lent")Vector<Undoable> s) {
    s.removeAllElements();
  }
}
```

Figure 17. Reducing annotations when they are not really needed.

Universes [36, 17] on an industrial software application and refactoring the code in the process. Although the subject system in the case study is larger than JHotDraw (around 55,000 lines of code), the author annotated only a portion of the system. The author manually generated visualizations of the ownership structure whereas we had access to tool support to visualize the ownership structure and adjust the annotations accordingly [1].

Nägeli [38] evaluated how the Universes and Ownership Domains type systems express the standard object-oriented design patterns [22]. However, in real world complex object-oriented code, design patterns rarely occur in isolation [43]. As we discussed earlier, these subtle interactions, combined with the single ownership constraint of the type system, make the annotations difficult.

In a previous case study, we re-engineered HillClimber using ArchJava [4] to specify a component-and-connector architecture in code and ownership domain annotations to specify the data sharing [2]. In the earlier case study, we performed refactorings similar to the ones described here. However, adding ownership domain annotations to the ArchJava program seemed easier. Indeed, ArchJava's port construct effectively reduces coupling; in the plain Java implementation, the same effect had to be achieved using programming to interfaces, using mediators, etc.

ArchJava's properties are available at the expense of various restrictions on object-oriented implementations. The previous case study also identified that adding ownership domain annotations required less effort than encoding the architectural structure in ArchJava [2, 6]. Fewer defects are introduced since code that passes object references need not be changed and the ownership annotations need not affect the runtime semantics of the program. Moreover, the ownership domain annotations, while tedious to add manually, are relatively straightforward once the top-level domains are decided, compared to re-engineering to use ArchJava.

Adding ownership domains annotations manually still required significant effort, and researchers are still looking at scalable inference of ownership domain annotations [6, 16]. Current inference techniques [35, 33] however only infer the equivalent of owned, shared, lent and unique annotations, i.e., they assume a strict owner-as-dominator hierarchy which is not flexible enough to represent many design patterns. Some approaches do not map the results of the analysis back to an ownership type system [35, 33]. A fully automated inference cannot create multiple public domains in one object and meaningful domain parameters, which are critical for representing the abstract design intent, as in the three top-level

Model, View, and Controller domains in JHotDraw. Existing inference algorithms often generate imprecise annotations, producing for each class a long list of domain parameters, often placing each field in a separate domain, and annotating many more objects as shared or lent than necessary [6, 16].

8. Conclusion

We presented an annotation-based system that re-implements the ownership domains type system as a set of Java 1.5 annotations, using the Eclipse infrastructure. Using annotations imposes many restrictions and requires changing the code slightly to add annotations to it, and the annotation language does take some getting used to. Still, the annotation-based system is an improvement over custom infrastructure, language extensions, and the resulting limited tool support: it enabled us to annotate larger object-oriented programs "in the wild" to study how ownership domains can express and enforce design intent related to object encapsulation and communication and to identify expressiveness limitations.

In future work, we plan on making the type system more flexible and extending the annotation language in a non-breaking way.

Acknowledgments

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

References

[1] M. Abi-Antoun and J. Aldrich. Compile-Time Views of Execution Structure Based on Ownership. In *Intl. Workshop on Aliasing, Confinement and Ownership*, 2007.

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

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

[4] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *ICSE*, 2002.

[5] J. Aldrich and D. Dickey. The Crystal Data Flow Analysis Framework 2.0. http://www.cs.cmu.edu/~aldrich/courses/654/, 2006.

[6] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, 2002.

[7] J. Bloch. *Effective Java*. Addison-Wesley, 2001.

[8] B. Bokowski and A. Spiegel. Barat — A Front–End for Java. Technical Report B-98-09, Freie Universität Berlin, 1998.

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

[10] C. Boyapati, B. Liskov, and L. Shrira. Ownership Types for Object Encapsulation. In *POPL*, 2003.

[11] N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple Ownership. In *OOPSLA*, 2007. To appear.

[12] H. B. Christensen. Frameworks: Putting Design Patterns into Perspective. In *SIGCSE Innov. & Tech. in Comp. Sci. Ed.*, 2004.

[13] D. Clarke. *Object Ownership & Containment*. PhD thesis, University of New South Wales, 2001.

[14] D. Clarke and T. Wrigstad. External Uniqueness is Unique Enough. In *ECOOP*, 2003.

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

[16] W. Cooper. Interactive Ownership Type Inference. Senior Thesis, Carnegie Mellon University, 2005.

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

[18] A. D. Eisenberg and G. Kiczales. Expressive Programs through Presentation Extension. In *AOSD*, 2007.

[19] M. D. Ernst and D. Coward. JSR 308: Annotations on Java types. http://pag.csail.mit.edu/jsr308/, 2006.

[20] Universes Tools. www.sct.ethz.ch/research/universes/tools/, 2007.

[21] R. M. Fuhrer, F. Tip, A. Kiezun, J. Dolby, and M. Keller. Efficiently Refactoring Java Applications to Use Generic Libraries. In *ECOOP*, 2005.

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

[23] Gamma, E. et al. JHotDraw. http://www.jhotdraw.org/, 1996.

[24] G. Goth. Beware the march of this IDE: Eclipse is overshadowing other tool techniques. *IEEE Software*, 22(4), 2005.

[25] T. Hächler. Applying the Universe Type System to an Industrial Application: Case Study. Master's thesis, ETH Zurich, 2005.

[26] JavaCC. https://javacc.dev.java.net/, 2006.

[27] JSR 175: A Metadata Facility for the Java Programming Language. http://jcp.org/en/jsr/detail?id=175, 2006.

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

[29] N. Krishnaswami and J. Aldrich. Permission-Based Ownership: Encapsulating State in Higher-Order Typed Languages. In *PLDI*, 2005.

[30] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and Verification Challenges for Sequential Object-Oriented Programs. *Formal Aspects of Computing*, 2007. Submitted.

[31] K. R. M. Leino and P. Müller. Object Invariants in Dynamic Contexts. In *ECOOP*, 2004.

[32] K. J. Lieberherr and I. M. Holland. Assuring Good Style for Object-Oriented Programs. *IEEE Software*, 6(5), 1989.

[33] Y. Liu and A. Milanova. Ownership and Immutability Inference for UML-based Object Access Control. In *ICSE*, 2007.

[34] Y. Lu and J. Potter. Protecting Representation with Effect Encapsulation. In *POPL*, 2006.

[35] K.-K. Ma and J. S. Foster. Inferring Aliasing and Encapsulation Properties for Java. In *OOPSLA*, 2007. To appear.

[36] P. Müller and A. Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, 1999.

[37] G. C. Murphy, M. Kersten, and L. Findlater. How are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4), 2006.

[38] S. Nägeli. Ownership in Design Patterns. Master's thesis, Department of Computer Science, Federal Institute of Technology Zurich, 2006.

[39] Omondo. EclipseUML. http://www.omondo.com/, 2006.

[40] A. Potanin. Ownership Generic Java. www.mcs.vuw.ac.nz/~alex/ogj/, 2005.

[41] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic Ownership for Generic Java. In *OOPSLA*, 2006.

[42] Annotation File Utilities. http://pag.csail.mit.edu/jsr308/annotation-fil 2006.

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

[44] J. Schäfer and A. Poetzsch-Heffter. Simple Loose Ownership Domains. In *FTfJP*, 2006.

[45] D. Werner, , and P. Müller. Exceptions in Ownership Type Systems. In *FTfJP*, 2004.

[46] ArchJava. http://www.archjava.org/, 2007.