

Compile-Time Views of Execution Structure Based on Ownership

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

A developer often needs to understand both the code structure and the execution structure of an object-oriented program. Class diagrams extracted from source are often sufficient to understand the code structure. However, existing static or dynamic analyses that produce raw graphs of objects and relations between them, do not convey design intent or readily scale to large programs.

Imposing an ownership hierarchy on a program’s execution structure through ownership domain annotations provides an intuitive and appealing mechanism to obtain, at compile-time, a visualization of a system’s execution structure. The visualization conveys design intent, is hierarchical, and thus is more scalable than existing approaches that produce raw object graphs.

We first describe the construction of the visualization and then evaluate it on two real Java programs of 15,000 lines of code each that have been previously annotated. In both cases, the automatically generated visualization fit on one page, and gave us insights into the execution structure that would be otherwise hard to obtain by looking at the code, at existing class diagrams, or at unreadable visualizations produced by existing compile-time approaches.

1. Introduction

When modifying an object-oriented program, both the code structure (static hierarchies of classes) and the execution structure (dynamic networks of communicating objects) must be understood. “For a developer unfamiliar with the system to obtain this understanding is a non-trivial task. Little work has been done on minimizing this learning curve” [38].

In many cases, developers cannot rely that external design documentation is up-to-date. Many tools can automatically generate class diagrams from program source [21]. However, a class diagram shows the code structure and does not explain the execution structure of the system. In object-oriented design patterns, much of the functionality is determined by what instances point to what other instances. For instance, in the Observer design pattern [15, p. 293], understanding “what” gets notified during a change notification is crucial for the function of the system, but “what” does not usually mean a class, “what” means a particular instance. Furthermore, a class diagram often shows several classes depending on a single container class such as `java.util.ArrayList`. However, different instantiations of such a class often correspond to different elements in the design, hence the need for an instance-based view to complement a class diagram.

A running object-oriented program can be represented as an *object graph*: nodes correspond to objects and edges correspond to relations between objects. Existing dynamic analyses can describe the runtime object graph of a system for a particular set of inputs and exercised use cases [12, 33]. Obtaining at compile time a finite and conservative abstraction of all possible runtime object graphs is more challenging because of aliasing, precision and scalability

issues. Static analyses [29, 40] that approximate the runtime object graph often produce large non-hierarchical graphs that do not convey design intent and do not scale to large programs (See visualizations [2] for examples).

Many type systems enforce *ownership* at compile time, i.e., make one object part of another object’s representation [8, 7, 3, 11]. In the ownership domains type system [3], each object contains one or more public or private *ownership domains* — conceptual groups of objects — and each object is in exactly one domain. As with most other ownership type systems, adding ownership domain annotations to a program’s source code can control aliasing and enforce *instance encapsulation* which is stronger than module visibility mechanisms. Moreover, ownership domains can express and enforce a tiered runtime architecture by representing a tier as an ownership domain. A *domain link* can abstract permissions of when objects can communicate [1].

Our contribution in this paper is to leverage ownership domain annotations to obtain at compile-time a sound visualization of the execution structure of a program with ownership domain annotations, the Ownership Object Graph. The visualization is hierarchical, conveys design intent and compares favorably with existing compile-time visualizations of two previously annotated Java programs, each consisting of 15,000 lines of code.

Currently, annotations are added mostly manually, however, active work in the area of semi-automated annotation inference [4, 9, 24, 25] promises to lower the annotation overhead. The visualization reflects the annotations, and the quality of the visualization reflects the quality of the annotations. The design intent is expressed by choosing the ownership domains and their structure, then adding annotations to the program — currently manually.

The ideas and techniques of ownership are fundamental for obtaining such a compile time visualization. First, ownership domains provide a coarse-grained ownership structure of an application with a granularity larger than an object or a class [37]. Second, ownership organizes a flat object graph into an ownership tree, and hierarchy is needed to achieve scalability and attain both high-level understanding and detail. Third, different ownership domains and different places in the hierarchy provide precision about inter-domain aliasing and conservatively describe all aliasing that could take place at runtime. Since two objects in two different domains cannot be aliased, the analysis can distinguish between instances that would be merged in a class diagram, allowing better understanding of the runtime structure of the system. Fourth, ownership domain names are specified by a developer and therefore can convey more design intent than the aliasing information obtained using a static analysis that does not rely on annotations [34].

We first define the Ownership Object Graph (Section 2) and describe the algorithm to construct it at compile time (Section 3). We then present concrete and in-depth examples of the visualization of two real annotated 15,000-line object-oriented programs (Section 4). Finally, we survey related work in Section 5 and conclude.

2. The Ownership Object Graph

This section discusses the challenges in visualizing an annotated program and describes the different intermediate representations we used to obtain the visualization.

A running object-oriented program can be represented as a *runtime object graph*: nodes correspond to *runtime objects* and edges correspond to relations between runtime objects such as creation, usage and reference [32]. The aim is to statically approximate all of the runtime object graphs that may be generated in any run of the program. The goals of the visualization are as follows:

- **Scalability**: to support high-level understanding, the visualization groups runtime objects into relatively few top-level “abstract” elements, each represented by a canonical object;
- **Hierarchy**: to provide detailed understanding, the visualization supports the ability to show the substructure of an abstract element. Thus the visualization can be viewed as a hierarchical tree of objects;
- **Design Intent**: the visualization groups runtime objects into clusters that are meaningful abstractions — e.g., that an object is in a tier — and documents design-level constraints using domain links — e.g., that two tiers may communicate. The user provides the design intent regarding object encapsulation and communication using ownership domain annotations [1];
- **Soundness**: to ensure that the visualization is a faithful representation of the runtime object graph, it must be *sound*. In particular, all objects and relations present at runtime should be represented. Furthermore, if two variables may alias at runtime, they should appear in the graph as a single “abstract” element.

The analysis builds two intermediate representations, an *abstract graph*, which is converted into a *visual graph*, which is then displayed as the Ownership Object Graph.

2.1 Abstract Graph

The *abstract graph* is built from ownership domain annotations in the source code (Figure 1). The syntax for declaring and using ownership domains follows that used for Java generics [3].

For each type in the program, the abstract graph shows the ownership domains declared in it, and shows field and variable declarations as *abstract objects* declared inside *abstract domains*. The abstract graph provides scalability through ownership hierarchy and captures design intent as described above, but is not adequate for visualization for several reasons (See Figure 2).

First, the abstract graph is not really hierarchical in the sense of an object having children; rather, an object has a type and the type has domains and the domains have object children. Second, it does not include all objects: a domain contains abstract objects only for the locally declared fields, but if that domain is passed as a domain parameter to another object, and that object declares its fields in that domain, those non-local fields will not be represented. Third, it does not show all aliasing: different field declarations — and therefore different abstract objects, could be aliased and thus must be shown as one. To realize the properties above, the abstract graph is converted into a *visual graph*.

2.2 Visual Graph

The visual graph is an intermediate representation which instantiates the types in the abstract graph and shows only objects and domains: each *visual object* contains *visual domains* and each *visual domain* contains *visual objects*. Thus, in the visual graph, one can view the children of an object without going through its declared type. Furthermore, to support the visualization goals listed earlier, the construction of the visual graph takes into account *object merging*, *object pulling* and *type abstraction*.

We visualize ownership domains as follows: a dashed border white-filled rectangle represents an actual ownership domain. A

```
class Branch< CUSTOMERS > /* Formal domain parameter */ {
    public domain TELLERS, VAULTS;
    link TELLERS -> VAULTS;

    CUSTOMERS Customer c1;
    TELLERS Teller t1;
    TELLERS Teller t2;
    VAULTS Vault v1;
    VAULTS Vault v2;
}
class Bank {
    domain owned; /* Private default domain */

    /* Bind Branch<CUSTOMERS> formal to 'owned' actual */
    owned Branch<owned> b1;
}
```

Summary of syntax for ownership domains annotations [3]:

d T o: declare object o of type T in domain d;
[public] domain a: declare private [or public] domain;
class C<d>: declare formal domain parameter d on class C;
C<actual> cObj: provide actual for formal domain parameter;
link b -> d: give domain b permission to access domain d;

Figure 1. Ownership domains illustrated with a simplified Bank system [3]. Branch declares two domains, TELLERS for Teller objects and VAULTS for Vault objects. Branch also declares a domain link from the TELLERS domain to the VAULTS domain to allow Teller objects to access Vault objects. Branch also takes a CUSTOMERS formal domain parameter to hold Customer objects. Bank references a Branch object in field b1, binding the CUSTOMERS formal domain of Branch to the Bank’s own private domain owned.

solid border grey-filled rectangle with a bold label represents an object. A dashed edge represents a link permission between two ownership domains. A solid edge represents a creation, usage, or reference relation between two objects. An object labeled “obj : T” indicates an object of type T as in UML object diagrams.

Object Merging. In the visual graph, a canonical visual object is created to represent all the abstract objects of a given type in a given source-level domain declaration. Two abstract objects in the same domain in the abstract graph, if related by inheritance, could indeed refer to the same runtime object, and thus are merged for soundness. In general, this object may summarize multiple runtime objects. For the annotated code in Figure 1, the visual graph in Figure 3 merges into one visual object (labelled with t1: Teller)

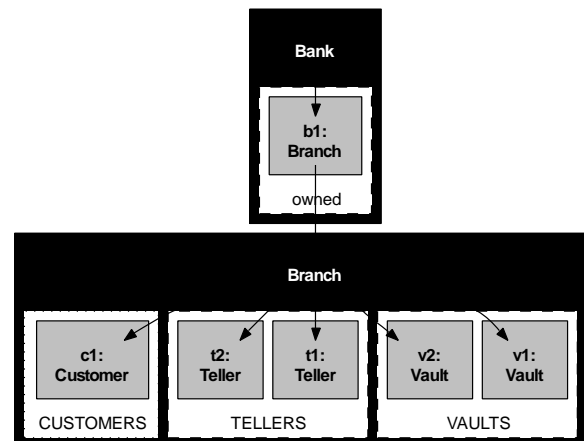


Figure 2. The abstract graph for the Bank system. A black-filled box represents a type, with white-filled domains declared inside it and grey objects declared inside each domain.

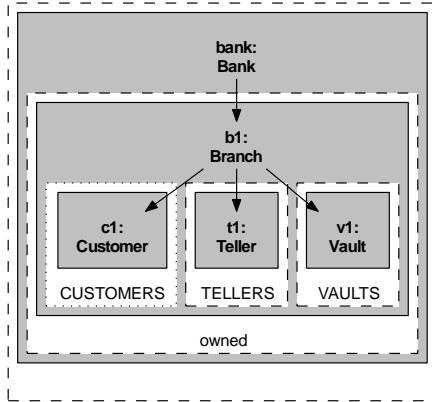


Figure 3. The visual graph for a Branch object *without* pulling: objects *t1* and *t2* are merged in domain TELLERS, and similarly, objects *v1* and *v2* in domain VAULTS. Object *c1* is shown in the formal domain parameter CUSTOMERS (dotted border).

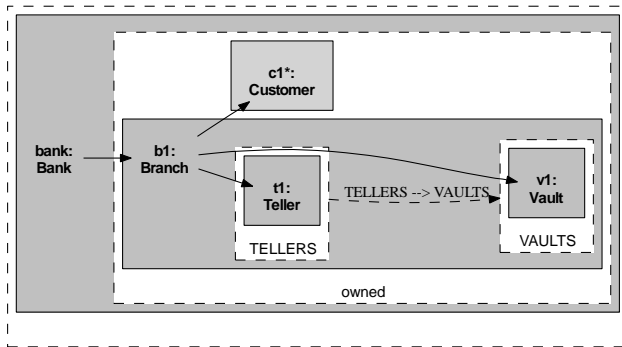


Figure 4. Object *c1** was pulled from the formal domain parameter CUSTOMERS in Figure 3 into the actual domain `Bank.owned` to which it is bound. The dashed edge represents a domain link between TELLERS and VAULTS.

the abstract objects *t1* and *t2* declared in domain TELLERS since they have the same declared type.

Merging objects of the same declared type that are in the same domain may be imprecise. For instance, two `Vector` objects in the same domain would get merged even if they are never aliased. Our analysis remains more precise than a class diagram which also summarizes objects by type, because the type system guarantees that two objects that are in two different domains can never be aliased. In some cases, adding generic types where applicable, e.g., for generic containers, can minimize excessive merging.

A developer can also prevent merging by placing two objects that should never get merged in separate domains, e.g., by defining two domains `CASHVAULT` and `GOLDVAULT` to store *v1* and *v2* in Figure 1 instead of using a single domain VAULTS.

Object Pulling. The abstract graph may display an object only in the domain where the domain is declared as a formal parameter. But in the visual graph, each runtime object that is actually in a domain must appear where that domain is declared. To ensure this property of visual graphs, an abstract object declared inside a formal domain is *pulled* into each domain that the formal domain is transitively bound to. Figure 3 shows object *c1* in the formal domain parameter CUSTOMERS (dotted border). In Figure 4, object *c1* — marked with * — was pulled from the formal domain CUSTOMERS in Branch to the actual domain `owned` in Bank (the former is bound to the latter using the annotation `Branch<owned>` on field *b1* in Figure 1).

Type Abstraction. For soundness, it may be necessary to merge abstract objects of different but compatible declared types. For example, consider the classes from the Java Abstract Window Toolkit (AWT) library in Figure 5. A variable of type `Window` and a different variable of type `Frame` in the same domain may alias each other, the corresponding abstract objects must therefore be merged for soundness.

In addition, it may be useful to do further heuristic merging to improve abstraction and reduce clutter in the graph. For example, if abstract objects of type `Button`, `Panel` and `Frame` were declared in the same domain, it may make sense to merge them into a single visual object of type `Component` or `Accessible`. On the other hand, merging can be taken too far: merging all the abstract objects in a domain into a single visual object of type `java.lang.Object` would result in a trivial and uninteresting visual graph. Thus, we heuristically merge abstract objects whenever they share one or more non-trivial *least upper bound* types. The resulting visual object is marked as having an intersection type that includes all the least upper bounds. In the example above, the least upper bound would be the intersection of the set `{Component, Accessible}`.

The definition of “trivial” is user-configurable; typically types such as `Object` and `Serializable` are trivial, and so abstract objects which share these as a supertype are not merged according to this heuristic. Again, a developer controls this heuristic by adding or removing types from the list of trivial types.

Instantiation-Based View. Merging abstract objects based on non-trivial least-upper-bound types can sometimes lead to unwanted merging. For instance, in the JHotDraw case study discussed in Section 4.2, both interfaces `Command` and `Tool` are in the same `Controller` domain and both extend the same interface `ChangeListener`. As a result, the abstract objects for `Command` and `Tool` get merged into the same visual object unless interface `ChangeListener` is added to the list of trivial types. However, this would not work since several variables have `ChangeListener` as their declared type.

The key insight however is that there are no object allocations of the interface `ChangeListener` since an interface cannot be instantiated directly. As an alternative to merging abstract objects, it is possible to achieve soundness by scanning object allocations instead of field and variable declarations, and then only adding visual objects for types that are actually instantiated and not the ones that are just declared. This technique is similar to how Rapid Type Analysis (RTA) [5] determines the receiver of a method call during the construction of a call graph.

In the example above, if the analysis encounters an object allocation of a `Tool` object but never that of a `ChangeListener` object, the analysis would only create a visual object for `Tool`, and similarly for `Command`, thus achieving the desired effect of keeping `Command` and `Tool` distinct. This solution can also prevent merging all the abstract objects in a domain into a single visual object of

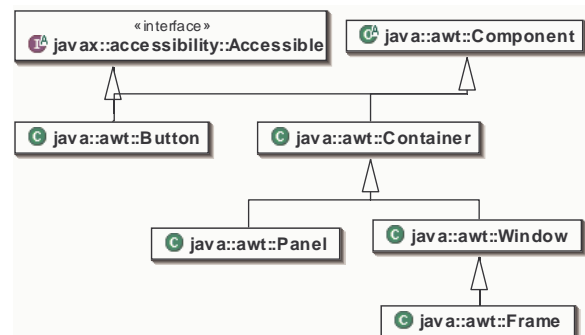


Figure 5. Type hierarchy excerpts from AWT.

type `java.lang.Object`. If the analysis does not encounter an allocation expression of the form `new Object()` in the code, it never creates a visual object for the `java.lang.Object` abstract type.

A class hierarchy analysis could determine that a variable of type `ChangeListener` could alias a variable of type `Object` — of course, an alias analysis could do better. A newly allocated object can be considered un-aliased or *unique* [3]. A standard flow analysis can track the flow of an object from its point of creation to the point at which it is first assigned to an ownership domain.

Design Intent Types. Since the visualization is instance-based, labelling instances is important for conveying design intent. A visual object can merge one or more abstract objects, and each abstract object has an abstract type corresponding to a declared type in the program. A visual object is labelled “`obj: T`” as in UML object diagrams — where `obj` is an optional instance name and `T` is an optional type name. An abstract object maintains the field name or variable name in the program. `obj` is selected from one of the abstract objects merged into a visual object. `T` is a list of least upper bound types as discussed above. The user can optionally specify a list of informative *design intent* types. A *design intent type* is the preferred abstract type used to label a visual object. A trivial type is not used in the label unless it occurs as a declared type in the program. Design intent types do not affect the soundness of the Ownership Object Graph and are just for labelling.

2.3 Ownership Object Graph

A visual object can contain itself so the visual graph must represent a potentially unbounded runtime object graph with a finite graph. For example, consider a class `C` which declares a domain `d` and a field of type `C` in domain `d`:

```
class C {
    domain d; /* Declare domain d */
    d C f;
}
```

Since there is a unique canonical object for each type in each domain, the object representing `C` in domain `d` must also represent the child object of type `C` in domain `d` of the parent; it is therefore its own parent in this representation. A finite representation is essential to ensure that the analysis terminates, but we want to show the user a hierarchical view where no object is its own parent. We therefore compute the Ownership Object Graph as a finite, depth-limited, unrolling of the visual graph. In the example above, we would show one `C` object within another down to a finite depth.

To summarize, an Ownership Object Graph is a graph with two types of nodes, objects and domains. The nodes form a hierarchy where each object node has a unique parent domain and each domain node has a unique parent object. The root of the graph is a top-level domain. In addition, the Ownership Object Graph has the object merging, object pulling and type abstraction properties. Finally, there are two kinds of edges: edges between objects correspond to object creation, usage and reference relations, and edges between domains correspond to domain links. Compared to earlier definitions of object graphs [32], the Ownership Object Graph explicitly represents clusters of nodes, i.e., domains, and edges between these clusters, i.e., domain links.

2.4 Soundness

For the Ownership Object Graph to be most useful, it should be a *sound* approximation of the true runtime object graph for any possible run of the program. In this section, we only present an operational definition of the soundness of the Ownership Object Graph and leave a proof of soundness for future work.

Intuitively, soundness means that every object, domain, and edge in the runtime object graph is represented in the Ownership Object Graph. However, the Ownership Object Graph may be an

approximation of the true runtime object graph, as it may represent multiple runtime objects with a single visual object, and similarly for domains and edges. The following invariants relate the Ownership Object Graph to the runtime object graph:

- **Unique Representatives:** Each object in the runtime object graph is represented by exactly one object in the visual graph. Similarly, each domain in the runtime object graph — as defined in the dynamic semantics of ownership domains [3, p. 15], is represented by exactly one domain in the visual graph;
- **Edge Soundness:** If there is a field reference from object o_1 to object o_2 in the runtime object graph, then there is a field reference edge between visual objects θ_1 and θ_2 in the visual graph, corresponding to o_1 and o_2 — similarly for domain links and edges;
- **Ownership Soundness:** If object o is in domain d in the runtime object graph, then object θ (corresponding to o) is in domain δ (corresponding to domain d) in the visual graph. Similarly, if o declares domain d in the abstract graph, then θ declares domain δ in the visual graph.

The Ownership Object Graph inherits other properties that are guaranteed by the soundness of the underlying ownership system — for example, that every object is assigned an owning domain which is consistent with all program annotations and does not change over time. These invariants are correct up to the following assumptions:

- **All Sources Available:** The program’s whole source code is available, and the program operates by creating some main object and calling a method on it (this justifies the Ownership Object Graph’s focus on a single root object, although multiple root objects could in principle be shown). The class of that main object is the type of the root of the Ownership Object Graph;
- **No Reflective Code:** Reflection and dynamic code loading may violate the above invariants by introducing unknown objects and edges, and possibly violating the guarantees of the underlying ownership system;
- **Flow Analysis:** Objects marked as `shared` and `unique` are not currently shown in the Ownership Object Graph. Objects that are `shared` would be trivial to add but would add many uninteresting edges to the Ownership Object Graph. Objects that are `unique` would require a flow analysis to be handled properly (See Section 3.5). Usage edges (e.g., method invocations, field accesses) could be generated for a system with only ownership, but a flow analysis is required for usage edges to be sound in the presence of `lent` objects.

Despite the assumptions about the whole program source being available and restrictions on reflection and dynamic loading, our system is still *relatively sound* in the presence of these features. In particular, as long as the reflective operations are annotated correctly and consistently with ownership information, then any object referred to by some field in the source code that is available will show up in the Ownership Object Graph, as specified above.

For edge soundness, all field references in external library code must be annotated. Since it is often not possible to annotate all such code, “virtual” [26] or “ghost” [13] fields may be declared as annotations in external files. A *virtual field* holds information that is closely related to the meaning of an object, but need not be kept directly in the object in a particular implementation [26]. These annotations do not affect the execution of the system at runtime but are treated as an object’s actual fields by the analysis.

3. Analysis

At a high-level, the analysis works as follows: (1) Obtain an abstract graph from ownership domain annotations; (2) Collapse the inheritance hierarchy by copying fields into subclasses; (3) Instantiate abstractly the types in the abstract graph into objects in the

visual graph, merging objects in the same domain by compatible types (two types are compatible if they have a non-trivial least upper bound); (4) Pull objects in the visual graph from formal domains to actual domains, again merging as necessary; (5) Add details to the visual graph, such as field references, domain links, etc.; and (6) Extract the display Ownership Object Graph as a depth-limited projection of the visual graph.

3.1 Data Representations

The analysis first creates from the program text an AbstractGraph and then converts it into a VisualGraph. The data type declarations of the AbstractGraph and VisualGraph are in Figure 6, and will be referred to by the metavariables shown in parentheses. To help keep the representations distinct, we use English letters (o, d, \dots) for elements of the AbstractGraph, and Greek letters (θ, δ, \dots) for elements of the VisualGraph.

The AbstractGraph consists of the AbstractTypes in the program, the AbstractDomains declared in each type, and the AbstractObjects declared in each domain. An AbstractType also lists AbstractEdges and AbstractLinks. The VisualGraph instantiates the types in the AbstractGraph and shows VisualObjects and VisualDomains: each VisualObject contains VisualDomains and each VisualDomain contains VisualObjects. The VisualGraph also has VisualEdges and VisualLinks.

The identifiers used for the elements in the AbstractGraph and VisualGraph do not correspond to the declared names of domains or objects (e.g., field or variable names) since these cannot be assumed to be globally unique, and do not take into account binding and scope. An implementation would typically have additional fields to hold the user-friendly display name. In addition, an AbstractType maintains its underlying TypeBinding to determine its sub-typing relationship with respect to other AbstractTypes.

The analysis maintains a one-to-one mapping between a VisualDomain δ and its corresponding AbstractDomain d to avoid extra copying. However, a VisualObject typically merges several AbstractObjects as discussed earlier.

3.2 Extract an AbstractGraph from Annotated Code

An AbstractGraph is obtained from the annotated program text using a visitor on the Abstract Syntax Tree of the annotated program. Most steps in Figure 7 are straightforward and are not shown in great detail. During the construction of the AbstractGraph, private ownership domains are given a *protected semantics*¹. The default domain `owned` is considered to be declared at the first point of use and inherited thereafter. If `owned` were to be declared in `java.lang.Object`, all the objects declared in the `owned` domain would be in the same inherited domain and would get unnecessarily merged if they have the same declared type. Singleton `shared`, `lent` and `unique` AbstractDomains are created.

To simplify the treatment of inheritance when creating the VisualGraph, the AbstractGraph is post-processed by collapsing the type hierarchy, i.e., pushing field references declared in the AbstractType corresponding to a given type t into each AbstractType of the sub-types of t .

While the algorithm described in Figure 7 is presented in terms of the ownership domains type system, it can be easily applied to other ownership type systems that do not have the concept of multiple ownership domains per object and assume a single domain or “context” per object [8]. In those cases, we consider that each class implicitly declares a single ownership domain `owned` and proceed according to the algorithm. The other details of the transformation and visualization are unchanged.

¹Domains declared in a class are inherited by its subclasses [3, *Aux-Domains rule* (Fig.14)], but are called somewhat confusingly `private`.

- AbstractGraph (g)
 - Root : AbstractObject */* the root */*
 - Types: List<AbstractType>
- AbstractType (t)
 - TypeBinding: TypeBinding */* Java type */*
 - Domains: List<AbstractDomain>
 - Links: List<AbstractLink>
 - Edges: List<AbstractEdge>
- AbstractDomain (d)
 - DomainType: public | private | parameter
 - Objects: List<AbstractObject>
 - DeclaringType: AbstractType
- AbstractObject (o)
 - Type: AbstractType */* declared type */*
 - Domain: AbstractDomain */* my owner */*
 - Bindings: List<Binding>
 - Visualized: boolean */* bookkeeping */*
- Binding (b)
 - Formal: AbstractDomain
 - Actual: AbstractDomain
- AbstractEdge (e)
 - From: AbstractType */* edge source */*
 - To: AbstractObject */* edge target */*
 - EdgeType: creation | usage | reference
- AbstractLink (s)
 - From: AbstractDomain */* link source */*
 - To: AbstractDomain */* link target */*
- VisualGraph (γ)
 - Root: VisualObject
 - Objects: List<VisualObject>
 - Edges: List<VisualEdge>
 - Links: List<VisualLink>
- VisualObject (θ)
 - Domains: List<VisualDomain>
 - Merged: List<AbstractObject> */* abstract objects merged into 'this' */*
 - Pulled: List<VisualObject> */* visual objects 'this' was pulled into */*
 - IsPulled: boolean */* bookkeeping */*
 - Parent: VisualDomain */* my owner */*
- VisualDomain (δ)
 - Objects: List<VisualObject> */* objects in this domain */*
 - Parents: List<VisualObject> */* objects this domain is part of */*
 - AbstractDomain: AbstractDomain */* map */*
- VisualEdge (η)
 - From: VisualObject */* edge source */*
 - To: VisualObject */* edge destination */*
 - EdgeType: creation | usage | reference
- VisualLink (σ)
 - From: VisualDomain */* link source */*
 - To: VisualDomain */* link destination */*

Figure 6. Data types used by AbstractGraph and VisualGraph. Some fields are for bookkeeping only.

3.3 Convert an AbstractGraph to a VisualGraph

Constructing the VisualGraph from an AbstractGraph takes into account the properties described earlier. The pseudo-code for the algorithm is presented in Figures 8, 9 and 10. The notation

for (T anObject : setOfObjects) ...

is similar to the Java 1.5 “enhanced for-loop” for iterating over collections and arrays. An overbar represents a sequence.

The transformation takes as input the AbstractGraph g whose root is the top-level AbstractObject o_{root} , and AbstractDomain d_{root} is the domain for o_{root} . The top-level procedure VISUALIZEGRAPH (Figure 8) first creates a top-level VisualDomain δ_{root} and then visualizes the AbstractObject o_{root} .

1. For each type declaration C in the program
 - (a) Create `AbstractType` t and add it to $g.Types$
 - (b) For each formal domain parameter in C
 - i. Create corresponding `AbstractDomain` d
 - ii. Add d to $t.Domains$
 - (c) For each declared ownership domain
 - i. Create corresponding `AbstractDomain` d
 - ii. Add d to $t.Domains$
 - (d) For each domain link between d_1 and d_2 in C
 - i. Create `AbstractLink` between the `AbstractDomain` of d_1 and the `AbstractDomain` of d_2
 - ii. Add `AbstractLink` to $t.Links$
 - (e) For each declaration $d C' \langle \bar{a} \rangle o$ in C
 - i. If C' has no `AbstractType`, create t' for C'
 - ii. If `AbstractType` t of Type C has no `AbstractDomain` d , create d and add d to $t.Domains$
 - iii. Create `AbstractObject` o and add to $d.Objects$
 - iv. Create bindings \bar{b} from formals \bar{f} of `AbstractType` t' to actuals \bar{a} of t and add to $o.Bindings$
 - v. If declaration is a field declaration
 - A. Create `AbstractEdge` e of type reference from `AbstractType` t to `AbstractObject` o
 - B. Add e to $t.Edges$
2. Collapse inheritance hierarchy
 - (a) Copy any public domains defined on an interface to the classes implementing the interface
 - (b) Push field references from each super-class into its sub-classes

Figure 7. Obtaining the AbstractGraph.

The conversion involves two mutually recursive functions, `VISUALIZEOBJECT` to convert an `AbstractObject` into a `VisualObject` and `VISUALIZEDOMAIN` to convert an `AbstractDomain` into a `VisualDomain`. Each `AbstractDomain` declared in the `AbstractType` of an `AbstractObject` is visualized in turn.

Before a `VisualObject` θ is created for an `AbstractObject` o of type t inside a `VisualDomain` δ , the analysis calls `FINDOBJECT` to look for an existing `VisualObject` in δ with which o can be merged, i.e., if δ has a θ of type t' where t and t' have *non-trivial least upper bounds* using procedure `GETLEASTUPPERBOUNDS`. If such an object does not exist, a new `VisualObject` is created. If θ exists, then it is used and o is added to the list of `AbstractObjects` that are merged by θ . Each call to `FINDOBJECT` takes into account the `AbstractTypes` of all the `AbstractObjects` that are merged into a `VisualObject`.

Procedure `ARENONTRIVIALTYPES` excludes from the computed types any type mentioned in the list of trivial types. By default, the list includes `java.lang.Object`, `java.io.Serializable` and other user-selected types. However, a trivial type is allowed to be part of the least upper bounds, if the `AbstractObject` is declared of that type.

Once `VisualObjects` and `VisualDomains` have been created, procedure `PULLOBJECTS` uses a worklist to pull existing `VisualObjects`: each `VisualObject` is pulled from a formal to an actual domain, potentially creating a new `VisualObject` if it cannot be merged with an existing one. If a new `AbstractObject` is merged into an existing `VisualObject`, the `VisualObject` is added back to the worklist. New `VisualObjects` are also added to the worklist so they get pulled in turn. The analysis tracks the `VisualObjects` that a given `VisualObject` is pulled into.

Finally, the top-level procedure `VISUALIZEGRAPH` calls `VISUALIZEFIELDREFS` to add field references to the `VisualGraph` and `VISUALIZEDOMAINLINKS` to add the domain links.

When adding the field references associated with a `VisualObject` θ , `ADDFIELDREFS` (Figure 10) takes into account all the field references declared in the `AbstractType` of each `AbstractObject` merged into a `VisualObject`. `ADDFIELDREFS` also adds field references to all the pulled `VisualObjects` that are tracked by the book-keeping fields.

The algorithm given in Figure 8 is sound for systems that use single inheritance and have no declared variables of a trivial type. In systems that do not meet these restrictions, the algorithm may produce multiple visual objects to represent the same runtime object. In this case, two possible approaches can be used to restore soundness. The first approach is the instantiation-based view described in Section 2 above, whereby visual objects are created for each object that is instantiated rather than for each field or variable declaration in the program.

In the second approach, the procedure `FINDOBJECT` in Figure 8 is modified to identify all `VisualObjects` that could be merged with the target `TypeBindings`. If there is more than one such `VisualObject`, the analysis unifies the `VisualObjects` and the resulting `VisualObject` has the union of the `VisualDomains`, merged `AbstractObjects`, etc. The analysis then unifies recursively all the `VisualObjects` that a unified `VisualObject` was pulled into. The `FINDOBJECT` procedure then returns the unified `VisualObject`.

3.4 Convert the VisualGraph into the Ownership Object Graph

The ownership object graph that is displayed is a depth-restricted projection of the visual graph, starting from a root object. The visualization currently uses the nested boxes discussed earlier but the algorithm is not tied to a specific graphical notation.

This step is depends on the visualization package used. In our prototype implementation, we use `GraphViz` [16]. Each dark grey box for each object and white-filled node for each domain must have a unique identifier — otherwise, nodes with the same identifier get unified. Since there is one `VisualDomain` corresponding to an `AbstractDomain`, and an `AbstractDomain` is shared across all the `AbstractObject` instances of a given `AbstractType`, each occurrence of a `VisualDomain` that appears in a `VisualObject` must be assigned a new identifier.

Because the Ownership Object Graph is a depth-limited projection, it may omit objects deeply nested in the ownership hierarchy. These objects are conceptually summarized by their containing object, and the visualization remains sound with this summarization. However, those objects may have field references to objects that are present in the projection; for soundness, the corresponding edges should be shown. In our approach, these field reference edges can be represented by summary fields in the leaf objects of the graph.

These summary fields are identified as follows. For each leaf object θ_{leaf} in the Ownership Object Graph, for each transitive child object θ_{child} of θ_{leaf} , in an *extended depth-limited projection* of the `VisualGraph`, we consider all actual field references from `VisualObject` θ_{child} to `VisualObject` θ_{target} , where θ_{target} is not a child of θ_{leaf} . Each such edge is represented by a summary edge from θ_{leaf} to θ_{parent} , where θ_{parent} is the nearest parent of θ_{target} that is visible in the Ownership Object Graph. This algorithm will find summary fields for all fields present at runtime as long as the *extended depth-limited projection* projects below the leaves of the graph until a cycle in the `VisualGraph` is reached — i.e., for each path downward from a leaf, the same `VisualObject` is reached a second time. This projection must still be depth-limited, as in general the `VisualGraph` may have an infinite depth due to reference cycles.

3.5 Limitations and Future Work

In future work, we plan on improving the precision of the analysis, proving the soundness of the Ownership Object Graph, and evaluating the scalability of the approach on large systems.

Precision. Merging objects of the same type that are in the same domain can lead to unwanted merging in some cases. Adding generic types improves the precision of the analysis, but for additional precision, an alias analysis may be needed [29].

```

Global: Map<AbstractDomain ,VisualDomain > map
Global: AbstractGraph g (input)
Global: VisualGraph  $\gamma$  (output)

VISUALIZEGRAPH()
 $\delta_{root} = \text{new VisualDomain} ()$ 
 $\delta_{root}.AbstractDomain = d_{root}$ 
 $\gamma = \text{new VisualGraph} ()$ 
 $\gamma.Root = \text{VISUALIZEOBJECT}(\delta_{root}, o_{root})$ 
PULLOBJECTS()
VISUALIZEFIELDREFS()
VISUALIZEDOMAINLINKS()

VISUALIZEOBJECT(VisualDomain  $\delta$ , AbstractObject  $o$ )
 $\bar{t} = \text{GETTYPEBINDINGS}(o.Type)$ 
 $\theta = \text{FINDOBJECT}(\delta, \bar{t})$ 
if ( $\theta == \text{NULL}$ )
  then  $\theta = \text{new VisualObject} ()$ 
   $\delta.Objects.add(\theta)$ 
   $\theta.Parent = \delta$ 
   $\gamma.Objects.add(\theta)$ 
 $\theta.Merged.add(o)$ 
 $o.Visualized = \text{TRUE}$ 
for ( $d_i : t.Domains$ )
  do  $\delta_i = \text{VISUALIZEDOMAIN}(\theta, d_i)$ 
   $\delta_i.Parents.add(\theta)$ 
   $\theta.Domains.add(\delta_i)$ 
return  $\theta$ 

VISUALIZEDOMAIN(AbstractDomain  $d$ )
 $\delta = \text{map.get}(d)$ 
if ( $\delta == \text{NULL}$ )
  then  $\delta = \text{new VisualDomain} ()$ 
   $\text{map.put}(d, \delta)$ 
   $\delta.AbstractDomain = d$ 
  for ( $o_i : d.Objects$ )
    do if ( $o_i.Visualized$ )
      then continue
   $\text{VISUALIZEOBJECT}(\delta, o_i)$ 
return  $\delta$ 

FINDOBJECT(VisualDomain  $\delta$ , List<TypeBinding>  $\bar{t}$ )
for ( $\theta_i : \delta.Objects$ )
  do  $\bar{t}_m = \text{GETMERGEDTYPES}(\theta_i)$ 
   $\bar{\ell} = \text{GETLEASTUPPERBOUNDS}(\bar{t}_m, \bar{t})$ 
  if ( $\text{ARENONTRIVIALTYPES}(\bar{\ell}, \bar{t})$ )
    then return  $\theta_i$ 
return NULL

GETTYPEBINDINGS(AbstractType  $t$ )
   $\triangleright$  Obtain list of transitive supertypes

GETLEASTUPPERBOUNDS(List  $\bar{\ell}$ , List  $\bar{t}$ )
   $\triangleright$  Compute least-upper-bounds if they exist

ARENONTRIVIALTYPES(List  $\bar{\ell}$ , List  $\bar{t}$ )
   $\triangleright$  Exclude from  $\bar{\ell}$  trivial types such as java.lang.Object
   $\triangleright$  or in the user-specified list of trivial types
   $\triangleright$  EXCEPT if it is one of the declared types in  $\bar{t}$ 
  return TRUE if remaining list of types non-empty

GETMERGEDTYPES(VisualObject  $\theta$ )
List  $l = \text{new List}()$ 
for ( $o_i : \theta.Merged$ )
  do  $l.add(o_i.Type)$ 
return  $l$ 

```

Figure 8. Pseudo-code for creating VisualGraph.

```

PULLOBJECTS()
Stack  $worklist = \text{new Stack}()$ 
for ( $\theta : \gamma.Objects$ )
  do  $worklist.push(\theta)$ 
while ( $!worklist.isEmpty()$ )
  do VisualObject  $\theta = worklist.pop()$ 
  PULLOBJECT( $\theta, worklist$ )

PULLOBJECT(VisualObject  $\theta$ , Stack  $worklist$ )
   $\triangleright$  List.add first checks if element exists to avoid duplicates
   $\triangleright$  and returns TRUE if element is added, FALSE otherwise.
   $\triangleright b_1 | = b_2$  is shorthand for  $b_1 = b_1 \text{ OR } b_2$ 
   $\delta_f = \theta.Parent$ 
   $d_f = \delta_f.AbstractDomain$ 
  for ( $d_a : \text{GETACTUALS}(d_f)$ )
    do if ( $d_a == d_f$ )
      then continue
       $\delta_a = \text{map.get}(d_a)$ 
       $\bar{t}_m = \text{GETMERGEDTYPES}(\theta)$ 
       $\theta_p = \text{FINDOBJECT}(\delta_a, \bar{t}_m)$ 
       $changed = \text{FALSE}$ 
      if ( $\theta_p == \text{NULL}$ )
        then  $\theta_p = \text{new VisualObject} ()$ 
         $\gamma.Objects.add(\theta_p)$ 
         $\theta_p.Parent = \delta_a$ 
         $\theta_p.IsPulled = \text{TRUE}$ 
         $\delta_a.Objects.add(\theta_p)$ 
         $changed = \text{TRUE}$ 
       $\theta.Pulled.add(\theta_p)$ 
      for ( $o : \theta.Merged$ )
        do  $changed | = \theta_p.Merged.add(o)$ 
       $\triangleright$  Add domains from merged object
      for ( $\delta_i : \theta.Domains$ )
        do  $changed | = \theta_p.Domains.add(\delta_i)$ 
         $\delta_i.Parents.add(\theta_p)$ 
       $\triangleright$  If anything changed, add back to  $worklist$ 
       $\triangleright$  so that merged objects get pulled too...
      if ( $changed$ )
        then  $worklist.push(\theta_p)$ 

GETACTUALS(AbstractDomain  $d_f$ )
List  $l = \text{new List}()$ 
 $\delta_f = \text{map.get}(d_f)$ 
for ( $\theta_i : \delta_f.Parents$ )  $\triangleright$  Pull “up” only
  do for ( $o_i : \theta_i.Merged$ )
    do for ( $b_i : o_i.Bindings$ )
      do if ( $b_i.Formal == d_f$ )
        then  $l.add(b_i.Actual)$ 
return  $l$ 

```

Figure 9. Pseudo-code for creating VisualGraph (continued).

An object marked `unique` is not shown until it is assigned to a specific domain. Thus, an inter-procedural flow analysis is needed to track an object from its creation (at which point it is `unique`) until its assignment to a specific domain. In the current tool, this flow analysis is not implemented, so a `unique` object returned from a factory method must be annotated with the domain in which it should be displayed. In addition, the flow analysis can determine what domain a `lent` object is really in. A precise handling of the `lent` annotation is needed to add to the Ownership Object Graph usage edges corresponding to method invocations and field accesses since many method parameters are annotated with `lent`. Those edges are currently missing.

Scalability. Finally, we lack empirical evidence of the scalability of the approach to large systems. In the absence of semi- or fully-automated annotation inference (a separate research problem), the main difficulty would be adding the ownership domain annotations to legacy code.

```

VISUALIZEFIELDREFS()
  for (  $\theta : \gamma.Objects$  )
    do ADDFIELDREFS(  $\theta$  )

ADDFIELDREFS(VisualObject  $\theta_{src}$ )
  for (  $o : \theta_{src}.Merged$  )
    do for (  $e : o.Type.Edges$  )
      do for (  $d_a : GETBINDINGS(o, e.To.Domain)$  )
        do  $\delta_a = map.get(d_a)$ 
           $\theta_{dst} = GETMERGED(\delta_a, e.To)$ 
          if (  $\theta_{dst} \neq NULL$  )
            then ADDFIELDREFS( $\theta_{src}, \theta_{dst}$ )

ADDFIELDREFS(VisualObject  $\theta_{src}, VisualObject \theta_{dst}$ )
   $\eta = new VisualEdge()$ 
   $\eta.From = \theta_{src}$ 
   $\eta.To = \theta_{dst}$ 
  if (  $\gamma.Edges.add(\eta)$  )
    then for (  $\theta_{src_p} : \theta_{src}.Pulled$  )
      do for (  $\theta_{dst_p} : \theta_{dst}.Pulled$  )
        do ADDFIELDREFS( $\theta_{src_p}, \theta_{dst_p}$ )

GETBINDINGS(AbstractObject  $o, AbstractDomain d$ )
  List  $l = new List()$ 
  for (  $b : o.Bindings$  )
    do if (  $b.Formal == d$  )
      then  $l.add(b.Actual)$ 
  return  $l$ 

GETMERGED(VisualDomain  $\delta, AbstractObject o$ )
  for (  $\theta_i : \delta.Objects$  )
    do for (  $o_m : \theta_i.Merged$  )
      do if (  $o_m == o$  )
        then return  $\theta_i$ 
  return NULL

VISUALIZEDOMAINLINKS()
  for (  $t : g.Types$  )
    do for (  $s : t.Links$  )
      do VisualLink  $\sigma = new VisualLink()$ 
         $\sigma.From = map.get(s.From)$ 
         $\sigma.To = map.get(s.To)$ 
         $\gamma.Links.add(\sigma)$ 

```

Figure 10. Pseudo-code for creating VisualGraph (continued).

4. Evaluation

To evaluate our approach, we built tools and conducted two case studies on real object-oriented implementations.

4.1 Ownership Object Graph Tool

The tool obtains the Ownership Object Graph of an annotated program, represents it as a GraphViz clustered graph [16] and offers the following features:

- **Top-Level Objects:** the displayed Ownership Object Graph is a depth-limited projection of the visual graph — the depth is user-selectable but cannot be too large. The user can interactively select an object as the root of the graph to view its substructure;
- **Trivial Types:** the tool allows the user to specify an optional list of trivial types;
- **Design Intent Types:** the tool allows the user to specify an optional list of design intent types for labelling objects;
- **Object Labels:** objects can be labelled with an optional field name or variable name and an optional type name. The type used in the label consists of a least-upper-bound type or a design intent type as discussed earlier;

- **Elide Private Domains:** the tool allows the user to elide all the private domains at once and show only the public domains in the visible Ownership Object Graph;
- **User Elision:** the tool can elide temporarily uninteresting elements. When the sub-structure of an object is elided, the symbol (+) is appended to its label;
- **Traceability:** the tool can show for a given visual object, the list of abstract objects and their abstract types merged into it, to help the user fine-tune the list of trivial types;
- **Navigation:** the tool supports zooming, searching by AbstractObject or AbstractType name, etc.

4.2 Case Study: JHotDraw

The subject system for the first case study is JHotDraw [20]. Version 5.3 has around 200 classes and around 15,000 lines of Java. The core types in JHotDraw were organized according to the Model-View-Controller pattern as follows:

- **Model:** consists of Drawing, Figure, etc. A Drawing is composed of Figures which know their containing Drawing. A Figure has a list of Handles to allow user interactions;
- **View:** consists of DrawingEditor, DrawingView, etc.;
- **Controller:** includes Handle, Tool and Command. A Tool is used by a DrawingView to manipulate a Drawing. A Command encapsulates an action to be executed.

Annotation Process. JHotDraw was annotated without making any structural refactoring such as extracting interfaces, etc. Since JHotDraw Version 5.3 did not use generic types and to improve the precision of the analysis, we used Eclipse refactorings [14] to infer the most specific generic types of containers such as Vector — and prevent objects of type Vector<Handle> and those of type Vector<Figure> from getting merged. The annotation process is described in detail elsewhere [1].

Ownership Object Graph. We made use of the visualization during the annotation process: for instance, visualizing the annotations encouraged us to make more use of the owned annotation since owned pushes objects down in the ownership hierarchy and avoids cluttering the top-level domains.

The list of trivial types includes interfaces implemented by many classes, e.g., Storable, Animatable, constant interfaces, e.g., SwingConstants², as well as interfaces implementing the Observer design pattern, e.g., ViewChangeListener. Both Tool and Command implement ViewChangeListener and are in the Controller domain, so they may get merged otherwise³.

Evaluation. Existing compile-time analyses [40, 19] cannot produce, for a program the size of JHotDraw, a readable flat object graph that fits on one page (See other visualizations [2]). The top-level Ownership Object Graph obtained from the annotated program using our approach is shown in Figure 11 and clearly illustrates the Model-View-Controller design.

Each gray box corresponds to a “canonical object” that represents many instances at runtime and is labeled with one or more “design intent” type from the core framework package (variable names were not particularly informative and are not shown).

In the visualization, the Controller domain clearly shows Command, Handle and Tool instances. The self-edge on Tool is explained by the fact that an UndoableTool wraps a Tool and similarly, an UndoableCommand wraps a Command. The View domain shows instances of DrawingEditor (the application itself) and DrawingView. The Model domain shows instances of Figure:

²Inheriting from a constant interface to access the constants without qualifying them is a bad coding practice, the Constant Interface *antipattern* [6, Item #17] and Java 1.5 supports *static imports* to avoid it.

³The tool currently scans field and variable declarations and not object allocations as discussed in Section 2.

a `Figure` has one or more `Connectors` that define how to locate a “connection point”.

Understanding why `Drawing` did not appear in the `Model` tier led us to discover that `StandardDrawing`, the base class implementing the `Drawing` interface, extends `CompositeFigure`, thus a `Drawing is-a Figure`⁴. Although this is not a design problem *per se*, it is inconsistent with the design intent in the core framework package: there, interface `Drawing` does not extend interface `Figure`. This finding was unexpected in a framework as carefully designed and as widely studied as `JHotDraw`. Although a class diagram could reveal that a `StandardDrawing` is a `Figure`, the Ownership Object Graph quickly pinpoints that.

The top-level domains have only 28 objects even though `JHotDraw` has 200 around types and presumably each type is instantiated at least once. This illustrates how the properties of the Ownership Object Graph provide more abstraction and more design intent than a visualization of the raw object graph [19, 40].

In fact, designers often employ similar techniques in a design-oriented class diagram, i.e., one not retrieved from an implementation using a tool: a) *merge interface and abstract implementation class* — although important for code reuse, such a code factoring is often unimportant from a design standpoint; and b) *subsume a set of similar classes under a smaller set of representative classes* — showing many similar subclasses that vary only in minor aspects on a class diagram often leads to needless clutter [36, pp. 139–140]. It seems the `JHotDraw` designers used similar techniques to present the `JHotDraw` design in their tutorials [36].

In the Ownership Object Graph, all runtime figure objects referenced in the program by the `Figure` interface, its abstract implementation class `AbstractFigure`, or any of its concrete subclasses `DecoratorFigure`, `ConnectionFigure`, etc., appear as a single `Figure` object in the `Model` domain.

The distinction between public and private domains within each object enables eliding all the private domains at once to show only the top-level `Model`, `View` and `Controller` domains in object `Main`. To illustrate the hierarchy however, objects were selected individually and their internals were elided — those have the symbol (+) appended to their labels. `DrawingEditor` shows its internals: its private owned domain has an `Iconkit` object among others, and `IconKit` has its own substructure, but the latter is elided.

Currently, the visualization does not show multiplicities: at runtime, there is one `DrawingEditor` (the application itself), one `IconKit`, but one or more `DrawingView` objects.

4.3 Case Study: 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.

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*.

Annotation Process. `HillClimber` was organized into a data ownership domain to store the `graph`, a `ui` domain to hold the user interface elements, and a `logic` domain to hold the engine, search objects, and associated objects. Unlike `JHotDraw`, adding annotations to `HillClimber` involved refactoring to decouple the code. Again, to increase the precision of the analysis, we 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>>`. The annotation process is described in detail elsewhere [1].

Evaluation. The Ownership Object Graph in Figure 12 shows clearly the core `HillClimber` top-level objects, `window`, `canvas`, `engine` and `graph`. Similarly, the `Search` object in the `LogicTier` domain merges many instances of sub-classes of class `Search` such as `MCHSearch`, `RandSearch`, etc.

The `Graph` base class declares a `nodes:Vector<Node>` field and its subclass `HillGraph` refers to that same object. Generic types improved the precision of the analysis and prevented the merging of `edges:Vector<Edge>` and `nodes:Vector<Node>`. The `graph:Graph` object merges both `Graph` and `HillGraph` and shows objects `nodes` and `edges` in its owned domain.

Since a domain is introduced where it is declared and then is inherited according to the protected semantics, `HillGraph` and `Graph` share the same owned domain. However, when two “unrelated” objects, e.g., a `Button` object and a `Panel` object get merged (since they have a non-trivial least upper bound) and each has its declared owned domain, it is possible to have multiple domains of the same name in a given visual object — in that case, a domain name is fully qualified with the type name where it was declared in the abstract graph.

The visualization highlights the need to potentially make object `edgesIn`, the incident edges on a node, encapsulated inside object `node:Entity`. This would require changing the annotations and the code as necessary to abide by the rules of the type system. This in turn would push the object down the ownership tree and remove it from the top-level domain.

The mediator: `ICanvasMediator` object was introduced during a refactoring to decouple the code [1] and mediate between the graph and the canvas. Finally, the object labeled `window:Frame` merges several user interface objects representing dialogs, etc., thus illustrating the type abstraction property.

5. Related Work

Program Visualization. There is a large body of software visualization research where the emphasis is on novel kinds of visualization using colors, shapes, 3D, etc. Our contribution in this paper is not the visualization *per se* — we’re using the simple but effective `GraphViz` package — it is in having developer-specified ownership annotations drive a sound compile-time visualization of the program’s execution structure.

Many dynamic analyses visualize the execution structure but ignore ownership: they instrument the running program, filter the program traces based on various query criteria and then visualize the summarized information in novel ways, often with a granularity not larger than an object or a class [23, 37, 35, 17, 39, 30, 10]. On the other hand, such analyses handle programs for which source code is not available, do not require source code annotations or changes to the source code to add the annotations and allow more fine-grained user interaction in producing the visualization.

Ownership Annotation Inference. Annotation inference is an active area of research using both static [4, 9, 24, 25] and dynamic [41] analyses. However, a fully automated inference cannot create multiple public domains in one object and meaningful domain parameters to represent the design intent, such as the separate `Model`, `View`, and `Controller` in the `JHotDraw` case study. Existing inference algorithms produce for each class a long list of domain parameters, often place each field in a separate domain, or annotate many objects with `shared` or `lent` [4].

Dynamic Object Graph Analyses. Dynamic analyses can infer the ownership structure of a running program based on its heap structure. Although these techniques have the advantage of not requiring abundant source code annotations, they can only infer the equivalent of `owned`, `shared`, `lent` and `unique` annotations. This

⁴ According to the Release Notes for `JHotDraw` Version 5.1, this change was made to support inserting a `Drawing` as a `Figure` inside another `Drawing`.

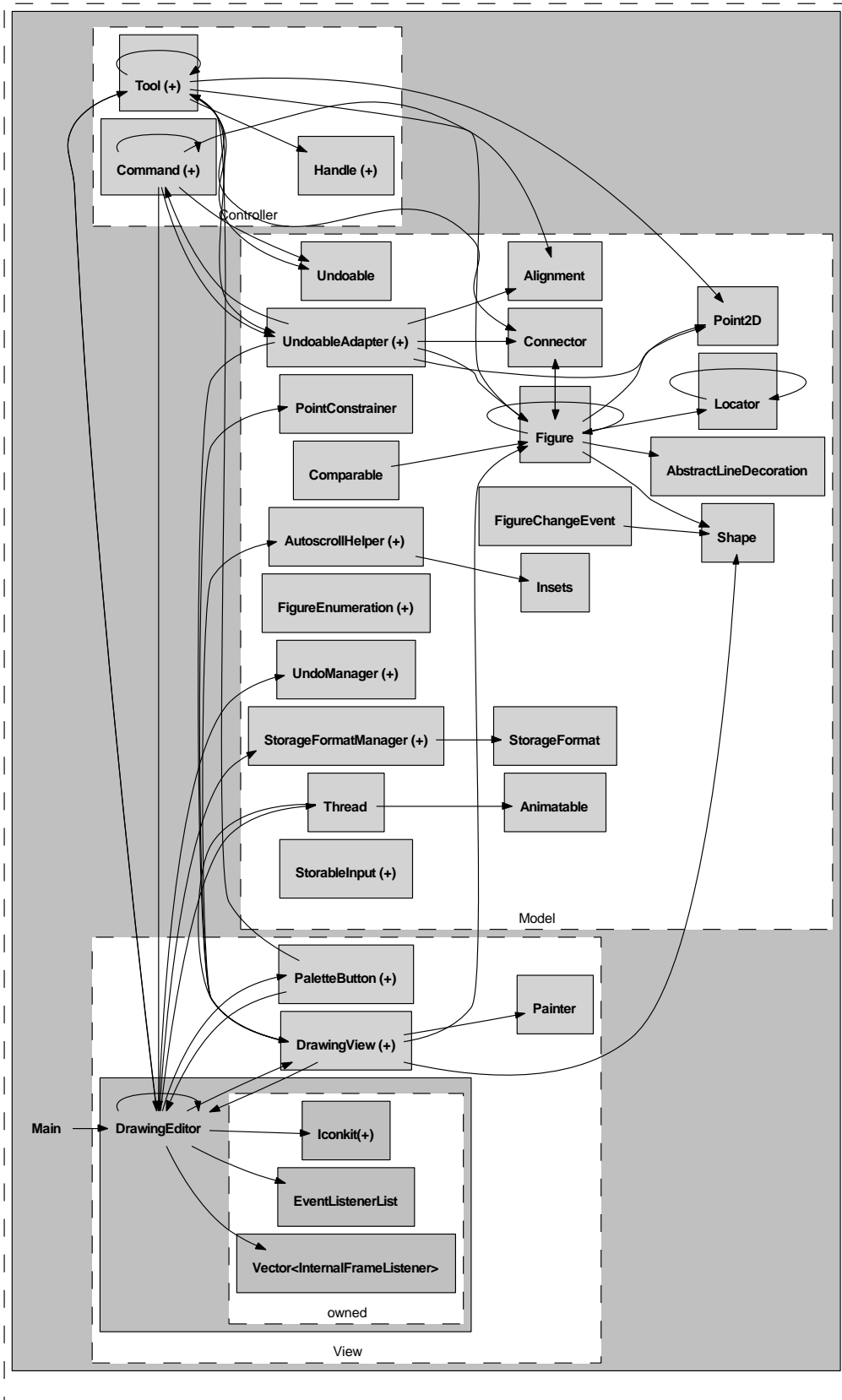


Figure 11. Top-level Ownership Object Graph for JHotDraw. This graph was laid out automatically by GraphViz without user intervention. The edges correspond to field references.

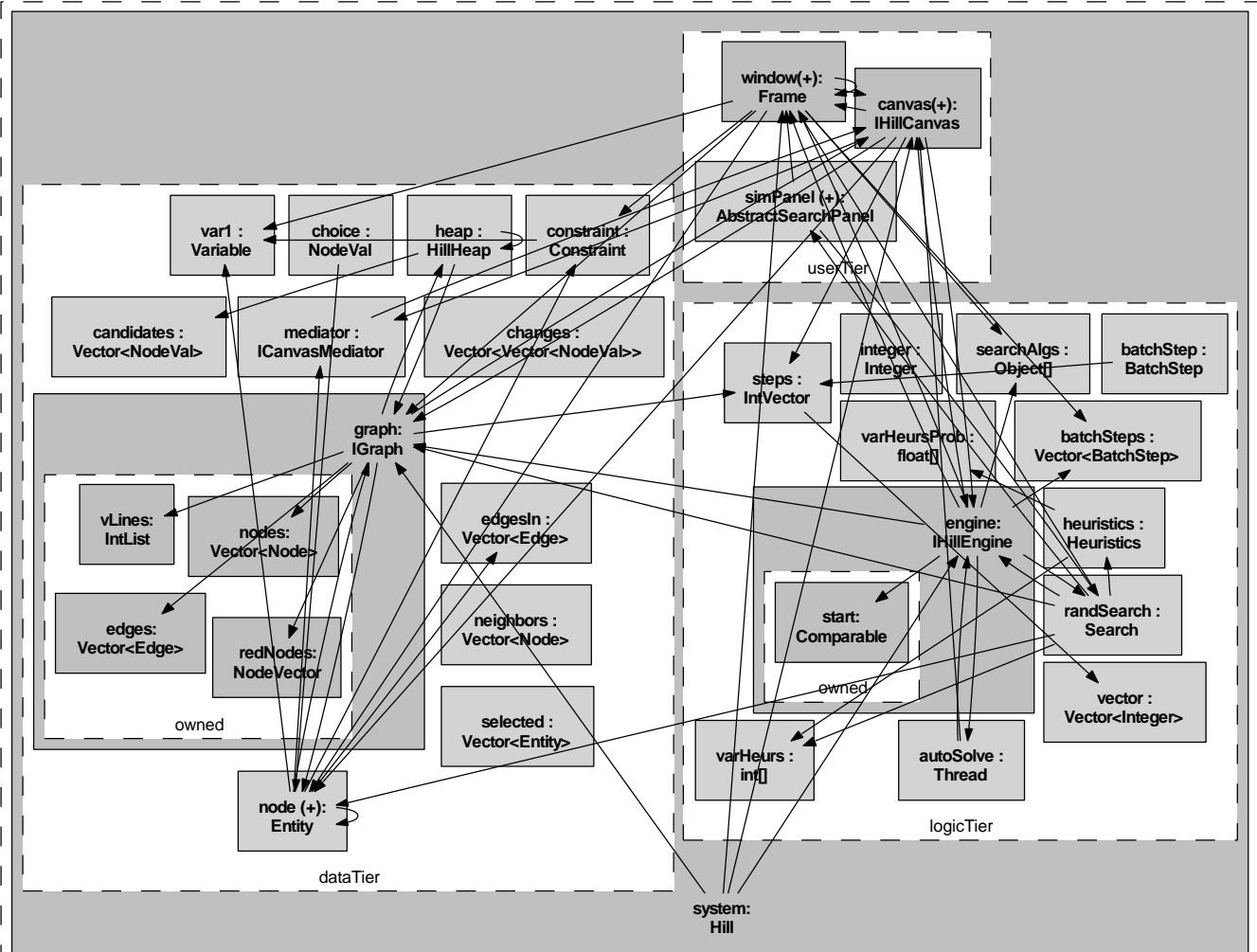


Figure 12. Ownership Object Graph for HillClimber, laid out automatically by GraphViz dot without user intervention.

assumes a strict owner-as-dominator hierarchy which is not flexible enough to represent design patterns such as the Composite pattern.

Rayside et al. [33] characterize sharing and ownership and produce a matrix display of the ownership structure. Similarly, Mitchell [27] uses lightweight ownership inference to examine a single heap snapshot rather than the entire program execution, and scales the approach to large programs through extensive graph transformation and summarization. Flanagan and Freund [12] proposed a dynamic analysis to reconstruct each intermediate heap from a log of object allocations and field writes, then apply a sequence of abstraction-based operations to each heap, and combine the results into a single object model that conservatively approximates all observed heaps from the programs execution. Their tool, AARD-VARK, has the notion of ownership and containment and uses simple heuristics to choose the most appropriate generalization. Noble et al. [18, 28] and Potanin et al. [31] also process heap snapshots and show both matrix and graph visualizations of ownership trees, indicating an object's “aliasing shadow” and “interior”.

There are several problems with dynamic analyses: first, runtime heap information does not convey design intent. Second, a dynamic analysis may not be repeatable, i.e., changing the inputs or executing different use cases might produce different results. Compared to dynamic ownership analyses — which are descriptive and

show the ownership structure in a single run of a program, the Ownership Object Graph obtained at compile time is prescriptive and shows ownership relations that will be invariant over all program runs. Third, a dynamic analysis cannot be used on an incomplete program still under development or to analyze a framework separately from a specific instantiation. Finally, some dynamic analyses carry a significant runtime overhead — a 10x-50x slowdown in one case [12], which must be incurred each time the analysis is run, whereas the main cost of adding annotations is incurred once.

Static Object Graph Analyses. Several static analyses produce various object graphs, but they do not use ownership and do not convey design intent. PANGAEA [40] produces a flat object graph. WOMBLE [19] uses syntactic heuristics and hard-coded heuristics for container classes to obtain an object model including multiplicities, but its analysis does not attempt to be sound and the flat object graph it produces does not scale to large programs: in particular, the WOMBLE visualization of the 15,000-line JHotDraw does not fit on one readable page [2] nor does it convey the Model-View-Controller design.

AJAX [29] uses an alias analysis to build a refined object model as a conservative compile-time approximation of the heap graph reachable from a given set of root objects, and simplifies it through a series of transformations. However, AJAX does not use ownership

