Object Graphs with Ownership Domains: an Empirical Study

Radu Vanciu and Marwan Abi-Antoun

Department of Computer Science
Wayne State University, Detroit, MI 48202 USA
{radu, mabiantoun}@wayne.edu,

Abstract. Researchers have proposed many ownership type systems but reported limited experience with most of them on real object-oriented code. Only a few systems have been implemented, and there have been few substantial case studies done with those systems.

In order to better empirically evaluate ownership type systems, we have therefore conducted a number of case studies applying the Ownership Domains type system to programs at a larger scale. To facilitate the study of legacy code, we reimplemented Ownership Domains using available language support for annotations. After annotating and typechecking a range of object-oriented systems, we extracted global, hierarchical, Ownership Object Graphs (OOGs) using static analysis. OOGs provide an abstracted view that is consistent with programmer design intent, compared to flat object graphs that can be extracted without the benefit of the ownership annotations. An OOG also visualizes the system's ownership structure and helps developers refine the annotations they add to better express the system's design.

This paper shares our observations from studying the annotations and the extracted OOGs across several subject systems. We compute metrics on the annotations and on the extracted OOGs, to gain insights into the ownership relationships latent within object-oriented code and to evaluate the effectiveness of the abstraction mechanisms in OOGs.

Keywords: ownership types, object graphs, empirical evaluation

1 Introduction

Researchers have proposed many ownership type systems [39, 21, 36, 17, 9, 23, 18] but have not reported significant experience with most of them on real object-oriented code. Only a few implementations of the type systems have been reported in the literature [11, 9, 23, 43], and there are few substantial case studies evaluating these systems.

To address this gap, we have conducted a number of case studies applying one ownership type system, Ownership Domains [9], to a broad range of larger-scale object-oriented systems. To facilitate the study of legacy code, we re-implemented Ownership Domains using available language support for annotations. We then annotated several systems and typechecked the annotations.

From the annotated systems, we extracted global, hierarchical, Ownership Object Graphs (OOGs) using static analysis [3, 2].

The nodes of the OOG represent abstractions of objects and ownership domains. An *ownership domain* is a conceptual group of objects with an explicit name and explicit policies that govern how it can reference objects in other domains [9]. A domain roughly corresponds to an architectural tier, a notion present in most architecture description languages. The nodes of the OOG form a hierarchy: an object can have child objects. A child object can be strictly owned by or conceptually part of the parent object. Instead of making objects direct children of other objects, ownership domains introduce an extra level of indirection. So, in an OOG, an object has domains, which contain other objects to form its substructure, and so on. Edges of the OOG represent reference relations between objects, where the references respect the explicit policies between domains.

Hierarchy is effective to shrink a large, flat graph [47], and enables an OOG to scale, so the size of the top-level diagram does not increase linearly with the size of the program. Hierarchy also provides abstraction since a hierarchical representation enables both high-level understanding and detail, and is commonly used to document software structure.

An OOG provides *abstraction by ownership hierarchy* when it shows architecturally significant objects near the top of the hierarchy and data structures further down. Furthermore, an OOG supports *abstraction by types*, which acts within a domain to collapse objects that share a common supertype into a single representative object. The OOG visualization helps developers refine the annotations they add to better express the system's design. Since there are multiple ways to express design intent, there are multiple ways to annotate a system. The type checker ensures that the annotations are consistent with each other and with the code. Moreover, the OOG is *sound* in two respects. First, each runtime object has exactly one representative in the OOG. Second, the OOG has edges that correspond to all possible runtime points-to relations between those runtime objects.

This paper shares our observations from studying the annotations and the extracted OOGs across several subject systems. To evaluate the subject systems quantitatively, we compute metrics on the annotations and on the extracted OOGs. To show how annotations can be refined over time, we annotated a few systems in stages, gathering metrics at two different milestones along the way.

Goals for annotation metrics. We computed metrics on the annotations to:
- understand the ownership relationships latent within legacy code;
- measure the quality of the annotations that we added.

Goals for OOG metrics. We computed metrics on the OOG to:
- better understand the ownership-based hierarchical structure of legacy code;
- measure the abstraction level of the extracted OOGs: for example, a large number of objects at the top level of the hierarchy indicates that the OOG is too cluttered and not sufficiently abstract.

Outline. The rest of this paper is organized as follows. Section 2 provides background information on the annotations and the OOGs. Section 3 describes our method. Section 4 presents our results for the annotation metrics and the OOG metrics. Section 5 discusses our observations and mentions some limitations and threats to validity. We discuss related work in Section 6 and conclude.

2 Background

This section provides background information on the annotations and the OOGs.

2.1 Review of Ownership Domains

Ownership Domains. An *ownership domain* is a conceptual group of objects with an explicit name and explicit policies that govern how the domain can reference objects in other domains. Each object is assigned to a single domain, which does not change at runtime. A developer indicates the domain of an object by annotating each reference to that object in the program. The annotations define two kinds of object hierarchy, logical containment and strict encapsulation.

1. **Logical Containment (part-of).** A public domain provides *logical containment*, by making an object conceptually *part of* another object. Having access to an object gives access to objects inside all its public domains.
2. **Strict Encapsulation (owned-by).** A private domain provides *strict encapsulation*, by making an object *owned by* its parent object. Then, a `public` method cannot return an alias to an object in a private domain, even though the Java type system allows returning an alias to a field marked with the visibility modifier `private`.

Why Ownership Domains? There are several key features of Ownership Domains that are crucial for expressing design intent in code and extracting OOGs. The first is having explicit "contexts" or domains. Other ownership type systems implicitly treat all objects with the same owner as belonging to an "ownership domain", but one that is implicit [22]. On the other hand, explicit contexts are useful, because developers can define multiple contexts per object to express their design intent as our metrics will show later (Section 4).

Second, Ownership Domains support pushing any object underneath any other object in the ownership hierarchy. A child object may or may not be encapsulated by its parent object, and can still be referenced from outside its owner, if it is part of a public domain of its parent. In addition, an object can grant particular external domains (named by a domain parameter) permission to access objects within one of its private domains, using a domain link specification. These mechanisms for expressing conceptual containment, rather than strict encapsulation, contrast with an *owners-as-dominators* type system [21], which requires any access to a child object to go through its owning object.

Public domains are ideal in an OOG to achieve hierarchy by capturing interesting part-of relationships among objects. Ownership and OOGs are only useful to the extent that a significant hierarchy exist; if most or all objects

$$
\begin{aligned}
CL &::= \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \texttt{assumes}\ \overline{\gamma} \rightarrow \overline{\delta}\ \{\ \overline{T}\ \overline{f};\ K\ \overline{D}\ \overline{L}\ \overline{M}\ \} \\
K &::= C(\overline{T'}\ \overline{f'}, \overline{T}\ \overline{f})\ \{\ \texttt{super}(\overline{f'});\ \texttt{this}.\overline{f} = \overline{f};\} \\
D &::= [\texttt{public}]\ \texttt{domain}\ d; \\
L &::= \texttt{link}\ d \rightarrow d'; \\
M &::= T_R\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ e;\ \} \\
e &::= x \mid \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(\overline{e}) \mid e.f \mid (T)e \mid e.m(\overline{e}) \mid \ell \mid \ell > e \\
n &::= x \mid v \\
T &::= C\texttt{<}\overline{p}\texttt{>} \\
p &::= \alpha \mid n.d \mid\ \texttt{shared} \\
v, \ell &\ \in\ \textit{locations}
\end{aligned}
$$

Fig. 1: Featherweight Domain Java Syntax [9].

are in top-level domains, ownership has not provided any real benefit. But an owners-as-dominators semantics can result in many top-level objects, as completely dominated objects are rare (our metrics will show this). A rule of thumb in software architecture is that each architectural tier should have only five to seven components [33]. Applying this guideline to ownership suggests that OOGs are more readable if top-level domains only contain five to seven abstract objects.

Language. We briefly review the Ownership Domains abstract syntax (Fig. 1) since we present some annotated examples and define the annotation metrics in terms of the abstract syntax. To facilitate the study of legacy code, we re-implemented Ownership Domains using available language support for annotations. For a detailed discussion of the concrete annotation language, see [2, Appendix A].

We discuss the Ownership Domains syntax using an example (Fig. 2). We show the same example using the language extensions (Fig. 2a) and Java annotations (Fig. 2b). The example illustrates how an application would read the contents of an initialization settings ("ini") file (Fig. 3). An "ini" file has multiple paragraphs, with name, value pairs in each paragraph, to persist the various settings, such as the size of the application window when it was last closed.

Domain declaration. Developers declare a domain before use, using the `@Domains` annotation (line 1). Domains are declared on a class but are treated like fields, in that fresh domains are created for each instance of that class. For a domain `D` declared on a class `C` and two instances `o1` and `o2` of type `C`, the domains `o1.D` and `o2.D` are distinct, for distinct `o1` and `o2`.

Domain use. Each object is assigned to a single ownership domain, which does not change at runtime. Developers indicate the domain of an object by annotating each reference to that object in the program. For example, the `@Domain` annotation declares the reference `f` of type `InputFile` in the domain `L` (line 12).

```java
public class IniFile<U,L,D> {
  domain owned;
  public domain PARAGS;
  shared String filename;

  owned Hashtable<shared String, PARAGS IniParagraph<U,L,D>> paragraphs;
  PARAGS IniParagraph<U,L,D> para;
  ...
}
public class IniParagraph<U,L,D> {
  domain owned;
  L InputFile f;
  ...
}
// Root class, used for OOG construction
public class System {
  domain UI, LOGIC, DATA;
  LOGIC IniFile<UI,LOGIC,DATA> iniFile;
  ...
}
```

(a) Ownership Domains: language extensions.

```java
@Domains({ "owned", "PARAGS" })
@DomainParams({ "U", "L", "D" })
public class IniFile {
  @Domain("shared") String filename;
  @Domain("owned<shared, PARAGS<U,L,D>>") Hashtable<String, IniParagraph> paragraphs;
  @Domain("PARAGS<U,L,D>") IniParagraph para;
  ...
}
@Domains({ "owned" }) // != IniFile's owned
@DomainParams({ "U", "L", "D" })
public class IniParagraph {
  @Domain("L") InputFile f;
  ...
}
// Root class, used for OOG construction
@Domains({"UI","LOGIC","DATA"})
public class System {
  // Outer LOGIC is the domain of the reference
  // Class IniFile is parameterized with <U,L,D>
  // We bind the domain parameters as follows:
  // U := UI, L := LOGIC, D := DATA
  @Domain("LOGIC<UI,LOGIC,DATA>") IniFile iniFile;
  ...
}
```

(b) Ownership Domains: annotations.

Fig. 2: Ownership Domains: language extensions vs. annotations.

```
// Contents of IniFile "sample.ini"
[Appearance]      // IniParagraph1
MainWinXpos = 437 // Name-value pair1
MainWinYpos = 332 // Name-value pair2
...
[Display]         // IniParagraph2
ScaleMethod = 3
...
```

Fig. 3: An "ini" file has multiple paragraphs, with (name, value) pairs in each.

This example illustrates the two kinds of object hierarchy, logical containment and strict encapsulation. For example, `IniFile` declares a public domain, `PARAGS`, to hold `para` objects. Strict encapsulation is accomplished using private domains. For example, `IniFile` stores the `Hashtable` in a private domain, `owned` (line 5).

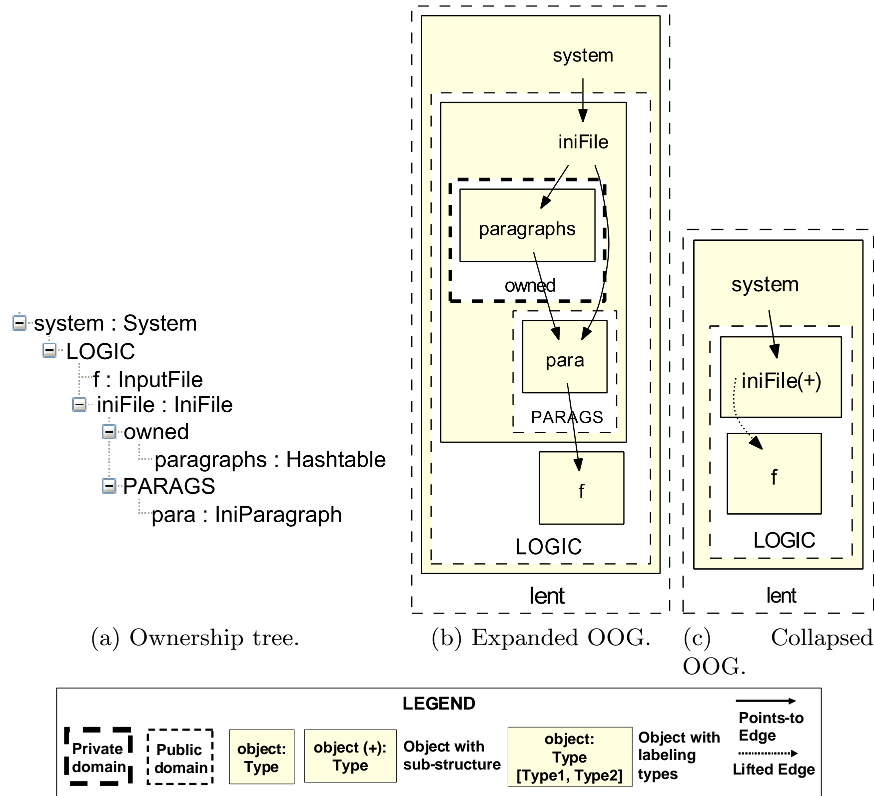
Domain parameters. Domain parameters on a type allow objects to share state. Developers declare domain parameters using the `@DomainParams` annotation. For example, the class `IniParagraph` takes the `U`, `L` and `D` domain parameters, and these are bound to the corresponding domain parameters on `IniFile`, respectively. This way, both an `IniParagraph` object and an `IniFile` object can reference the same `InputFile` object in the `L` domain parameter (see the two incoming edges from `para` and `iniFile` to `f` in the OOG).

Special annotations. There are additional annotations that add expressiveness to the type system [11]: `unique` indicates an object to which there is only one reference, such as a newly created object, or an object that is passed linearly from one domain to another. One ownership domain can temporarily lend an object to another and ensure that the second domain does not create persistent references to the object by marking it as `lent`. An object that is `shared` may be aliased globally but may not alias non-`shared` references. Unfortunately, the global nature of `shared` objects makes reasoning difficult, so it is best practice to avoid using `shared` where possible.

2.2 Review of the Ownership Object Graph (OOG)

While ownership annotations show the relationship between two objects or domains that are currently in scope, OOGs show how these relationships compose to form a global hierarchy. The extraction of an OOG from ownership annotations is subtle, because annotations are expressed in terms of locally visible domain names, which may be type parameters that refer to a domain declared elsewhere. For example, the `InputFile` object is declared to be in domain `L`, but `L` is only a domain parameter (line 12, Fig. 2b). In order to place the `InputFile` in a hierarchy, we must determine to what declared domain `L` is bound. In this case, `L` is bound to the `LOGIC` domain, (line 22, Fig. 2b) so in our OOG diagram we put the `InputFile` in the `LOGIC` domain of the `System` object (see Fig. 4b).

To start the OOG extraction process, the developer picks a root class as a starting point, the `System` class in the example (Fig. 2b). The root class is as-



(a) Ownership tree. (b) Expanded OOG. (c) Collapsed OOG.

Fig. 4: OOG for the earlier example (Fig. 2).

sumed to be instantiated into a root object. The static analysis then uses the ownership annotations in the code to impose a conceptual ownership hierarchy on the objects in the system, starting at the root object (Fig. 4a). In the ownership hierarchy, a low-level object such as `paragraphs` is underneath a more architecturally interesting object such as `iniFile`. Fig. 4b and Fig. 4c show the corresponding OOG of the above example (Fig. 2). Typically, we do not show the root object, we consider the domains declared on the root class as the top-level domains, and we call the objects in the top-level domains as top-level objects.

To construct the OOG, the static analysis abstractly interprets the program with the annotations. In particular, it maps formal domain parameters to the actual domains to which they are bound, which as described above shows `InputFile` inside the actual domain `LOGIC` on the root object. In addition, the OOG shows points-to edges that correspond to field references.

An OOG is abstract in the sense that one "canonical object" in the OOG corresponds to one or more objects in the system at runtime. Also, an OOG does not pin things down to individual objects. Instead, it abstracts objects by domains and types. For example, it merges several objects of the same or similar types that are in the same domain. If at runtime multiple instances of

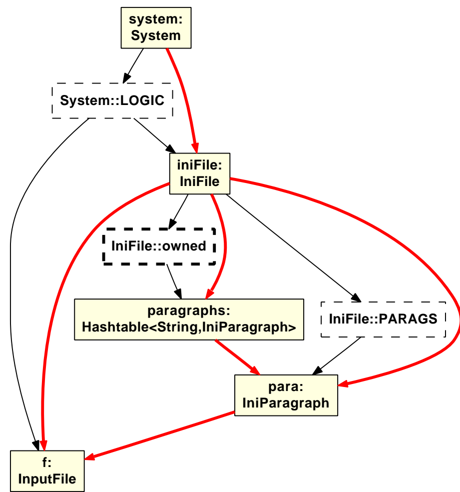
the `IniFile` class and of the `IniParagraph` class are created, and the latter were all placed into the `PARAGS` domain on the `iniFile` object, the OOG would show one `para: IniParagraph` object in the `PARAGS` domain. Developers can trace from the canonical object in the OOG to all the lines of code that may create such an object.

Aliasing invariant. By abstracting objects by domains and types, the OOG also enforces an aliasing invariant: no one runtime object appears as two different canonical objects in the OOG. To enforce this invariant, the analysis relies on the ownership domain annotations, which also give precision about aliasing, without requiring an alias analysis. The type system guarantees that two objects in different domains cannot alias, but two objects in the same domain may alias.

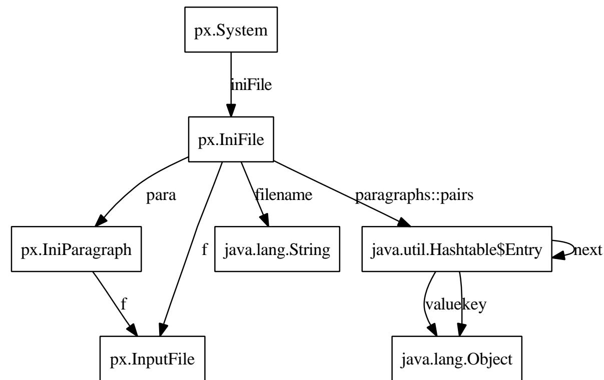
Visualization. The OOG uses box nesting to indicate containment of objects inside domains and domains inside objects (Fig. 4). Dashed-border, white-filled boxes represent domains. Solid-filled boxes represent objects. Solid edges represent field references. An object labeled `obj:T` indicates an object reference `obj` of type `T`, which we then refer to either as "object `obj`" or as a "`T` object", meaning "an instance of the `T` class". A private domain has a thick, dashed border; a public domain, a thin one. Having a hierarchical representation allows expanding or collapsing objects to control the level of visual detail. In Fig. 4b, we fully expanded the OOG. In Fig. 4c, we collapsed the sub-structure of `iniFile`. The symbol (+) on an object indicates that it has a collapsed sub-structure.

Object Graph. The internal representation of the OOG is the Object Graph (OGraph). An OGraph has two types of nodes, objects and domains. Edges between objects correspond to field reference points-to relations (the thick edges). The root of the graph is the root object, the instance of the developer-specified root class. The metrics we discuss later operate on an OGraph. As a first approximation, the nodes in an OGraph form a hierarchy, where each object node has a unique parent domain, and each domain node a unique parent object (the thin edges illustrate the ownership relations) (Fig. 5a). In fact, the graph is not a strict hierarchy. In order to handle recursive types, the graph can have cycles and have the same domain appear as the child of two objects. For more information about handling recursion, refer to [2, Chap. 3].

Abstraction by types. An OOG provides architectural abstraction primarily by ownership hierarchy. In addition, an OOG can abstract objects within each domain by their declared types. In many object-oriented systems, many types extend common base classes or implement common interfaces. One heuristic for abstraction by types merges objects in the same domain based on an ordered list of design intent types (DIT) provided by the developer. To decide whether to merge two objects of type C_1 and C_2 in a domain, the analysis finds in DIT a class or interface C, such that $C_1 <: C$ and $C_2 <: C$. If DIT does not include such a type, then abstraction by types does not apply (Fig. 6). Another heuristic merges objects whenever their types share one or more non-trivial types. A developer provides the list of trivial types (TT), which includes

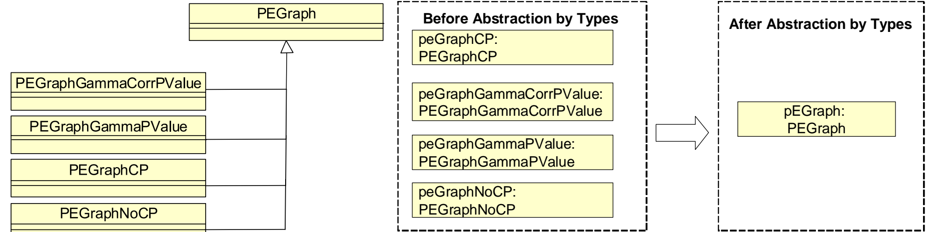


(a) Internal representation of the OOG in Fig. 4 (OGraph).



(b) Flat object graph.

Fig. 5: OGraph vs. flat object graph for the earlier example (Fig. 2).



(a) Class inheritance hierarchy.

(b) Abstraction by types on OOG.

Fig. 6: Effect of abstraction by types (example from PX system [29]).

Table 1: List of subject systems. The *Version* is stated if it is known. *Abbrev.* is the abbreviated name we use to refer to the system later in this paper.

System	Version	Abbrev.	Open Source?	Domain	Type
JHotDraw	5.3	JHD	Yes	GUI framework	Open Source
DrawLets	2.0	DL	Yes	GUI framework	Commercial
MiniDraw		MD	Yes	Board game framework	Educational
Aphyds		APD	No	Circuit design	Academic
HillClimber		HC	No	Simulation	Academic
PathwayExpress		PX	No	Bioinformatics tool	Academic
ApacheFTPServer	1.0.5	AFS	Yes	FTP server	Open Source
CryptoDB		CDB	Yes	Cryptography database	Educational

interfaces such as `Cloneable` and `Serializable` from Java Standard Library. The heuristic does not merge objects that share only trivial types as supertypes.

Flat object graph. We also extracted a flat object graph (Fig. 5b) using Womble [30] for the same example, ignoring the ownership annotations in place. The flat object graph shows low-level objects (e.g., `para:IniParagraph`) at the same level as the interesting objects (e.g., `iniFile:IniFile`).

3 Method

We first discuss the subject systems we selected (Section 3.1), and the procedure we used to add annotations and extract OOGs from those systems (Section 3.2). We then state our research questions (Section 3.3), and we answer these questions with support from metrics on the annotations (Section 3.4) and on the OOGs (Section 3.5).

3.1 Subject Systems

We selected the subject systems according to the following criteria: large enough to be more interesting than prior case studies but small enough to complete in the time available; having design information available that we could use in order to specify ownership; and covering a diversity of object-oriented applications. The systems selected are discussed below and shown in Table 1. Some aspects of the code structure, including inheritance, interface implementation, and type parametrization, are relevant in the annotation process; these aspects are summarized in Table 2. For brevity, we assign an abbreviated name to each system, and we show it between parentheses in the next paragraphs. In the rest of the paper, we refer to the systems using the abbreviated name.

JHotDraw (JHD). JHotDraw [31] is an open-source, object-oriented framework that is rich with design patterns, uses composition and inheritance heavily and has evolved through several versions. JHotDraw is a significant example

Table 2: Information about the systems' code structure, obtained using the Eclipse Metrics Plugin [25]. **KLOC** is the size of the system. **#clsss**, **#intrfcs**, **#pckgs** are the number of classes, interfaces and packages. We show the maximum and the mean for the depth of the inheritance tree. **#static mthds** and **flds** are the number of static methods and fields.

Abbr.	**KLOC**	**#clsss**	**#intrfcs**	**#pckgs**	**uses**	**inh. tree depth**		**#static**	
					generics?	**max**	**mean**	**mthds**	**flds**
JHD	18.0	224	33	8	No	8	2.41	75	121
DL	8.8	94	23	12	No	7	3.66	65	27
MD	1.4	31	17	5	Yes	6	2.00	3	10
APD	8.2	66	0	1	No	6	2.39	6	29
HC	15.6	93	7	2	No	5	3.16	5	131
PX	36	163	9	30	No	7	2.41	153	186
AFS	14.4	159	39	24	Yes	5	1.68	97	109
CDB	2.3	21	1	3	Yes	3	1.57	7	11
Total	104.7	661							

in the object-oriented community. The details of the annotation process are discussed elsewhere [2, Chap. 4.6]. The JHD OOG was used to analyze conformance against a reference architecture. It was also shown to three developers during a study exploring the questions that developers ask about object structures [6].

DrawLets (DL). DrawLets [24] is an object-oriented framework implemented in Java for building graphical applications. DrawLets supports a drawing canvas that holds figures and lets users interact with them. The details of adding annotations and extracting OOGs from DrawLets are discussed elsewhere [14, Sec. 4]. The DrawLets OOG was carefully refined by a developer who was implementing code modification tasks, to evaluate if the OOG was useful to understand the system's runtime structure [4].

MiniDraw (MD). MiniDraw [35] is a pedagogical object-oriented framework that is a scaled-down version of JHotDraw. Also, MiniDraw comes with insights into its design, discussed in a textbook [20]. The framework supports the graphical aspects of board games and other applications that let users manipulate two dimensional graphical objects. Some of the sample applications supported by MiniDraw include board games such as BreakThrough. We added annotations to the framework and to the BreakThrough sample application [13]. The MD OOG was carefully refined by a developer while she piloted code modification tasks for a controlled experiment [12, Chap. 3] to evaluate if the OOG helps developers to understand the system's runtime structure during coding activities.

Aphyds (APD). Aphyds [28] is a pedagogical circuit layout application that an electrical engineering professor wrote for one of his classes. Students in the class are given the program with several key algorithms omitted, and are asked to code the algorithms as assignments. Aphyds is interesting because it has available architectural documentation [10]. The APD OOG was used to ana-

lyze conformance against a reference architecture. The details of the annotation process are discussed elsewhere [2, Section 7.5].

HillClimber (HC). HillClimber [41] is a Java application developed by undergraduates at the University of British Columbia. HillClimber is part of a collection of Java applications to graphically demonstrate artificial intelligence algorithms, built on the `CIspace` framework. In particular, HillClimber demonstrates stochastic local search algorithms for constraint satisfaction problems. HillClimber is also interesting because it uses a framework and its architectural structure had degraded over the years. The details of the annotation process are discussed elsewhere [2, Chap. 4.7].

Apache FTP Server (AFS). Apache FTP Server [1] is an open-source implementation of a complete and portable server engine based on the standard File Transfer Protocol (FTP). We added annotations and extracted OOGs guided by the reference architecture available [49].

Pathway-Express (PX). Pathway-Express [29] is an object-oriented, web application implemented in J2EE. PX is part of the Onto-Tools developed in the Intelligent System and Bioinformatics Laboratory at Wayne State University. PX finds, builds, and displays a graphical representation of gene interactions, and has thousands of users spread across several bioinformatics research groups. We studied this system because we had access its maintainers to help us refine the OOGs. After completing the annotations and extracting an initial OOG [8, Sec. 3], we evaluated the PX OOG by showing it to a lead maintainer of the system. We then refined the OOG according to the maintainer's requests [5].

CryptoDB (CDB). CryptoDB [32] is a secure database system designed by a security expert. CryptoDB follows a database architecture that provides cryptographic protections against unauthorized access, and includes a sample implementation in Java. We selected CryptoDB because it has both a Java implementation and an informal architectural description, that guided the process of adding annotations and extracting OOGs [2, Sec. 7.8]. The OOG was then compared against a detailed runtime architecture [7].

3.2 Procedure

We now discuss the procedure for adding and checking the annotations, and extracting and refining the OOGs.

Annotation Goals. When adding annotations, our goals included generating the least number of typechecker warnings. Such warnings indicate that the annotations are inconsistent with the code and with each other. The second goal was making the fewest changes to the code, since we were interested in describing the as-built ownership structure of the legacy systems. Some changes were needed to be able to add annotations. These changes are fairly cosmetic; for example, we extracted local variables in the presence of complex expressions. Finally, the third goal was minimizing the effort involved in adding the annotations. Ideally, we need smart ownership inference tools to generate annotations. Instead, we developed tools for adding boilerplate annotations.

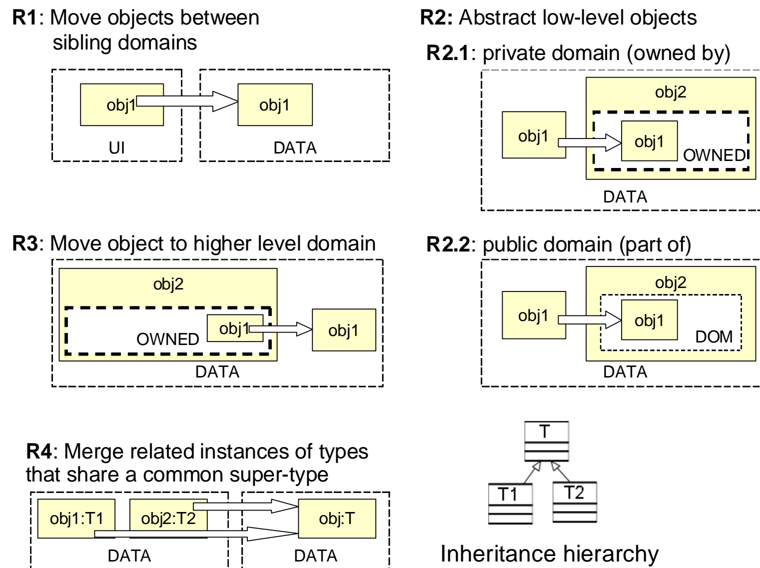


Fig. 7: Possible OOG refinements.

Extraction Goals. When extracting OOGs, our goals included the following: extracting OOGs that are sufficiently abstract, clutter-free, and that convey design intent; and extracting the as-built OOGs, which may or may not match the maintainers' expectations.

OOG Refinement. Once extracted, one can refine an initial OOG as follows (Fig. 7):

R1 Move an object between sibling domains, e.g., between the top-level domains `UI` and `LOGIC`;

R2 Abstract a low-level object from a top-level domain by pushing it underneath a more architecturally relevant object, i.e.:

- R2.1 Make an object conceptually *part of* another object, by pushing it into a public domain of the parent object;
- R2.2 Make an object *owned by* another object, by pushing it into a private domain of the parent object;

R3 Move an object from a lower-level to a higher-level domain;

R4 Collapse related instances of subtypes, using abstraction by types, by adding selected types to DIT or TT list;

R5 Other, such as make an object `shared`.

To distinguish between architecturally-relevant objects and low-level objects, one can create an application-specific list of low-level types (LLT), and consider all the instances of a type in LLT to be low-level objects. For most applications, LLT contains data structures such as `ArrayList`, `HashMap`, or utility classes. A type may be in LLT for one application, but architecturally-

relevant for another. For example, an instance of the `Rectangle` class might be architecturally-relevant for a drawing application, while in a UML editor, an instance of `Rectangle` might be part of a more complex shape and might be considered low-level.

3.3 Research Questions

Research Question (RQ1). The first research question is about annotations:

> *RQ1: Do Ownership Domains specify, within the code, hints about strict encapsulation, logical containment and architectural tiers?*

We decompose this question, and introduce several metrics that measure the degree to which Ownership Domains are used to capture the relevant design qualities. When applicable, we organize the results of the metrics according to the following scale: *significant* (80% or higher), *large* (80% – 50%), *moderate* (20% – 50%), or *partial* (20% or lower).

RQ1.1 What amount of strict encapsulation do Ownership Domains specify?
We measure the number of references annotated with private domains, and the number of objects in private domains.

RQ1.2 What amount of logical containment do Ownership Domain specify?
We measure the number of object references annotated with public domains, the number of objects in public domains, and the number of incoming edges into public domains.

RQ1.3 Do ownership domains specify architectural tiers?
We measure the number of references annotated with top-level domains, and the number of objects in top-level domains.

Research Question(RQ2). The second research question is about the OOGs:

> *RQ2: Does a global, hierarchical, Ownership Object Graph (OOG) provide more architectural abstraction compared to a flat object graph?*

We decompose this question and measure the degree to which abstraction mechanisms are used and how effective they are. For each of the following research questions, we formulate a corresponding hypothesis.

Abstraction by ownership hierarchy. Some research questions are related to abstraction by ownership hierarchy:

RQ2.1 To what extent does an OOG hide low-level objects underneath more architecturally relevant objects?
We hypothesize that the majority of objects that are of a type in LLT are at the lower levels of the OOG. We measure the average depth of low-level objects.

RQ2.2 How many low-level objects does an OOG show in the top-level domains?
We hypothesize that only a few objects of a type in LLT are in the top-level domains. We measure the number of objects that are in top-level domains and are of a type in LLT.

RQ2.3 To what extent does an OOG show architecturally-relevant objects at the top-level, compared to a flat object graph?
We hypothesize that the number of architecturally-relevant objects are one order of magnitude fewer than the number of objects in a flat graph. We measure the number of top-level objects in an OOG divided by the number of objects in a flat graph.

Abstraction by types. Other research questions are related to abstraction by types:

RQ2.4 To what extent does abstraction by types further merge objects within a domain?
We hypothesize that abstraction by types further reduces the cluttering of the OOG. We compare the OOG metrics before and after using abstraction by types.

RQ2.5 How long are the list of design intent types and list of trivial types?
When using abstraction by types, OOGs uses DIT and TT for customizing the abstraction. It is desirable that these lists are short in order to minimize effort in refining the OOG. We hypothesize that DIT and TT are much smaller, compared to the list of types (classes and interfaces) in the system. We measure the size of the lists for the various systems.

Precision. Some research questions are related to the precision of the OOG:

RQ2.6 How often does an OOG merge objects of the same or similar types that are in the same domain?
An OOG does not pin things down to individual objects. Instead, it merges several objects of the same or similar types that are in the same domain. We hypothesize that by using abstraction by ownership hierarchy and by types an OOG has sufficient precision. We measure the average number of `new` expressions in the code that correspond to one object in the OOG, and the average number of objects in the OOG that correspond to a `new` expression.

RQ2.7 How often does an OOG appear as a fully connected graph?
We hypothesize that rarely does an OOG appear as a fully connected graph. We measure the number of points-to edges, and average number of incoming edges into top-level, public, and private domains.

RQ2.8 How often does an OOG collapse all the objects in a domain into a single object?
We hypothesize that rarely does an OOG collapse all the objects in a domain into a single object, in the absence of aliasing information more precise than what ownership annotations provide. We measure the average number of objects per domain.

Overall. Other research questions are related to the combined effects of both abstraction mechanisms on the OOG:

RQ2.9 To what extent does the size of the top-level OOG increase linearly with the size of the program?
The size of the top-level OOG does not increase linearly with the size of the program. We plot the size of the top-level OOG vs. the size of the system[1].

3.4 Annotation Metrics

To describe the defined annotation metrics, we refer to the symbols defined in the FDJ abstract syntax (Fig. 1): $\overline{D}$ for a list of actual domains, $\overline{\alpha}$ and $\overline{\beta}$ for a list of domain parameters, $\overline{L}$ for a list of domain links, and $\overline{\gamma} \rightarrow \overline{\delta}$ for a domain assumption, etc. For $\overline{D}$, we distinguish between private (`domain d`) and public (`public domain d`) domains.

Object-level metrics. The object-level metrics distinguish between annotations on fields and those on variables. Variables include both local variables and formal method parameters. For each field or variable of a reference type, we record the owning domain, then compute the frequency of the occurrence of a domain, across all fields or variables of a reference type in the program.

For each type of annotation x, we record the frequency $\%x$. The annotation x can be one of the following:

- One of the special annotations: `lent shared`, or `unique`. For example, using excessively the `shared` annotation indicates that we are not reasoning about many objects in the system.
- Domain parameter p_i in scope;
- Special domain parameter, `owner`;
- A public domain of a final field, $ff.d_i$, where ff is a final field in scope and d_i is a public domain declared on the class of the field;
- Locally declared domain d_i in scope. Further, we distinguish between private domains (`domain` d_i) and public domains (`public domain` d_i);
- "other" refers to special cases, i.e., annotations on static methods, inner classes, or an explicit annotation on the receiver (by default, the receiver annotation is either "lent" or "owner").

Some annotations may not occur, so we omit the metrics that are always zero:

- `lent` annotation cannot occur on fields or method return values;
- `unique` fields, though valid in AliasJava [11], are not currently supported by the type system and the typechecker.

Class-level metrics. For each class we measure the size of annotations related to a class. These metrics correspond to the size of the sequence $|\overline{x}|$ averaged over all the classes in the system, where $\overline{x}$ can be one of the following:

- locally declared domains $\overline{D}$;
- domain parameters $\overline{\beta}$;
- domain inherits $\overline{\alpha}$;
- domain links $\overline{L}$;
- domain assumes $\overline{\gamma \rightarrow \delta}$.

[1] For correlation, we measure the value of Pearson's coefficient which is between -1 and +1, inclusively. A value of -1 indicates a perfect negative correlation between the two variables. Correspondingly, a value of +1 indicates a perfect positive correlation between the two variables. If the correlation coefficient is close to 0, then there is no linear relationship between the two variables. We consider a correlation of 0–0.1 trivial, 0.1–0.3 minor, 0.3–0.5 moderate, 0.5–0.7 large, 0.7–0.9 near perfect, and 0.9–1 perfect.

3.5 OOG Metrics

We define three categories of object graph metrics to measure the effect of abstraction by ownership hierarchy, abstraction by types, and finally, the differences with flat object graphs. To get better insights, when a metric is an average, we also compute the maximum, minimum, and standard deviation.

Abstraction by Ownership Hierarchy

First, we measure the effect of abstraction by ownership hierarchy.

Graph size. We count Number of Objects (#O), which is the sum of Number of Top-Level Objects (#TLO), Number of Objects in Public Domains (#OPD), and Number of Objects in Private Domains (#OPrD). We also count Number of Domains (#D), which is the sum of Number of Top-Level Domains (#TLD), Number of Public Domains (#PD), and Number of Private Domains (#PrD). These metrics provide insights on the size of the ownership hierarchy(#O, #D), and give hints about architectural tiers (#TLD), the amount of logical containment(#OPD) and strict encapsulation(#OPrD).

Number of Points-to Edges (#PtE) ranges between 0 and #O(#O-1), the maximum number of possible edges between the objects. The OOG is a directed graph, and there might be two directed edges with different direction between every pair of objects. This metric captures points-to edges that are different from the ownership edges in the internal representation of the OOG, which are covered by other metrics.

Cluttering (COOG) is Number of Points-to Edges (#PtE) divided by #O(#O-1). COOG ranges between 0 and 1, with a value close to 1 indicating a cluttered and imprecise graph. Imprecision occurs due to possible aliasing of references in the same domain. For example, if the type of a field is an interface, the analysis may create points-to edges to all the objects that have a class that implement the interface, and that are in the same domain. Since the analysis uses only the aliasing information provided by domains, and aims to achieve soundness, some of the points-to edges might be false positives [2, Section 2.6.3].

Maximum Depth of Ownership Hierarchy (MXD) is the longest path length in an OGraph counting only ownership edges. We consider the root object to be at level 1. For example, in a path $O_{root} \rightarrow D_1 \rightarrow O_1 \rightarrow D_2 \rightarrow O_2$, the depth is 2. In the presence of cycles, the path is truncated when a cycle is found. For example, in a path $O_{root} \rightarrow D_1 \rightarrow O_1 \rightarrow D_1 \rightarrow O_1$ the depth is 1. MXD ranges between 1 and #O-1. A value of #O-1 indicates that the ownership tree has no branches and one object per domain. A value of 2 indicates that all the objects in the OGraph except the root are in the top-level domains.

Percentage of Top-Level Objects (PTLO) is the number of objects in the top-level domains, divided by the number of objects in the OOG. This metric indicates the effectiveness of abstraction by ownership hierarchy at reducing the number of objects in the top-level domains. The number ranges between 0 and 1, with a value close to 1 indicating that most of the objects are in the top-level domains. A low number indicates that the majority of objects in an application

are architecturally irrelevant and abstraction by ownership hierarchy is effective. We measure the effectiveness of ownership hierarchy as 1-PTLO.

Number of Low-Level Objects in Top-Level Domains (#LLOTLD) is the number of low-level objects that are in the top-level domains. A small value (ideally, zero) indicates that the OOG conveys architectural abstraction. Otherwise, the OOG should be further refined.

Since *LLT* varies across applications, for this metric, we consider those types that are common for all the subject systems. To compute #LLOTLD, the list of low-level types include the Java standard library packages `java.util`, `java.awt`, or `java.lang`, and array of primitive types. Examples of low-level types are `java.util.List`, `java.awt.Polygon`, `java.lang.String`, `java.lang.Object`, `java.lang.Boolean`, and `int[ ][ ]`.

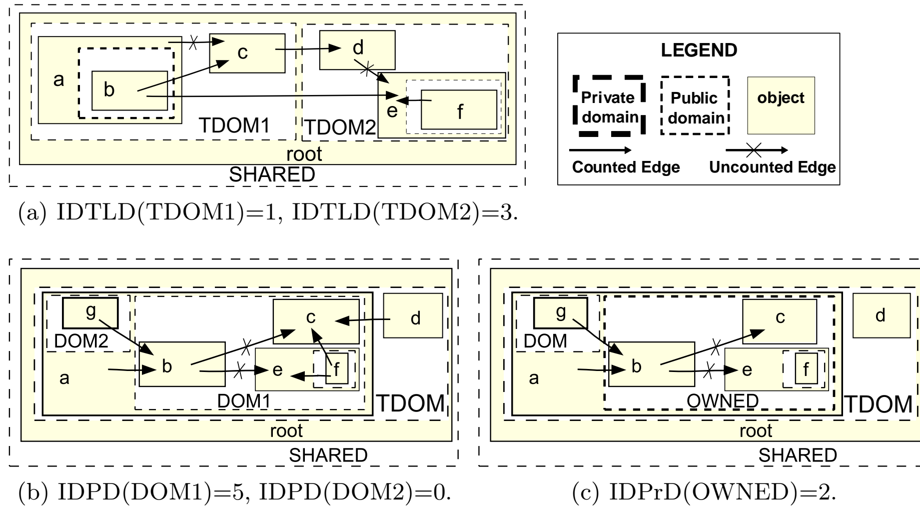
Depth of Low-Level Objects (LLOD) is the sum of the depth of the low-level objects divided by the total number of low-level objects. It ranges from 0 to maximum depth of the ownership hierarchy, where a high value indicates that an OOG hides low-level objects underneath more architectural ones.

Average Public Domains per Object (PDO) (PrDO, respectively) is the sum of public (private, respectively) domains divided by the total number of objects. PDO and PrDO provide insights about the number of internal nodes in an OOG.

Average In-Degree per Top-Level Domain (IDTLD) is the number of incoming points-to edges for every object in a top-level domain, divided by #TLO. We count only those points-to edges that have as a source a non-sibling object, i.e., an object that has a different parent in the ownership hierarchy (Fig. 8a). IDTLD ranges between 0 and #O-1, with the higher limit indicating a cluttered OOG at the top-level. It indicates how popular the objects in top-level domains are. However, if the number is high, the graph may be too cluttered, and more difficult to understand.

Average In-Degree per Public Domain (IDPD) is the number of incoming points-to edges for objects in a public domain divided by the total number of objects in a public domain. We count only those points-to edges that have as a source a non-sibling object (Fig. 8b). It ranges between 0 and #O-1, with the higher limit indicating a cluttered OOG. Public domains logically encapsulate objects, without restricting access to these objects from the outside. IDPD gives an indication of how popular the objects in public domains are. A high value justifies the need for public domains.

Average In-Degree per Private Domain (IDPrD) is the number of incoming points-to edges for objects in a private domain divided by the total number of objects in a private domain. Objects in private domains are less accessible given the restrictions imposed by ownership types. They are accessible to the parent object, to other objects in the same domain, or to other objects in sibling domains (Fig. 8c). We intend to use it for comparison with IDPD and IDTLD. A significant difference between IDPrD and the previous two metrics confirms the need of public domains.



(a) IDTLD(TDOM1)=1, IDTLD(TDOM2)=3.

(b) IDPD(DOM1)=5, IDPD(DOM2)=0.

(c) IDPrD(OWNED)=2.

Fig. 8: Incoming edges from non-sibling objects in top-level, public, and private domains.

Average Objects per Domain (AOD) measures the average number of objects in a domain. The number can be 0 for domains that are only declared without being used. A high value indicates that the graph is precise and does not excessively merge objects. We use this metric to verify the hypothesis that OOGs after abstraction by types provide sufficient precision. We do not include in this metric `shared` objects, which may be globally aliased because little reasoning can be done about them.

Average Trace to Code per Object (ATCO) is the average number of object allocation expressions in the code that are represented by one object in the OOG. This metric measures the number of traceability links between an object in the OOG and the `new` expressions in the code. ATCO illustrates how the OOG is summarizing `new` expressions with an object in the OOG. To distinguish between objects created for two `new` expressions, a developer can use separate domains, unless the code assigns to each other the references that are in different domains (Fig. 9). A value of ATCO close to 1 supports the hypothesis that rarely such an assignment exists.

Average Scattering of Objects (ASO) is the average number of object allocation expressions in different files that are merged into one object in the OOG. ASO is similar to ATCO but accounts only for the `new` expressions that are scattered across different files.

Average Objects of the Same Class (AOSC) is the average number of instances of the same class owned by different domains. A number greater than 1 indicates that the same class is instantiated in different contexts, and the OOG shows separate objects for these instances. AOSC highlights the difference between an OOG and a class diagram, which shows each class as one node.

```
1 class C{
2  domain DOM1;
3  domain DOM2;
4  A<DOM1> a1 = new A<DOM1>();
5  A<DOM2> a2 = new A<DOM2>();
6  // a1=a2 if such an assignment exists, DOM1 = DOM2
7 }
```

Fig. 9: Placing objects in separate domains.

AOSC supports our hypothesis that an OOG has more precision than a class diagram and distinguishes between different instances of the same class.

Abstraction by types

Next, we measure the effect of abstraction by types in combination with abstraction by ownership hierarchy.

Number of Top-Level Objects after Abstraction by Types (ABTTLO) measures the size of top-level OOG using both abstraction mechanisms. A rule of thumb in architectural documentation is to have 5 to 7 components per tier [33], which in our case translates in having 5 to 7 objects per top-level domain. Ideally, ABTTLO ranges between $5\times$#TLD and $7\times$#TLD.

Number of Design Intent Types (DIT) is the number of types in the *DIT* list. A small number indicates that the system has a small number of key interfaces or base classes that many classes implement or extend from.

Number of Trivial Types (NTT) is the number of types in the *TT* list. The list includes by default types such as `java.io.Serializable`, `java.lang.Cloneable`, `java.lang.Iterable`, and `java.lang.Object`. The list of trivial types is typically reusable across systems.

In the following metrics, we take 1 minus the value of the division so that they express the reduction of the abstraction mechanisms.

Abstraction by Types (ABTF) is Average Objects per Domain (AOD) after abstraction by types, divided by AOD before abstraction by types, and indicates the effectiveness of abstraction by types at reducing the cluttering of the OOG by reducing the number of objects at all levels.

Abstraction by Types at Top-Level (ABTTL) is the Number of Top-Level Objects after Abstraction by Types (ABTTLO) divided by the Number of Top-Level Objects (#TLO) before abstraction by types, and indicates the effectiveness of abstraction by types at reducing the cluttering of the OOG at the top level.

Abstraction by Ownership Hierarchy and by Types (ABHBT) is the Number of Top-Level Objects (#TLO) after abstraction by types, divided by the Number of Objects (#O) before abstraction by types and indicates the combined effectiveness of abstraction by ownership hierarchy and abstraction by types at reducing the cluttering of the OOG at the top-level.

Flat object graphs

Finally, we compare the OOG against flat object graphs that we extract from the same programs using Womble [30], by ignoring the ownership annotations in place. A flat object graph has nodes that represent objects with no hierarchical organization. It shows objects representing data structures at the same level as architecturally relevant objects, and may not be suitable to express the system's architecture. By design, Womble graph is unsound with respect to aliasing, i.e., multiple nodes in the flat object graph correspond to the same runtime object.

Percentage of Low-Level Objects in Flat Graph (LFO) is the number of low-level objects divided by the total number of objects in a flat graph. LFO indicates what percentage of the objects in a flat graph are not architecturally relevant. The metric ranges between 0 and 1 with higher values indicating a higher density of data structures.

Hierarchical Reduction (HR) is the number of objects in a flat graph divided by the Number of Top-Level Objects (#TLO) in an OOG with abstraction by types. HR estimates the effectiveness of both abstraction mechanisms compared to a flat object graph. Effective abstraction mechanisms would reduce the number of top-level objects by a number close to an order of magnitude.

4 Empirical Results

We first report the quantitative results[2] of the annotation metrics (Section 4.1) and the OOG metrics (Section 4.2) across all systems. Then, for a few systems, we study how the OOG metrics evolved across two main refinement iterations (Section 4.3).

4.1 Annotation Metrics

We first report the results of the annotation metrics (Tables 3 and 4).

Object-level metrics. We first discuss the metric results regarding object references, namely fields, local variables, method parameters and method return values (Fig. 10). For convenience, we summarize the values of a metric as a boxplot, which shows: minimum, lower quartile, median, upper quartile, and maximum. A boxplot may also indicate outliers (if any).

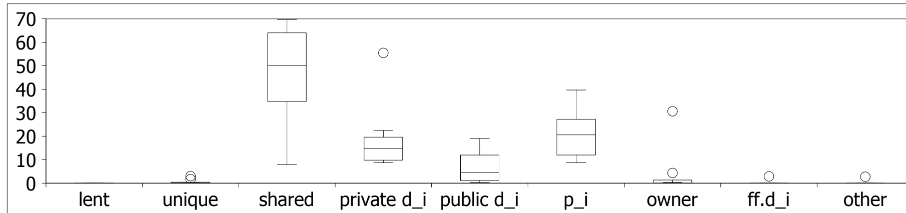
Regarding fields, there is a high proportion of `shared` annotations (Fig. 10a). This is understandable, since we used `shared` for objects such as `String`, `Font` and `Color`, among others. One system in particular, PX, has nearly 70% of its fields annotated as `shared` (Fig. 11a).

The systems that have a carefully refined OOG have a higher proportion of fields in public domains. In particular MD has about 19% of the fields in public domains (Fig. 11d). On the other hand, most of the subject systems have between 10% and 20% of the fields in private domains. In particular, APD has 55% of the fields in private domains (Fig. 11e).

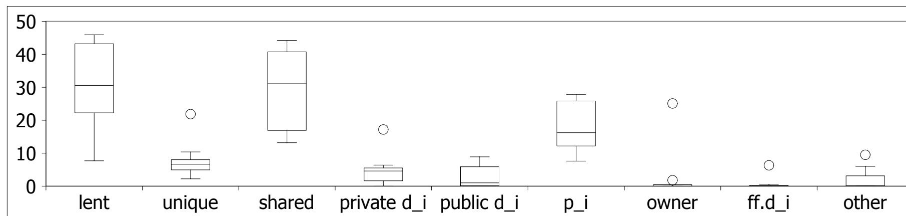
[2] All the data generated during this study is available at the following location: `http://www.cs.wayne.edu/~mabianto/oog_data/`

Table 3: Annotation metrics.

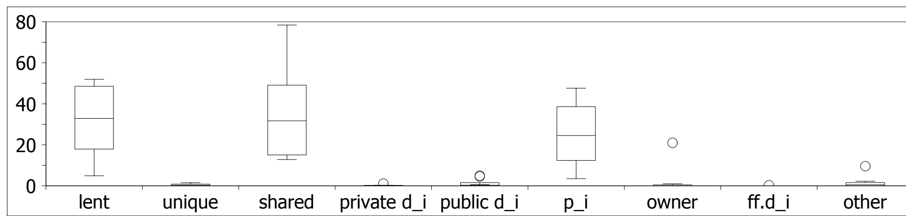
	JHD		DL		MD		APD		HC		PX		AFS		CDB	
	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#
Fields																
shared	40.4	128	36.6	49	29.3	17	7.9	23	64.8	344	69.8	609	60.0	242	63.8	44
private d	22.4	71	18.7	25	12.1	7	55.5	162	9.4	50	16.9	147	9.9	40	8.7	6
public d	1.3	4	0.8	1	19.0	11	7.5	22	1.5	8	0.8	7	13.2	53	11.6	8
p_i	36.0	114	10.4	14	39.7	23	24.3	71	24.3	129	12.5	109	16.9	68	8.7	6
owner	0.0	0	30.6	41	0.0	0	0.3	1	0.0	0	0.0	0	0.0	0	4.3	3
ff.d	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0	2.9	2
other	0.0	0	0.0	0	0.0	0	2.7	8	0.0	0	0.0	0	0.0	0	0.0	0
total	100	317	100	134	100	58	100	292	100	531	100	872	100	403	100	69
Local Variables																
lent	25.6	380	43.8	248	25.6	23	45.9	237	43.0	414	35.5	911	12.3	108	7.7	17
unique	21.9	324	3.9	22	2.2	2	6.8	35	10.4	100	5.3	136	6.5	57	7.2	16
shared	16.3	241	17.1	97	31.1	28	13.2	68	31.1	299	43.6	1120	44.2	388	39.8	88
private d	0.7	10	1.9	11	4.4	4	5.2	27	0.0	0	4.8	122	6.4	56	17.2	38
public d	0.4	6	0.0	0	8.9	8	5.4	28	0.0	0	0.2	4	1.6	14	7.2	16
p_i	25.5	378	7.6	43	27.8	25	16.9	87	15.6	150	10.7	276	26.8	235	12.7	28
owner	0.0	0	25.1	142	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0	1.8	4
ff.d	0.1	2	0.2	1	0.0	0	0.6	3	0.0	0	0.0	0	0.0	0	6.3	14
other	9.5	141	0.3	2	0.0	0	6.0	31	0.0	0	0.0	1	2.2	19	0.0	0
total	100	1482	100	566	100	90	100	516	100	963	100	2570	100	877	100	221
Method Parameters																
lent	41.5	575	51.9	401	16.5	31	51.2	189	47.6	386	24.3	361	18.4	191	4.9	5
unique	0.2	3	0.7	5	0.0	0	1.4	5	0.4	3	1.6	23	0.0	0	0.0	0
shared	15.7	217	12.8	99	45.2	85	13.3	49	29.5	239	60.7	903	33.9	352	78.4	80
private d	0.3	4	0.0	0	0.0	0	1.1	4	0.0	0	0.2	3	0.0	0	0.0	0
public d	0.6	8	0.5	4	0.0	0	4.6	17	0.0	0	0.0	0	0.1	1	4.9	5
p_i	39.6	549	3.5	27	38.3	72	26.6	98	22.5	182	13.3	197	47.6	494	9.8	10
owner	0.0	0	21.0	162	0.0	0	0.3	1	0.0	0	0.0	0	0.0	0	1.0	1
ff.d	0.0	0	0.0	0	0.0	0	0.3	1	0.0	0	0.0	0	0.0	0	0.0	0
other	2.2	31	9.6	74	0.0	0	1.4	5	0.0	0	0.0	0	0.0	0	1.0	1
total	100	1387	100	772	100	188	100	369	100	810	100	1487	100	1038	100	102
Method Return Values																
unique	18.8	126	9.3	31	9.1	5	17.2	25	5.5	13	29.0	204	4.8	24	15.4	10
shared	35.5	238	41.9	139	43.6	24	18.6	27	46.2	109	51.9	365	65.9	329	70.8	46
private d	1.3	9	2.7	9	0.0	0	1.4	2	0.0	0	0.0	0	1.6	8	0.0	0
public d	0.9	6	0.0	0	7.3	4	13.1	19	0.0	0	0.1	1	0.4	2	6.2	4
p_i	42.2	283	2.4	8	40.0	22	47.6	69	48.3	114	19.0	134	27.3	136	3.1	2
owner	0.0	0	32.8	109	0.0	0	0.7	1	0.0	0	0.0	0	0.0	0	3.1	2
ff.d	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0	1.5	1
other	1.3	9	10.8	36	0.0	0	1.4	2	0.0	0	0.0	0	0.0	0	0.0	0
total	100	671	100	332	100	55	100	145	100	236	100	704	100	499	100	65
Class Metrics																
private d	0.52	133	0.98	116	0.58	28	0.88	58	0.59	59	0.94	161	0.79	159	0.5	11
public d	0.04	9	0.04	5	0.23	11	0.29	19	0.13	13	0.03	5	0.07	15	0.36	8
p_i	0.43	111	0.18	21	1.02	49	1.97	130	0.84	84	1.7	293	0.83	168	0.55	12
inherits	1.62	416	0.56	66	1.67	80	0.15	10	0.57	57	0.75	129	1.46	294	0	0
links	0.16	41	0.1	12	0	0	0	0	0	0	0	0	0	0	0.82	18
assumes	3.43	881	0.84	99	0	0	0.03	2	1.1	110	0	0	0	0	0.82	18



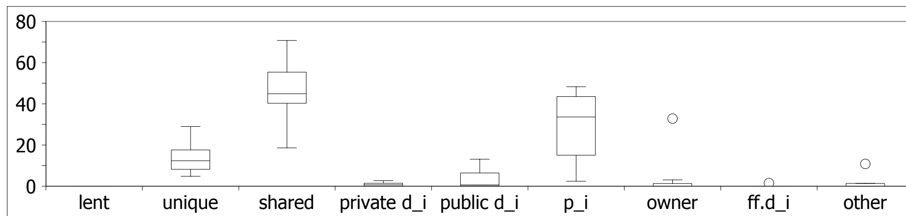
(a) Field annotations.



(b) Local variable annotations.



(c) Method parameter annotations.



(d) Method return value annotations.

Fig. 10: Distribution of annotations per subject system.

Regarding local variables, a high proportion of them have the `lent` annotation, which is unsurprising, since `lent` is used for temporary aliases (Fig. 10b). There is one exception, JHD, which has one third of its local variables annotated as `unique` (Fig. 11c).

Among method parameters, a large proportion are `shared`, `lent` or use domain parameters (Fig. 10c). In particular, in CDB, almost 80% of the method parameter annotations are `shared`, since most of them are of type `String`.

Table 4: Global annotation metrics (excluding method return values).

	JHD		DL		MD		APD		HC		PX		AFS		CDB	
	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%
lent	955	30.0	649	44.4	54	16.1	426	36.5	800	34.8	1272	25.9	299	12.9	22	5.6
shared	586	18.4	245	16.7	130	38.7	140	12.0	882	38.3	2632	53.6	982	42.4	212	54.1
private d	85	2.7	36	2.5	11	3.3	193	16.5	50	2.2	272	5.5	96	4.1	44	11.2
public d	18	0.6	5	0.3	19	5.7	67	5.7	8	0.3	11	0.2	68	2.9	29	7.4
unique	324	10.2	22	1.5	2	0.6	35	3.0	100	4.3	136	2.8	57	2.5	16	4.1
p_i/owner	1041	32.7	429	29.3	120	35.7	258	22.1	461	20.0	582	11.9	797	34.4	52	13.3
$ff.d_i$	2	0.1	1	0.1	0	0.0	4	0.3	0	0.0	0	0.0	0	0.0	16	4.1
other	172	5.4	76	5.2	0	0.0	44	3.8	0	0.0	1	0.0	19	0.8	1	0.3

Among method return values, a large percentage is `unique`, which indicates that objects are being passed linearly (Fig. 10d). JHD in particular has almost 40% of the return values as `unique` (Fig. 11c).

We observe outliers for “other” annotations. They occur in DL for method domain parameters used for the formal parameters or return value of static methods. In JHD, a high proportion of “other” annotations is for local variables in inner classes, and a small proportion for static methods. APD is the only system that has “other” annotations for the fields in inner classes (Fig. 11f).

Overall, we observe that the percentage of `shared` annotations is lower for local variables and a higher for method parameters (Fig. 11a). However, for most systems, the percentage of `lent` is higher than `shared` for method parameters.

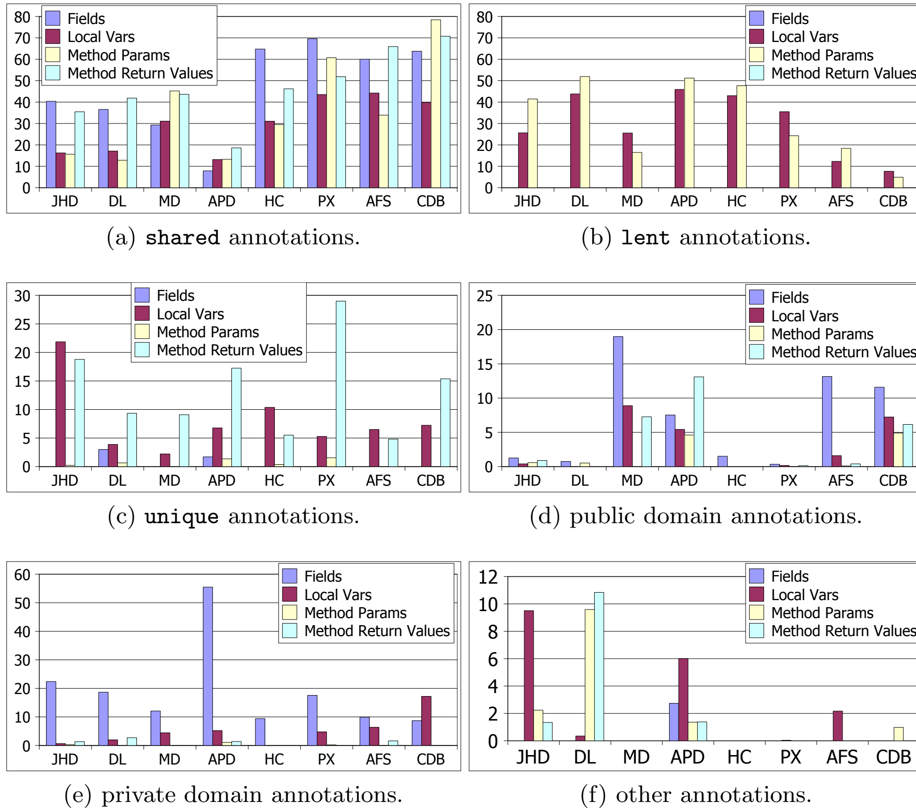
In general domain parameters are heavily used, with all the systems using them. In particular, only in DL `owner` replaces one of the domain parameters.

Class-level metrics. Next, we discuss the metric results for class-level declarations, namely locally declared (private or public) domains, domain parameters, domain inheritance, domain links and domain link assumptions (Fig. 12).

Regarding locally declared domains, the average number of private domains is almost 1.0 since every class declares an `owned` private domain, except interfaces. Some systems, such as MD, have a much higher proportion of interfaces compared to others, such as PX. The systems that have refined annotations and OOGs, such as MD, have a higher average number of public domains per class to express design intent.

Regarding domain parameters, a class has on average one domain parameter, and inherits 1–2 domain parameters. This is unsurprising since most of the systems are annotated according to a two-tiered (Document-View) or a three-tiered (Model-View-Controller) architecture. DL made use of the `owner` annotation, and saved one domain parameter per class.

Regarding domain inheritance, the number is higher than the number of domain parameters for the systems that have a richer type structure, such as MD and DL, compared to the systems that do not use inheritance, such as PX, HC, and APD. Regarding domain links, not all systems have domain link annotations.



(a) **`shared`** annotations.

(b) **`lent`** annotations.

(c) **`unique`** annotations.

(d) public domain annotations.

(e) private domain annotations.

(f) other annotations.

Fig. 11: Percentage of `shared`, `lent`, `unique`, and public domain annotations.

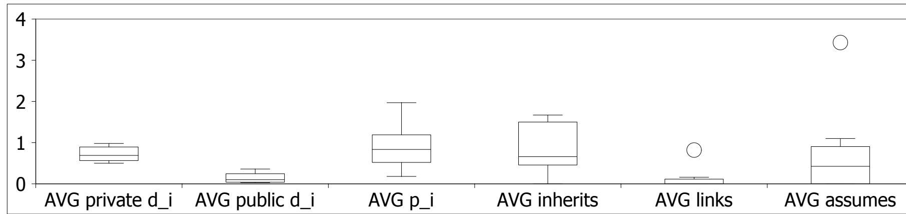


Fig. 12: Class annotations.

4.2 OOG Metrics (across all systems)

We now report the results of the OOG metrics (Table 5). The table has two columns for each system before and after abstraction by types.

Abstraction by Ownership Hierarchy

Graph size. Number of Objects (#O) ranges between 27 and 525, while Number of Top-Level Objects (#TLO) ranges between 8 and 120. For both metrics, the

Table 5: OOG metrics.

	JHD		DL		MD		APD		HC		PX		AFS		CDB	
		abt		abt		abt		abt		abt		abt		abt		abt
#O	525	461	313	284	37	33	133	127	99	60	240	222	163	102	27	25
#TLO	120	18	41	15	14	12	11	11	55	16	59	46	55	45	8	7
#OPD	3	3	1	1	9	7	16	16	16	16	2	2	54	5	5	5
#OPrD	402	440	271	268	14	14	106	100	28	28	179	174	54	52	14	13
#D	173	126	40	14	14	14	38	38	26	14	80	75	35	32	17	16
#TLD	3	3	2	2	3	3	2	2	3	3	3	3	3	3	4	4
PD	3	2	1	1	5	5	10	10	8	1	2	2	5	5	4	4
PrD	167	121	37	11	6	6	26	26	15	10	75	70	27	24	9	8
#PtE	5608	620	334	89	63	53	162	166	350	52	306	234	157	94	29	30
COOG	2.04	0.29	0.34	0.11	4.73	5.02	0.92	1.04	3.61	1.47	0.53	0.48	0.59	0.91	4.13	5.00
MXD	4	5	3	3	4	4	5	5	4	4	5	5	4	4	3	3
AVGD	2.98	3.36	2.86	2.94	2.68	2.70	3.29	3.30	2.47	2.78	3.18	3.24	2.75	2.69	2.67	2.68
PTLO	0.23	0.04	0.13	0.05	0.38	0.36	0.08	0.09	0.56	0.27	0.25	0.21	0.34	0.44	0.30	0.28
LLOTL	0	0	0	0	0	0	0	0	2	2	0	0	0	0	0	0
LLOD	3.32	3.45	3.00	3.00	3.24	3.24	3.65	3.65	3.04	3.04	3.55	3.55	3.32	3.32	3.08	3.08
PDO	0.01	0.01	0.00	0.00	0.14	0.16	0.09	0.08	0.08	0.14	0.01	0.01	0.04	0.05	0.15	0.17
MIN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MAX	1	2	1	1	3	3	2	2	1	8	1	1	2	2	1	1
STD	0.08	0.10	0.06	0.06	0.54	0.57	0.31	0.30	0.28	1.04	0.09	0.09	0.22	0.30	0.37	0.38
PrDO	0.35	0.40	0.12	0.13	0.17	0.19	0.20	0.21	0.15	0.25	0.33	0.34	0.17	0.27	0.35	0.38
MIN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MAX	1	31	1	9	1	1	1	1	1	5	1	2	1	2	1	2
STD	0.48	1.86	0.32	0.91	0.38	0.40	0.40	0.41	0.36	0.76	0.47	0.52	0.37	0.51	0.49	0.58
IDTLD	23.9	14.9	3.73	1.73	1.86	1.75	1.91	1.91	0.35	0.44	1.05	0.93	0.51	0.36	0.88	0.86
MIN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MAX	56	47	7	6	9	8	7	7	3	2	6	5	4	3	2	2
STD	20.1	17.8	2.46	2.09	3.13	2.96	1.81	1.81	0.78	0.73	1.18	0.98	0.98	0.77	0.64	0.69
IDPD	0.67	0.00	0.00	0.00	0.89	0.71	1.19	1.19	1.38	0.19	1.00	1.00	0.98	0.60	1.60	1.60
MIN	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1
MAX	1	0	0	0	1	1	8	8	2	1	1	1	1	1	4	4
STD	0.58	0.00	0.00	0.00	0.33	0.49	1.94	1.94	0.50	0.40	0.00	0.00	0.14	0.55	1.34	1.34
IDPrD	0.52	0.31	0.10	0.04	0.43	0.43	0.86	0.85	0.82	0.46	0.66	0.61	0.37	0.38	0.36	0.38
MIN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MAX	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1
STD	0.50	0.51	0.30	0.20	0.51	0.51	0.35	0.36	0.39	0.51	0.47	0.49	0.49	0.49	0.50	0.51
AOD	3.17	1.82	7.80	5.36	2.57	2.29	3.47	3.32	3.77	2.50	2.99	2.79	4.63	2.94	1.53	1.44
MAX	74	22	34	8	9	7	12	10	23	7	27	23	43	22	5	4
STD	7.03	2.21	4.77	3.10	2.14	1.73	3.06	2.81	5.81	2.07	4.21	3.78	8.89	5.05	1.07	0.81
ATCO	1.28	1.64	1.37	1.49	1.08	1.21	1.68	1.76	1.49	2.35	1.38	1.49	1.23	1.97	1.44	1.56
MAX	13	56	7	12	3	3	12	12	5	21	15	15	12	43	6	6
STD	1.02	3.16	0.76	1.22	0.36	0.6	1.59	1.71	0.85	2.9	1.21	1.34	0.99	4.40	1.09	1.16
ASO	1.03	1.09	1.26	1.32	1.03	1.06	1.00	1.00	1.09	1.40	1.05	1.08	1.12	1.21	1.15	1.16
MAX	3	10	6	10	2	2	1	1	5	12	2	4	3	5	2	2
STD	0.20	0.68	0.69	0.94	0.16	0.24	0.00	0.00	0.45	1.63	0.23	0.33	0.4	0.63	0.36	0.37
AOSC	3.67	8.23	6.39	9.47	1.19	1.18	1.58	1.57	1.43	1.62	1.80	1.87	1.12	1.19	1.13	1.14
MAX	38	57	34	32	3	3	8	8	9	9	26	26	4	4	2	2
STD	7.2	12.3	11.6	13.8	0.48	0.48	1.36	1.34	1.56	1.78	2.76	2.9	0.42	0.52	0.34	0.35

lowest value occur for CDB, and the highest value occurs for JHD. The size of the OOG does not necessarily grow at the same rate as the size of the program. For example, in KLOC, DL is roughly one quarter the size of PX, but the DL OOG has more nodes than the PX OOG.

Number of Objects in Public Domains (#OPD) ranges between 1 and 54 and is lower than Number of Objects in Private Domains (#OPrD), which ranges between 14 and 402. Number of Domains (#D) ranges between 14 in MD and 173 in JHD. Number of Top-Level Domains (#TLD) is relatively constant, 2, 3, or 4 and indicates a 2-tier or 3-tier architecture. Number of Public Domains (#PD) ranges between 1 and 10, and indicates a small number of public domains. In contrast, Number of Private Domains (#PrD) ranges between 6 and 167, and indicates that the majority of domains are private domains. Indeed, most of the classes use a private domain (Fig. 13).

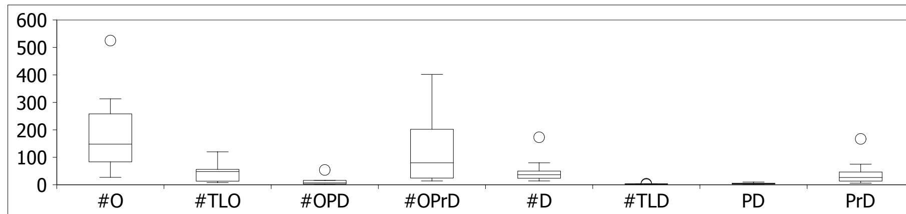


Fig. 13: Object graph size.

Number of Points-to Edges (#PtE) ranges between 29 in CDB and 5608 in JHD, and is in general of the same order of magnitude as #O. It indicates a relatively small number of edges, which can highlight interesting reference relations. It also indicates that there are objects with no incoming or outgoing points-to edges. For example, there are 47 such objects in AFS, and 237 objects in DL. In particular, JHD has 10 times more edges than objects and indicates a higher level of cluttering. After applying abstraction by types, the value substantially decreases. However, this metric does not give us sufficient details on the effectiveness of abstraction by types, which we discuss in a later metric.

Cluttering (COOG) ranges between 0.0034 and 0.0473, and indicates a graph with low cluttering. Such a graph can make visually obvious the runtime structure of the program. Since the value is very low, we display the value multiplied by 100.

Maximum Depth of Ownership Hierarchy (MXD) ranges between 3 and 5 and indicates that the depth is relatively constant with respect to the size of the OOGs. The average is relatively constant, and ranges between 2.47 and 3.29. To better understand these values, we also compute the median, which is 2, 3 or 4. For more insight, we computed Pearson's correlation coefficient between MXD and KLOC. The coefficient (0.58) indicates a large positive correlation. Indeed, MXD is relatively constant (Fig. 14) while the size of the system varies. In particular, the median depth for HC is 2, and indicates that in this particular

case ownership hierarchy is less effective. The following metric illustrates better the effectiveness of having an ownership hierarchy.

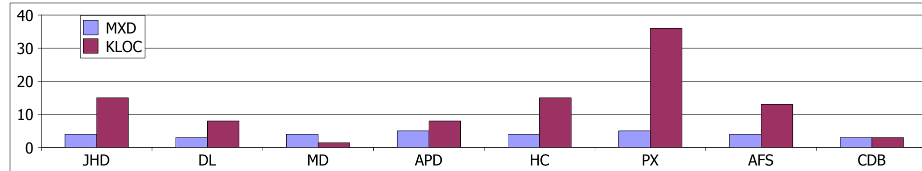


Fig. 14: KLOC vs. OOG depth.

Percentage of Top-Level Objects (PTLO) ranges between 0.08 and 0.56, with the smallest value for APD, and indicates that abstraction by ownership hierarchy is effective. Given that the depth of the graph is relatively constant, one can assume that the number of top-level objects increases as the graph grows. However, there is no linear relationship between the percentage of top-level objects is and the size of the program (Pearson's correlation coefficient ≈ 0.02). Using ownership hierarchy, developers can collapse and hide more than half of the objects in most cases. For HC, where the median depth is 2, PTLO is 0.56, and indicates that ownership hierarchy hides 44% of the objects in the top-level domains. In the case of DL, this percentage increases to 87% of the objects. For APD, it increases to 92%. So, in some cases, abstraction by ownership hierarchy reduces the number of objects at the top level by one order of magnitude compared to the entire graph.

Number of Low-Level Objects in Top-Level Domains (#LLOTLD) is zero in most systems, and supports our hypothesis that rarely does an OOG show low-level objects in top-level domains.

Depth of Low-Level Objects (LLOD) ranges between 3 and 3.65, and indicates that the majority of low-level objects are in the lower parts of the ownership trees. It indicates that the low-level objects are pushed underneath more architecturally-relevant ones. One system in particular, DL, has all its low-level objects at level 3, the lowest level of the ownership tree.

Average Public Domains per Object (PDO) ranges between 0 and 0.15, while Average Private Domains per Object (PrDO) ranges between 0.12 and 0.35, and indicates that less than half of the objects are represented as internal nodes of the ownership hierarchy. This is unsurprising given that the depth of the OOG is almost constant and relatively low. At most, an object has 3 public domains and one private domain. Since there are more private domains declared, it is unsurprising that PrDO is higher than PDO.

Average In-Degree per Top-Level Domain (IDTLD) ranges between 0.45 and 23.94. For most systems, IDTLD is low with an outlier for JHD (Fig. 15). Before abstraction by types, the JHD OOG is cluttered, since there are many concrete sub-classes of the interfaces `Command`, `Figure`, or `Tool`. In the code the interface `Command` has a field of type `Figure`. The OOG shows

points-to edges from all the instances of a class that extends `Command` in domain `CONTROLLER` to all the instances of a class that extends `Figure` in domain `MODEL`. The high number of edges lead to an almost fully connected graph. After abstraction by types, the values of IDTLD decrease for all systems and range between 0.36 and 14.89. This reduction indicates that abstraction by types can also reduce the edge cluttering in the OOG.

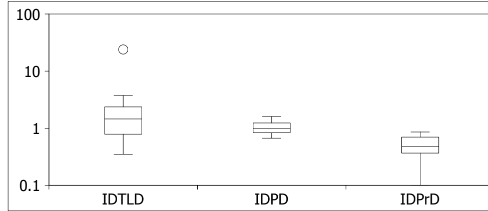


Fig. 15: In degree per top-level, public and private domains.

Average In-Degree per Public Domain (IDPD) ranges between 0 and 1.60, and indicates that objects in public domains are accessed by outside objects, other than their siblings or parents. Such a relation would not be possible in the absence of public domains. APD in particular has 8 incoming points-to edges. The low IDPD indicates that public domains do not introduce additional clutter. For example, IDPD is 0.51 for JHD, indicating a lower number of points-to edges for objects in public domains compared to the top-level objects.

For additional insight, we counted the in-degree for objects in public and top-level domains from non-sibling objects, and projected these values against each level of the JHD OOG. The x-axis indicates the level, while the y-axis indicates the in-degree values that occur at that level. We observe that the high in-degree occurs only for top-level objects, while objects at level 3 and 4 have only one incoming edge from a non-sibling object (Fig. 16).

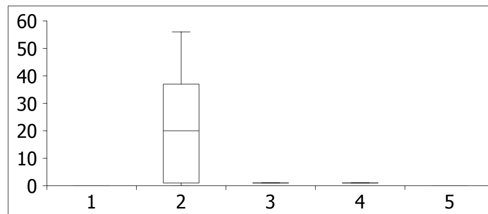


Fig. 16: In-degree from non-siblings objects in JHD.

Average In-Degree per Private Domain (IDPrD) ranges between 0.10 and 0.86, and indicates that objects in private domains have at most one incom-

ing points-to edge. This is unsurprising since private domains restrict access to their objects, and few systems use domain links, which can relax this restriction.

Average Objects per Domain (AOD) ranges between 1.53 and 7.80, and indicates a relatively low number of objects per domain. For additional insight, we compute the median and the standard deviation (Fig. 17). For all the systems, the median is less than 10, but given the high number of outliers we cannot draw further conclusions. A closer inspection of the numbers reveals that in most cases, the top-level domains have the highest number of objects, up to 74 for JHD. One system in particular, APD has no outliers. For APD, the highest number of objects occurs for the private domain of a top-level object of type `PlaceRouteUI` that has 12 objects. After applying abstraction by types, the values decreases and range between 1.44 and 5.36 with a maximum of 23 objects in a top level domain of PX, and 22 objects in the top level domains of JHD. So, abstraction by types can reduce the clutter in the OOG. This metric also indicates that the OOG is precise and does not excessively merge all the objects in the same domain into one object.

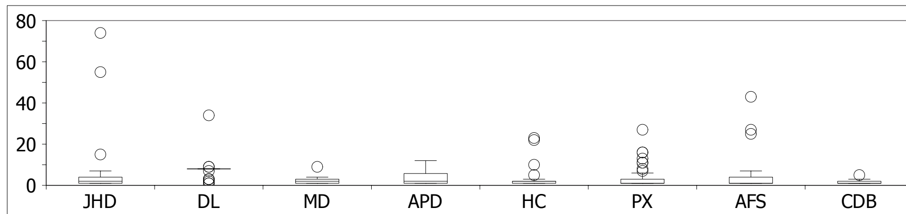


Fig. 17: Average Objects per Domain (AOD).

Average Trace to Code per Object (ATCO) ranges between 1.08 and 1.68 and indicates that there are objects in the OOG that correspond to multiple lines of code, ones that have `new` expressions. For additional insight, we measured the median and the standard deviation, and we highlighted the outliers (Fig. 18). For example, an object in the PX OOG corresponds to 15 instances of type `LinkedList<FunctionBar>` in the class `BarGraphPanel` and each of these objects represents a dataset represented as a block in the bar chart. Similarly, an object in the AFS OOG corresponds to 12 instances of type `AtomicInteger` in the class `DefaultFtpStatistics`, to represent various statistics stored by the server such as the number of logins, failed logins, and uploaded files. Overall, the low value of the median of ATCO (1 or 2), and the relatively small number of outliers, indicate that a relatively small number of `new` expressions correspond to the same object in the OOG.

Average Scattering of Objects (ASO) ranges between 1 and 1.26, while the maximum value ranges between 1 and 6. ASO indicates that an object in the OOG may correspond to multiple lines of code in different files. For example, in DL, there are 7 instances of the class `DrawingPoint` in 6 different classes, that are all represented by one object in the OOG.

Table 6: OOG metrics – abstraction by types.

	JHD	DL	MD	APD	HC	PX	AFS	CDB
ABTTLO	18	15	12	11	16	46	45	7
DIT	11	4	1	0	0	14	8	1
NTT	48	0	0	0	35	0	0	0
ABTF	0.30	0.26	0.11	0.00	0.29	0.04	0.36	0.02
ABTTL	0.79	0.48	0.14	0.00	0.71	0.16	0.13	0.13
ABHBT	0.97	0.93	0.68	0.92	0.84	0.80	0.72	0.74
LFO	0.21	0.66	0.61	0.20	n/a	0.24	n/a	0.72
HR	3.39	3.93	3.83	19.4	n/a	2.80	n/a	4.50

Average Objects of the Same Class (AOSC) ranges between 1.12 and 6.39, while the maximum value ranges between 2 and 38, where JHD, DL, and PX appears to be outliers. For a better insight, we inspected the OOGs for the maximum values. For example, 34 objects of type `Vector<Figure>` in DL, and 17 objects of type `HashTable<String,String>` in PX. The values are lower in AFS, and MD. Although one may argue that most of these instances represent collections, we found an interesting example in AFS OOG that shows 4 objects of type `FileInputStream`. We argue that it is important to reason about different instances since they may play different roles at runtime. Indeed, we traced to code the objects of type `FileInputStream` in AFS OOG, and found that one object refers to a user configuration file, while another one refers to data transferred from the server to the client.

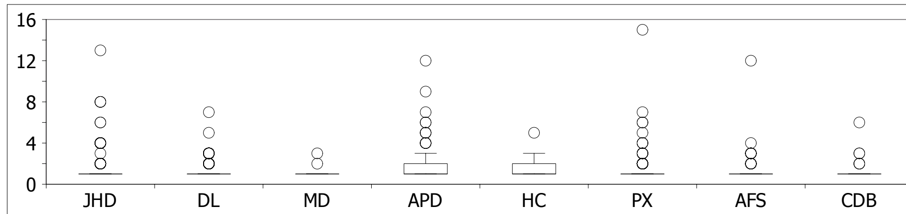


Fig. 18: Average Trace to Code per Object (ATCO).

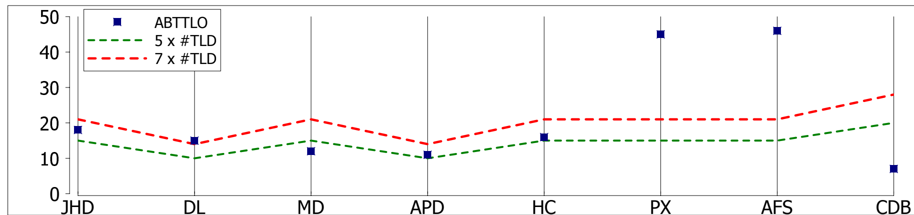
Abstraction by Types

In systems such as JHD where Average In-Degree per Top-Level Domain (IDTLD) is high, the OOG can be cluttered if there are many related subtypes. The metrics in this section are computed on the display graph, and measure the effectiveness of abstraction by types at reducing the clutter (Table 6).

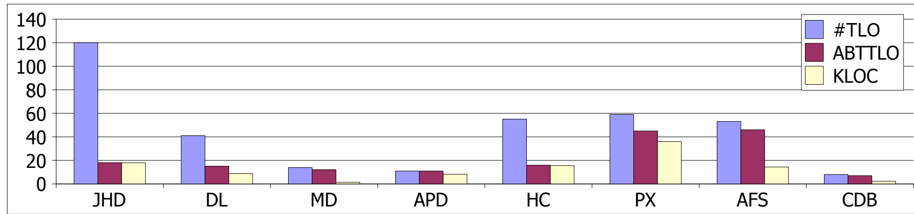
Number of Top-Level Objects after Abstraction by Types (ABT-TLO) ranges between 7 and 46. For additional insights, we compare these values to an ideal range 5–7 objects per tier (Fig. 19a). The result of the comparison indicates that the JHD, DL, MD, APD, and HC OOGs are more refined than

PX and AFS. On the other hand, the CDB OOG might be too abstract with one or two objects in a top-level domain.

We also plot ABTTLO vs. #TLO and vs. the size of the systems (KLOC). The result of the comparison indicates a weak support for our hypothesis that the size of the top-level OOG does not increase linearly with the size of the program (Fig. 19b). There is no perfect correlation between KLOC and ABTTLO, but the value of the Pearson's coefficient is at the low-end of the near perfect range (0.74). In fact, the value of Pearson's coefficient increases after abstraction by types from 0.58 to 0.74.



(a) ABTTLO vs. an ideal range.



(b) ABTTLO vs. #TLO vs. KLOC

Fig. 19: Number of Top-Level Objects after Abstraction by Types (ABTTLO).

Number of Design Intent Types (DIT) ranges between 0 and 14, which means that the list is not very long. Furthermore, for some systems (JHD, DL) all the types in the DIT list are from a few packages that contain the core framework interfaces. For JHD, the list is much smaller than the total number of types (classes and interfaces) in the system (11 vs. 257).

Number of Trivial Types (NTT), when used, is much larger. The default list includes only four types. For the two systems where it was used (JHD, HC), we had to manually pick 35–50 types. For JHD, abstraction by design intent types is more effective than by trivial types [2, Section 4.6.2].

Abstraction by Types (ABTF) ranges between 0 and 0.36 and indicates a higher effectiveness of abstraction by types for systems with a rich inheritance hierarchy. For JHD, DL, HC and AFS, abstraction by types hides approximately one quarter of all the objects in the OOG. Abstraction by types is less effective for systems that favor composition over inheritance (MD, APD).

Abstraction by Types at Top-Level (ABTTL) ranges between 0 and 0.77, and indicates a higher effectiveness for programs with a rich inheritance hierarchy, when the objects in the top-level domains are of related types. It indicates that abstraction by types merges more than 60% of the objects in the top-level domains for JHD and HC.

Abstraction by Ownership Hierarchy and by Types (ABHBT) ranges between 0.68 and 0.97, and indicates a high effectiveness for all the systems. For three of the systems (JHD, DL, and APD), the combined abstraction mechanisms decrease the number of objects in the top-level domains by one order of magnitude. The higher reduction occurs in JHD, where after abstraction by types, the reduction increases from 0.77 to 0.97 (Fig. 20).

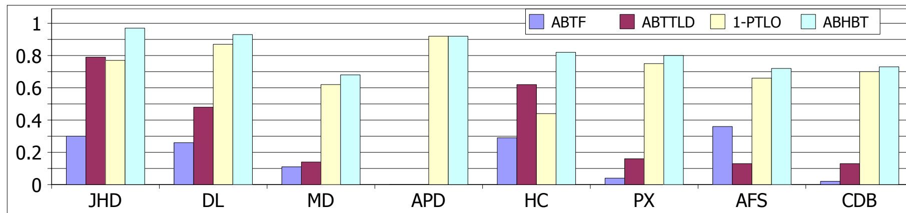


Fig. 20: Abstraction by Ownership Hierarchy and by Types (ABHBT).

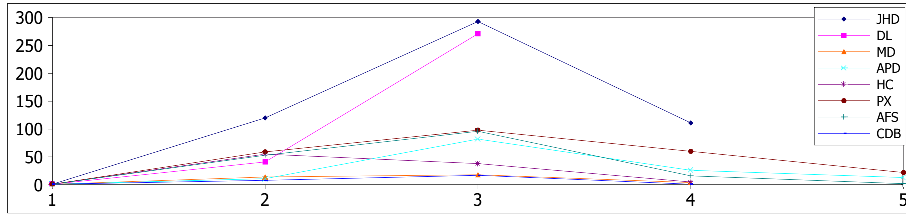
In order to capture the reduction more precisely, we count the number of objects at each level of the OOG for each system before and after abstraction by types. We observe that level 3 has the highest number of objects. Similarly to the case of median depth, the exception is HC, for which level 2 has the highest number of objects (Fig. 21a). After using abstraction by types, in most cases, the number of top-level objects (level 2) decreased without a change on the next levels. So abstraction by types is more effective for top-level objects. Abstraction by types changes the trend for HC, so that level 3 has the highest number of objects (Fig. 21b). For the AFS OOG, abstraction by types is more effective on level 3, where the number of objects is reduced by half. Indeed, abstraction by types collapses 43 objects in the public domain `COMMAND` and 6 objects in the public domain `SITE`.

Flat Object Graphs[3]

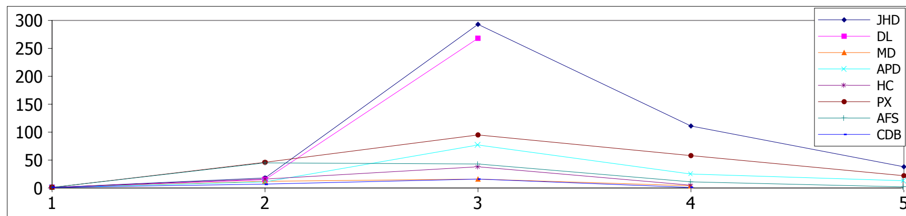
Percentage of Low-Level Objects in Flat Graph (LFO) is about 60% for all the systems, indicating a high proportion of low-level objects in flat graphs (Table 6). Since low-level objects are not architecturally-relevant, the numbers confirm that flat object graphs do not convey much architectural abstraction.

Hierarchical Reduction (HR) ranges between 2.80 and 19.45. The numbers support our hypothesis that an OOG has fewer objects at the top level,

[3] For a few systems (HC, AFS), the Womble analysis did not converge for unknown reasons, so we reported those numbers as Not Available (n/a).



(a) Before abstraction by types.



(b) After abstraction by types.

Fig. 21: Number of objects at each level of the OOG.

compared to a flat object graph. The OOG might have more objects than a flat graph (JHD, DL, PX), but the number of architecturally relevant objects (top-level objects) is smaller than the number of objects in a flat graph in all the systems. For example, the DL OOG has 5 times more objects than the DL flat graph. After using abstraction by ownership hierarchy and by types the number of top-level objects is 4 times smaller than the number of objects in the flat graph (Fig. 22). For APD, the number of top-level objects is one order of magnitude smaller than the number of objects in the corresponding flat graph.

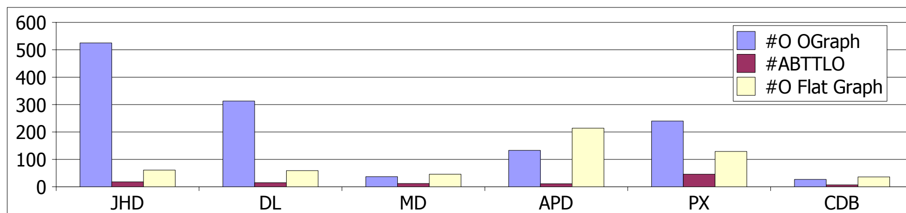


Fig. 22: Number of objects in OOG vs. number of objects in flat graph.

4.3 OOG Metrics (within the same system, across refinements)

After analyzing the OOG metrics across all the subject systems, we studied the evolution of the OOG metrics across two main refinement iterations for three

Table 7: Major milestones for selected systems.

System	Milestone	Notes
DrawLets	DL_PRE	Finished annotations; before refinement
DrawLets	DL	After refinements requested during case study [4]
MiniDraw	MD_PRE	Finished annotations; before refinement
MiniDraw	MD	After refinements requested during pilot [12, Chap. 3]
PathwayExpress	PX_PRE	Finished annotations; before meeting with maintainer [8]
PathwayExpress	PX	After incorporating refinements requested by maintainer

of the systems. Between the iterations, the only things that changed are the annotations and possibly the list of design intent types and trivial types. As a result, only the OOGs changed, so we no longer compare against flat graphs.

Procedure. We used our version control history to check-out a snapshot of the code with the annotations in place at the major milestones. We re-extracted OOGs from the earlier versions of the annotated code, and studied how the OOG metrics evolved between the two iterations (Table 7). The OOG extraction tool also stores the list of design intent types and trivial types in separate files that we checked in, so this enabled us to study the evolution of abstraction by types. The refinements involved two actors: the extractor and the maintainer for PX, and the extractor and the developer for DL and MD. For each system, at the first milestone, the extractor finished the first iteration of the annotations, and ensured that the annotations at least typechecked, with no major warnings left. Prior to the refinement iteration, however, the extractor used abstraction by types only lightly, e.g., the list of design intent types had only one type compared to the final list, which had 8 types. During refinement, the extractor refined both the annotations and used abstraction by types to make the OOG convey more design intent as the developer or the maintainer requested.

Results. We now show and discuss the evolution of the OOG metrics (Table 8).

Abstraction by Ownership Hierarchy. For all systems, Number of Objects (#O) and Number of Private Domains (#PrD) slightly increased due to a refinement that moved objects from `shared` to an existing domain, or to a new private domains, as the maintainer requested.

In PX, the Percentage of Top-Level Objects (PTLO) decreased by 7%, while Maximum Depth of Ownership Hierarchy (MXD) slightly increased. Since the maintainer made 40 requests to abstract low-level objects, they were pushed to lower-level domains. Also, no additional objects were moved up to a top-level domain, since the number of objects in the top-level domains was already high [27, Section 4.3]. In DL and MD, Number of Low-Level Objects in Top-Level Domains (#LLOTLD) became 0, and supports our hypothesis that low-level objects need to be pushed to lower levels of the OOG. In MD, since the OOG was too abstract, the extractor reduced abstraction by ownership hierarchy, and PTLO slightly increased.

Table 8: OOG metrics before and after refinement.

	DL_PRE	DL	MD_PRE	MD	PX_PRE	PX
#O	270	313	42	37	222	240
#TLO	40	41	15	14	72	59
#D	37	40	10	14	73	80
#PD	0	1	1	5	2	2
#PrD	35	37	6	6	68	75
#PtE	246	334	62	63	269	306
PTLO	0.15	0.13	0.36	0.38	0.32	0.25
#LLOTLD	2.00	0.00	1.00	0.00	0.00	0.00
IDTLD	1.60	3.73	1.93	1.86	0.22	1.05
MAX	5	7	12	9	2	6
AOD	7.27	7.80	4.10	2.57	3.03	2.99
MAX	32	34	7	9	30	27
ABTTLO	18	15	14	12	68	46
DIT	1	4	1	1	1	8
ABTF	0.12	0.26	0.01	0.11	0.00	0.04
ABTTL	0.40	0.48	0.07	0.14	0.00	0.16
ABHBT	0.93	0.93	0.83	0.68	0.68	0.80

In MD, the Number of Public Domains (#PD) increased from 1 to 5 in response to a request to push objects that were at the top-level underneath more architecturally-relevant ones. This indicates that public domains are used during the refinement of an OOG.

In all three systems, Number of Points-to Edges (#PtE), Cluttering (COOG), and Average In-Degree per Top-Level Domain (IDTLD) remained relatively low, but slightly increased. One exception in MD, where the maximum IDTLD decreased from 12 to 9 for the objects of type `BoardFigure`, and `BoardDrawing` in the top-level domain `MODEL`. Also, Average Objects per Domain (AOD) slightly increased for DL and decreased for MD and PX with a maximum of 34 for DL.

Abstraction by Types. For PX, the maintainer requested that extractor collapse 20 related objects based on their types [8, Table 2]. The Number of Design Intent Types (DIT) increased from 1 to 8, and allowed the extractor to hide 4% of the objects and 16% of the top-level objects. For DL, in response to a refinement request, the extractor added 3 classes to the list of DIT. Abstraction by Types (ABTF) increased from 12% to 26%, while the Number of Top-Level Objects after Abstraction by Types (ABTTLO) increases from 40% to 48%.

Abstraction by Ownership Hierarchy and by Types (ABHBT) increased for PX, decreased for MD, and remained constant for DL. For MD, the developer suggested that the OOG was too abstract. After the refinement, Number of Objects (#O) decreased from 42 to 37, ABHBT decreased from 0.83 to 0.68, while PTLO increased from 0.36 to 0.38. All these values indicate a less abstract graph. For PX, the maintainer requested additional abstraction. After using abstraction by ownership hierarchy and by types combined, ABHBT increased from 0.68 to 0.80. For DL, where abstraction by ownership hierarchy

was more effective in the initial OOG, the extractor relied more on abstraction by types. Since the number of objects in the OOG increased, ABHBT remained constant at 0.93, a value close to the ideal value of five to seven objects per top-level domain. We measured the distances between the ABHBT value and the ideal range before and after refinements. For DL the distance slightly increased from 0.01 to 0.02. For MD and PX the distance decreased from 0.19 to 0.09, and 0.23 to 0.11, respectively. Therefore, for MD and PX, ABHBT got closer to the ideal values (Fig. 23). The extractor consider these ideal values as guidance,

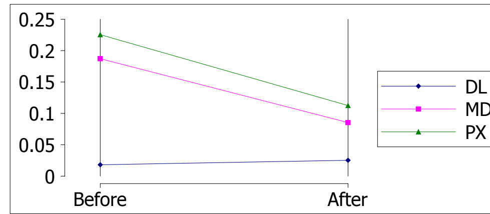


Fig. 23: ABHBT variation after refinement, normalized to an ideal interval.

rather than rigid targets. There is an ideal level of abstraction targeted while refining the OOG, and this level is difficult to express by simply getting the highest value for one metric. Hypothetically, an OOG with Number of Objects (#O) as 100 and ABHBT as 0.99 would have a single top-level object, and would be too abstract. In fact, for MD, ABHBT decreases from 0.83 to 0.68, while the ideal values range between 0.5 and 0.64. This ideal range becomes very narrow for large systems, and it might be unrealistic to treat it as a rigid value. Getting the ideal value for ABHBT may be insufficient, as indicated by the DL OOG that has ABHBT as 0.93, which is very close to the ideal range of 0.94–0.96. However, the developer requested additional refinements.

Another solution similar to Pareto's Principle (80-20% rule) is to target ABHBT to be higher than a fixed relative percentage. Based on Pareto's Principle, the extractor may assumes that 20–30% of the objects are architecturally relevant. The solution would apply for PX, where ABHBT increases from 68% to 83% after the meeting with the maintainer. Simply aiming for a fixed percentage might lead to insufficient abstraction for large OOGs. It is the case of JHD OOG where the current ABHBT is 97%. Following this rule, it would also be difficult to justify the decrease of ABHBT for MD from 83% to 68%, and the fact that DL needs additional refinement when ABHBT is 93%.

5 Discussion

We discuss our observations, mention some limitations and threats to validity.

5.1 Observations

Our metrics attached quantitative data to our research questions.

Object graphs partially follow strict encapsulation. In practice, we made use of private domains for 2–17% of all fields and variables of a reference type (excluding method return values).

Object graphs partially follow logical containment. In practice, we made use of public domains for 0.3–7.4% of all fields and variables of a reference type (excluding method return values). In particular, systems that had carefully refined OOGs used public domains more heavily. During refinement, developers make the OOG reflect their mental model of the system by making an object part of another object.

In fact, for some systems, logical containment (with public domains) was used more often than strict encapsulation (with private domains). For example, in both MD and AFS, there were more fields annotated with a public domain than were annotated with a private domain. Our data suggests that for these programs, and possibly for others, if we pushed harder to create hierarchical OOGs, ownership type systems that only provide strict encapsulation will produce hierarchies that are substantially flatter. For expressing design, at least, logical containment appears to be quite important for some systems.

We observed that the percentage of objects in public domains was in general greater than the percentage of fields and variables of a reference type in a public domain (Fig. 24). That means, for example, that multiple objects in public domains correspond to one declaration of a field in a public domain. This observation supports the case that public domains are important for making graphs more hierarchical. The most interesting cases are MD, AFS, and CDB with 20% or more objects in public domains.

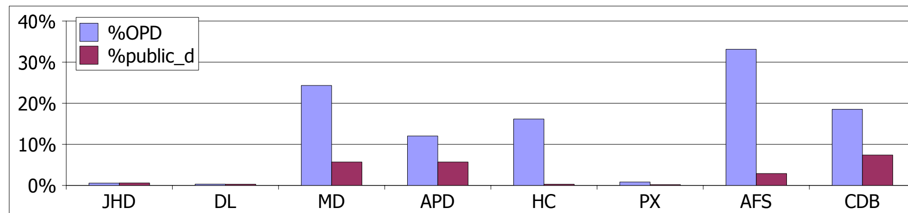


Fig. 24: Percentage of objects in public domains vs. the percentage of variables and fields in a public domain.

Abstraction by ownership hierarchy is effective in object graphs. Our results show a reduction in the number of objects compared to a flat object graph, but the reduction was smaller than an order of magnitude. There is a caveat, however, in that the Womble flat object graphs we compared against were potentially unsound. Sound flat object graph may actually show more objects

and the reduction would be higher. Further, abstraction by ownership hierarchy alone is insufficient, in that we still end up with many objects per domain in some cases.

In principle, abstraction by ownership hierarchy can be applied until the OOG is clutter-free, but at some point, it will become a less natural expression of design intent. For example, JHD has many sub-classes that are related, such as the subclasses of `Figure`, `Command` and `Tool`. Ideally, developers want the instances of these types to be all peers in the ownership hierarchy. It would be less convincing to make an instance of one subclass of `Figure` (e.g., `EllipseFigure`) be part of another instance of another subclass of `Figure` (e.g., `RectangleFigure`), just to reduce the number of objects in a domain.

Abstraction by types is needed in object graphs. Our results indicate that abstraction by types is needed to complement abstraction by ownership hierarchy. If abstraction by ownership hierarchy produces depth compression in an object graph, abstraction by types produces width compression, by collapsing related instances that are in the same domain. Furthermore, the technique was generally applicable across several systems.

Abstraction by ownership hierarchy and by types is effective in object graphs. Together, both mechanisms reduce the number of top-level objects in an OOG compared to flat object graphs.

Limits of abstraction by ownership hierarchy and by types. There were a few cases where, despite our efforts, the final OOGs were still too cluttered. For example, for PX, we wanted to use abstraction by types more heavily, but were limited by the system design. For PX, the design issues include the lack of a rich inheritance hierarchy, many loosely typed containers (for many containers, the most precise generic types that can be added are `Object` or `Serializable`), and the lack of user-defined types (many fields are `Strings`) [8, Sections 5.3 and 5.4].

Refining the PX OOG further would have required re-engineering the code, sometimes in simple ways. For example, we could have defined an empty interface, made related classes implement that interface (this is the "marker interface" pattern[4]), then added that interface to the list of design intent types to further control the abstraction by types. Other re-engineering would be more complex.

Other possible abstraction mechanisms. The challenges in achieving an uncluttered graph with PX suggest that additional abstraction mechanisms, beyond ownership hierarchy and types, might be necessary, especially as OOGs are applied to larger and larger systems.

One idea, for example, is to allow "abstraction by name". During the refinement of the PX OOG, the maintainer wanted to merge several objects that have `Table` in their declared types, like `pdTable: PathwayDetailsTable`, `ipIdGenesTable: InputIdGenesTable` and `model: PETableModel`. Even the maintainer was surprised that the types of these objects do not share a common super-type, other than `Object`. A similar situation occurred in AFS, where the only super-type shared by the interfaces `AuthorizationRequest` and

[4] Item 37: Use marker interfaces to define types [16].

`FtpRequest` is `Object`. As a result, we could not use abstraction by types to accomplish this merging. If the OOG were to allow "abstraction by name", such a merging would be possible.

More generally, after applying abstraction by ownership hierarchy and types, developers might specify groupings among the remaining objects by their conceptual purpose, even if they are not related by type or by name. This is similar to the manual specification of task-specific abstractions in approaches such as Reflexion Models [37].

The abstraction mechanisms provided by ownership could be quite complementary to the mechanisms provided in Reflexion Models. While task-specific abstractions allow the developer to focus on parts of the system relevant to a particular task, ownership provides a basic form of abstraction that can be customized in a task-specific way. Furthermore, ownership helps developers focus on the important parts of the system quickly, which may facilitate task-specific modeling. Ownership also provides information about logical containment and strict encapsulation that is missing in alternative abstraction mechanisms and also in plain Java code.

5.2 Limitations

Limitations of type system. Most of the systems we studied still have a number of annotation warnings. As a result, their OOGs may be unsound.

Limitations of metrics. Metrics alone do not determine if an OOG is good. Ultimately, a human must decide if the OOG is good enough. One way to evaluate an OOG, like any architectural diagram, is to compare it against a target. This is an approach we followed previously, by abstracting an OOG into a runtime architecture and analyzing conformance [3]. Another way is to give the OOG to developers making code modification tasks, and evaluate if it helps during program comprehension [4].

Unsound flat graphs. One big limitation of comparing OOGs to flat graphs is that we used an unsound flat object graph analysis, Womble [30]. Ideally, we should have used a sound one, such as Ajax [40]. To our knowledge, we could have chosen one of the following three candidates: Womble [30], Ajax [40] or Pangaea [46].

Womble [30] starts with a class diagram and uses heuristics for container classes and multiplicities to refine the object model. The follow-on tool, SuperWomble [50], uses additional heuristics for merging types but does not attempt to be sound. The unsoundness is an engineering tradeoff that is claimed to produce correct object models in practice, by masking problems due to other weaknesses of the analysis (namely, that it is flow-insensitive). SuperWomble also uses built-in and user-defined abstraction rules for containers that coalesce a chain of edges in the object model into a single edge [50]. SuperWomble also analyzes all classes that are transitively referenced (through constructor calls, field references, etc.) from the root set of classes. To avoid analyzing a large

number of classes, most of which would not affect the output, a *stop-analysis configuration file* controls what classes or packages the tool analyzes [51].

Ajax[5][40] uses a sound alias analysis to build a refined object model as a conservative static approximation of the heap graph reachable from a given set of root objects. However, Ajax does not use ownership and produces flat object graphs. Ajax relies heavily on post-processing raw object graphs, such as by eliding all "lumps" with more than seven incoming edges or eliding all subclasses of a given type, e.g., `InputStream` (p. 248). Moreover, the object models that Ajax generates tend to expose internal implementation details (p. 252). OOGs do not typically suffer from this problem since the annotations typically store an object's internal implementation details in private domains. On the other hand, Ajax is able to detect fields that are actually unused. In addition, Ajax can automatically and soundly split classes in the object model, i.e., determine that an object is indeed of type `Y` and not of type `Z`—even if `Z` is a subclass of `Y`, and without any information other than the code. Finally, Ajax's heavyweight but precise alias analysis does not scale to large programs.

Pangaea [46] produces a flat object graph without performing an alias analysis and is unsound. For several programs, the Pangaea output is even more complex than that of Womble. To compare the Womble and Pangaea flat graphs on JHotDraw, see [2, Figures 4.18 and 4.19].

Another issue when comparing OOGs to flat object graphs is how to deal with library code. In principle, a flat graph can pull in many objects from all the libraries that are in use, and grow arbitrarily large. In an OOG, objects from library code would be considered low-level, and be pushed underneath more architecturally-relevant objects. To avoid an overwhelming number of objects in flat object graphs, we used the Womble feature of a stop-analysis configuration file. Womble used the same configuration for all the systems, and included the packages `java.*`, `com.sun.java.*`, as well as packages from system-specific, third-party libraries.

5.3 Threats to Validity

Threats to Internal Validity. Our study may include several threats to internal validity. First, the developers adding ownership annotations had varying experience with object-oriented design and with ownership types. Some of the less experienced developers initially added low-quality annotations. For example, they made too many fields as `shared` or used heavily the `lent` annotation, just to quickly reduce the number of annotation warnings. However, in all cases, the typechecker kept the developers on track, by ensuring that the annotations were consistent with each other and with the code. Also, by computing the annotation metrics above, we can better understand their quality.

[5] Ajax [40] is not publicly available. O'Callahan was kind enough to give us the sources for his tool. But we were unable to run Ajax successfully, even on trivial examples because it requires an obsolete environment (JDK 1.1), and has undocumented dependencies.

Second, we refined some of the OOGs more than others. For example, we carefully refined the JHotDraw, DrawLets and MiniDraw OOGs since they were going to be used by other developers during coding tasks. Other OOGs, such as the ones for Aphyds and CryptoDB, were compared against reference architectures. But some of remaining OOGs were not fully evaluated.

Threats to External Validity. Several factors could be affecting the generalization our findings. First, the subject systems we selected may not be representative of all object-oriented code bases. Some of the systems (HC and PX) were developed by students rather than professional programmers. Others (JHD, DL, and MD) were designed by object-oriented experts for illustrative or pedagogical purposes, and as a result, may have been over-engineered to use many of the standard object-oriented design patterns. We mitigated this threat in some of the later subject systems by avoiding extremes, such as systems developed by undergraduates (HC) or by object-oriented experts (JHD). Instead, we picked an open-source application that was developed by professional programmers and that had been evolving (AFS).

Second, the number of subject systems in our study may be lower than is typically seen in empirical studies of the code structure [15] or studies of runtime heaps using dynamic analysis [42]. Those studies consist of running a fully automated analysis on a large number of systems and comparing the results across the systems. In our study, we had to manually add annotations to each subject system before we could analyze it.

The effort to add annotations that typecheck and to extract initial OOGs is measured to be around 1 hr/KLOC in a carefully timed setting, on the PX system [8]. Additional, but lesser effort, is required to refine the OOG. For PX, we spent 3 hours after meeting with the maintainer, compared to the 31 hours we spent adding annotations and extracting initial OOGs. The meeting with the maintainer to get the list of refinements lasted a little over 1 hour.

Our subject systems total over 100 KLOC in size, so we estimate the annotation effort to be at least 100 hours, and likely much higher. It is possible that some of the systems that were annotated earlier while the tools were still immature took much longer to annotate. It is also possible that some of the OOGs (JHD, HC, APD, CDB) took more effort to refine because the OOG was compared against a target architecture, or evaluated by other developers during code modification tasks (JHD, DL, MD). To minimize outliers, we could have excluded the systems for which the OOG was not sufficiently refined.

Finally, our subject systems are relatively small- to medium-sized. So, it is unclear if our results will hold on systems with hundreds of thousands of lines of code or larger. The scale of the systems we analyzed may pale in comparison to studies that analyzed the code structure of large systems [15], because our type-based technique requires developers to manually specify architectural intent using annotations. Without automated inference of ownership types, analyzing large systems is prohibitively costly. Since tools for analyzing the runtime structure statically are immature compared to tools for analyzing the code structure, the scale of the systems we analyzed is not uncommon. For example, related

prior work that used annotations to statically extract object models [34] was evaluated on one 1700-line system.

6 Related Work

We organize the related work into metrics on the runtime structure and metrics on the code structure. We also discuss case studies of ownership type systems.

Runtime Structure. Potanin et al. studied aliasing in object-oriented Java-like programs by analyzing heap snapshots [42]. They measured uniqueness, ownership and confinement. To calculate the metrics, they analyzed a corpus of 58 heap snapshots from around 35 programs of different sizes.

For each object graph snapshot, they calculated the general metrics, the number of objects in the object graph (NO), the number of unique roots in the Java heap root set (NUR) and the number of objects accessible by reference traversal (ART) from the root set. For uniqueness, they calculated the percentage of aliased objects (PAL), where the aliased object is an object with more than one incoming reference. For object ownership, they calculated the metrics for the average depth (AD) of an object in the ownership tree and the average depth after folding (ADF). For confinement, they calculated the number of not confined (NC) objects if there are referrers from a different package, weakly confined (WC) if all referrers are in the same top-level package, strongly confined (SC) if the classes of all the objects which refer to an object of interest are in the same package as that object's class, and the total number of classes (TNC). The study shows that object-oriented programs exhibit evidences of encapsulation in practice, and models of uniqueness, ownership, and confinement can describe the aliasing structures of object-oriented programs.

Our work differs from Potanin et al.'s in that we used only static analysis. Static information is less precise than dynamic information. In particular, our statically extracted and sound OOGs are necessarily more abstract than the runtime heaps that a dynamic analysis can monitor.

Code Structure. Baxter et al. analyzed a number of Java programs to determine whether relationships between different artifacts follow a power law distribution [15]. The paper points out that for small datasets, the results are inconclusive and contradict other studies that indicated that power laws were quite common in software [44, 48]. In our work, we have small datasets, and as a result, we did not analyze whether the relationships obey a power law distribution.

Baxter et al. computed metrics on the static code structure, such as number of methods (nM), number of fields (nF) and Return as Client (RC). Our focus is on the runtime structure, but some code structure metrics are related to our annotation metrics. For example, Subclasses as Provider (SP) counts the number of classes extending a given class, and Interfaces as Provider (IP) counts the number of classes implementing a given interface. In Ownership Domains, domain parameters declared on a sub-class must be related to the domain parameters declared on a super-class, and some the annotation metrics reflect that (e.g., "inherits" in Table 3).

We also computed metrics of the code structure such as the Depth of the Inheritance Tree, defined elsewhere [19]. Information about inheritance is relevant since the abstraction by types mechanism in an OOG uses the sub-typing relations in the system.

Case Studies of Ownership Type Systems

Many ownership type systems have been tested only to check if they can express the canonical iterator example [17]. Others have applied ownership types to code that implement one of the standard design patterns in isolation [38]. However, many expressiveness challenges arise in real object-oriented code, and when the same objects are involved in several design patterns at once. In addition, there are multiple ways to implement a standard design pattern in object-oriented code.

Hächler documented a case study in applying the Universes type system [36, 23] on an industrial software application and refactoring the code in the process [26]. Although the subject system in the case study was relative large (around 55,000 lines of code), Hächler annotated only a portion of the system, and did not report the exact number of lines of code he annotated. Hächler also manually generated visualizations of the ownership structure. In contrast, during our case studies, we used a tool to extract OOGs and to visualize the ownership structure, then refined the annotations accordingly.

Nägeli evaluated how the Universes and ownership domain type systems express the standard object-oriented design patterns [38]. However, in real world complex object-oriented code, design patterns rarely occur in isolation [45]. Our case studies indicated that it is often these subtle interactions, combined with the type system constraints, that can make adding the annotations difficult in some cases.

Case studies that retro-fit ownership types into legacy code often do not judge the quality of the added ownership types. Even if the ownership types typecheck, they may not be "good" or precise enough. For example, in Ownership Domains, one can annotate every reference as `shared`, which enables little reasoning. In this paper, we extracted OOGs based on the ownership types, and computed metrics which help us measure the quality of the inserted annotations. One could use the metrics to flag a large percentage of `shared` annotations for example.

7 Conclusion

Over the course of several years, we added ownership domain annotations to over 100 KLOC of real object-oriented code from a range of systems, and extracted OOGs. In this paper, we computed metrics on the annotations and on the OOGs. The annotation metrics helped measure the quality of the annotations. The OOG metrics helped explain the shape of object graphs in object-oriented code.

Our results demonstrate that ownership can make substantial contributions to expressing designs more abstractly. The top-level size of extracted ownership diagrams seems to scale sub-linearly with system size, suggesting that ownership will provide increasing benefits as it is applied to larger systems. We found

evidence that both strict encapsulation mechanisms and logical containment mechanisms were important in expressing the designs of some programs. We also found evidence that the design characteristics of some programs mean they benefit from ownership but still need additional abstraction mechanisms in order to produce succinct diagrams of the object structures, suggesting there are fruitful directions for continuing work in the area.

Acknowledgements. The authors thank Jonathan Aldrich for detailed and insightful comments on an earlier version of this work. The authors also thank several contributors (Nariman Ammar, Fayez Khazalah, Zeyad Hailat, and Anwar Mohammadi) who, as course projects or for independent study, added annotations to and extracted OOGs from several object-oriented systems (DL, MD, PX, AFS) [14, 13, 5, 49]. Vanciu was supported by Wayne State University. Abi-Antoun was supported by his faculty startup fund at Wayne State University Finally, the authors thank the anonymous reviewers for their helpful feedback.

References

1. Apache FtpServer 1.0.5. `http://mina.apache.org/ftpserver/`.
2. Marwan Abi-Antoun. *Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure*. PhD thesis, Carnegie Mellon University, 2010. Available as Technical Report CMU-ISR-10-114.
3. Marwan Abi-Antoun and Jonathan Aldrich. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *OOPSLA*, 2009.
4. Marwan Abi-Antoun and Nariman Ammar. A Case Study in Evaluating the Usefulness of the Run-time Structure during Coding Tasks. In *Workshop on Human Aspects of Software Engineering (HAoSE)*, 2010.
5. Marwan Abi-Antoun, Nariman Ammar, and Zeyad Hailat. Extraction of ownership object graphs from object-oriented code: an experience report. In *ACM Sigsoft Conference on the Quality of Software Architectures (QoSA)*, 2012.
6. Marwan Abi-Antoun, Nariman Ammar, and Thomas LaToza. Questions about Object Structure during Coding Activities. In *Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, 2010.
7. Marwan Abi-Antoun and Jeffrey M. Barnes. Analyzing Security Architectures. In *ASE*, pages 3–12, 2010.
8. Marwan Abi-Antoun and Zeyad Hailat. A Case Study in Extracting the Runtime Architecture of an Object-Oriented System. Technical report, Wayne State University, 2011.
9. Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.
10. Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *ICSE*, 2002.
11. Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, 2002.
12. Nariman Ammar. Evaluation of the Usefulness of Diagrams of the Run-Time Structure for Coding Activities. Master's thesis, Wayne State University, 2011. Available: `www.cs.wayne.edu/~mabianto/`.

13. Nariman Ammar and Marwan Abi-Antoun. Adding Ownership Domain Annotations to and Extracting Ownership Object Graphs from MiniDraw. Technical report, Wayne State University, 2011. www.cs.wayne.edu/~mabianto/oog_data/.
14. Nariman Ammar, Fayez Khazalah, and Marwan Abi-Antoun. A Case Study in Adding Ownership Domain Annotations. Technical report, Wayne State University, 2010. www.cs.wayne.edu/~mabianto/oog_data/.
15. Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the Shape of Java Software. In *OOPSLA*, 2006.
16. Josh Bloch. *Effective Java*. Addison-Wesley, 2001.
17. Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership Types for Object Encapsulation. In *POPL*, 2003.
18. Nicholas Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. Multiple Ownership. In *OOPSLA*, 2007.
19. Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *OOPSLA*, 1991.
20. Henrik B. Christensen. *Flexible, Reliable Software Using Patterns and Agile Development*. Chapman and Hall/CRC, 2010.
21. David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, 1998.
22. David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety. In *VAMP*, pages 20–51, 2007.
23. Werner Dietl and Peter Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8), 2005.
24. DrawLets. www.rolemodelsoft.com/drawlets/, 2002. Version 2.0.
25. Eclipse Metrics Plugin. http://metrics.sourceforge.net/, 2010.
26. Thomas Hächler. Applying the Universe Type System to an Industrial Application: Case Study. Master's thesis, Department of Computer Science, Federal Institute of Technology Zurich, 2005.
27. Zeyad Hailat and Marwan Abi-Antoun. P-X Case Study: Online Appendix. http://www.cs.wayne.edu/~mabianto/px/, 2011.
28. Scott Hauck. Aphyds: The academic physical design skeleton. In *International Conference on Microelectronics Systems Education*, pages 8–9, 2003.
29. Intelligent Systems and Bioinformatics Laboratory (ISBL). http://vortex.cs.wayne.edu/projects.htm/, 2003.
30. Daniel Jackson and Allison Waingold. Lightweight Extraction of Object Models from Bytecode. *TSE*, 27(2), 2001.
31. JHotDraw. www.jhotdraw.org, 1996. Version 5.3.
32. Kevin Kenan. *Cryptography in the Database*. Addison-Wesley, 2006. Accompanying code at http://kevinkenan.blogs.com/downloads/cryptodb_code.zip.
33. Henk Koning, Claire Dormann, and Hans van Vliet. Practical Guidelines for the Readability of IT-Architecture Diagrams. In *SIGDOC*, 2002.
34. Patrick Lam and Martin Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP*, 2003.
35. MiniDraw. www.baerbak.com/src/frs-src.zip.
36. Peter Müller and Arnd 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. Gail Murphy, David Notkin, and Kevin J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *TSE*, 27(4), 2001.

38. Stefan Nägeli. Ownership in Design Patterns. Master's thesis, Department of Computer Science, Federal Institute of Technology Zurich, 2006.
39. James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In *ECOOP*, 1998.
40. Robert W. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie Mellon University, 2001.
41. David Poole and Alan Macworth. CISpace: Tools for learning Computational Intelligence. `http://www.cs.ubc.ca/labs/lci/CIspace/`, 2001.
42. Alex Potanin, James Noble, and Robert Biddle. Checking Ownership and Confinement. *Concurrency and Computation: Practice and Experience*, 16(7), April 2004.
43. Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic Ownership for Generic Java. In *OOPSLA*, 2006.
44. Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale-Free Geometry In OO Programs. *Commun. ACM*, 48(5), 2005.
45. Dirk Riehle. *Framework Design: a Role Modeling Approach*. PhD thesis, Federal Institute of Technology Zurich, 2000.
46. André Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, FU Berlin, 2002.
47. Margaret-Anne D. Storey, Hausi A. Müller, and Kenny Wong. Manipulating and Documenting Software Structures. In P. Eades and K. Zhang, editors, *Software Visualization*, 1998.
48. S. Valverde, R. Ferrer-Cancho, and R. V. Sole. Scale-free networks from optimal design. *Europhysics Letters*, 60(4), 2002.
49. Radu Vanciu and Marwan Abi-Antoun. Adding Ownership Domain Annotations and Extracting OOGs from Apache FTP Server. Technical report, Wayne State University, 2011. `www.cs.wayne.edu/~mabianto/oog_data/`.
50. Allison Waingold. Automatic Extraction of Abstract Object Models. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, 2001.
51. Allison Waingold and Robert Lee. SuperWomble Manual. `http://sdg.lcs.mit.edu/womble/`, 2002.