Are Object Graphs Extracted Using Abstract Interpretation Significantly Different from the Code? (Extended Version)[1]

Marwan Abi-Antoun **Sumukhi Chandrashekar**
Radu Vanciu **Andrew Giang**

September 2014

Department of Computer Science
Wayne State University
Detroit, MI 48202

Abstract

To evolve object-oriented code, one must understand both the code structure in terms of classes, and the runtime structure in terms of abstractions of objects that are being created and relations between those objects. To help with this understanding, static program analysis can extract heap abstractions such as object graphs. But the extracted graphs can become too large if they do not sufficiently abstract objects, or too imprecise if they abstract objects excessively to the point of being similar to a class diagram that shows one box for a class to represent all the instances of that class. One previously proposed solution uses both annotations and abstract interpretation to extract a global, hierarchical, abstract object graph that conveys both abstraction and design intent, but can still be related to the code structure. In this paper, we define metrics that relate nodes and edges in the object graph to elements in the code structure to measure how they differ, and if the differences are indicative of language or design features such as encapsulation, polymorphism and inheritance. We compute the metrics across eight systems totaling over 100 KLOC, and show a statistically significant difference between the code and the object graph. In several cases, the magnitude of this difference is large.

[1] *This technical report revises an earlier workshop paper [AVA13]. A shorter version will appear as: Abi-Antoun, M., Chandrashekar, S., Vanciu, R., and Giang, A. Are Object Graphs Extracted Using Abstract Interpretation Significantly Different from the Code? In IEEE International Working Conference on Source Code Analysis and Manipulation, 10 pages. 2014.*

This material is based upon work supported by the Maryland Procurement Office under Contract No. H98230-14-C-0140.

Keywords: corpus analysis; object-oriented metrics; points-to analysis; heap abstractions

Contents

1 Introduction

When evolving object-oriented code, one must understand both the code structure in terms of classes and inheritance relationships between them, and the runtime structure in terms of abstractions of objects that are being created and relations between those objects. To help with this understanding, many techniques extract views of the code structure such as class diagrams, as well views of the runtime structure such as object graphs. Most object graphs employ abstraction to prevent the graph from becoming too large. But the object graphs can become too imprecise if they abstract objects excessively to the point of being similar to a class diagram that shows one box for a class to represent all the instances of that class.

Previous techniques for extracting heap abstractions have used dynamic analysis [Mit06, BBM13], by analyzing a finite number of executions, or static analysis by analyzing the code only [JW01]. The results of any dynamic analysis are inherently unsound, i.e., may not reflect all possible objects and relations, since dynamic analysis considers only a finite number of executions using specific inputs and test cases. But static analysis can extract sound abstractions that approximate all possible executions, for any possible input. To organize large heaps, object ownership or hierarchy, where an object is the child of another object, is effective. However, fully automated abstractions infer only restricted patterns of hierarchy, for example, cases where a child object is dominated by its parent object and there are no incoming direct references to the child object.

To achieve soundness, one approach proposed by Abi-Antoun and Aldrich [AA09] uses abstract interpretation to approximate an abstract runtime structure in the form of a hierarchical object graph, the Ownership Object Graph (OOG). Since object hierarchy is not directly available in plain object-oriented code, the static analysis requires that ownership types be added to the code, in the form of annotations. The annotations implement an ownership type system, Ownership Domains [AC04]. Although the annotations are amenable to automated inference, the annotations used in this paper are manually added to express design intent, and are checked using a typechecker to be consistent with each other and with the code. In particular, the OOG supports a more flexible object hierarchy, that of *logical containment*, where one object can be the child of another, but is still accessible to other objects.

There is no good sense, however, of how the abstract runtime structure of a given system, represented by its OOG, differs from the code structure, or of what causes these differences to be larger in one system than in another. By code structure, we mean elements in the Abstract Syntax Tree (AST) of a system. For example, a code element can be a class declaration, a field declaration, or an expression in the abstract syntax. By abstract runtime structure, we mean specifically the OOG. In this paper, we compute metrics to relate the structure of the OOG back to the code structure.

Contributions. The contributions of this paper are as follows:

- A definition of several metrics to quantify the differences between the code structure and the abstract runtime structure (Section 4);
- The results of the metrics on over 100 KLOC from eight subject systems across applications domains, and quantitative statistical analysis of the metrics across the systems, discussed first quantiatively (Section 5) then qualitatively (Section 6);
- A qualitative analysis of the outliers to identify language features, designs or code patterns that contribute to larger differences (Section 7); and
- A brief analysis to study if the metrics identify program comprehension difficulties, on one of the systems that was previously used in a controlled experiment (Section 8).

2 Positioning: Intuition behind metrics

For example, it is interesting to find two abstract objects of the same declared type in different locations in the abstract object hierarchy, e.g., `figMap:HashMap` and `pMap:HashMap`. It is also interesting to find two abstract objects of the same declared type (`figMap:HashMap` and `propMap:HashMap`) in two sibling domains (`MAPS` and `OWNED`, respectively) that are children of the same abstract object (`bd:BDrawing`).

Hierarchy of classes

```
+- java
  +- util
    +- ArrayList
    +- HashMap
      +- Entry
+- minidraw
  +- boardgame
    +- BFigure
    +- BDrawing
  +- standard
    +- SelectionTool
    +- handlers
    | +- DragTracker
    | +- SATracker
    | +- SDCLHandler
    | +- SRBSStrategy
    | +- SSHandler
```

Hierarchy of objects

```
+- main:Breakthrough
  +- CTRL
    +- srbss:SRBSStrategy
    +- st:SelectionTool
      +- TRACKERS
      | +- dt:DragTracker
      | +- sat:SATracker
  +- MODEL
    +- bf:BFigure
      + OWNED
      | +- pMap:HashMap
    +- bd:BDrawing
      + OWNED
      | +- fgrs:ArrayList
      | +- propMap:HashMap
      +- HANDLERS
      | +- dh:SDCLHandler
      | +- sh:SSHandler
      +- MAPS
      | +- figMap:HashMap
```

Figure 1: MiniDraw: hierarchy of classes vs. hierarchy of abstract objects.

Why not compare the code structure to the true runtime structure? The main reason that the metrics relate the code structure to an abstract runtime structure—rather than the true runtime structure or dynamic heap—is that an abstract runtime structure is a *finite* representation, just like the code structure. It is also possible to measure the properties of soundly abstracted heaps [BBM13]; that is an approach that was explored independently, as we discuss next.

Measuring properties of an abstracted heap. Barr et al. [BBM13] do not directly compare the code structure to an abstracted heap, but their work is related in several ways. They also infer abstractions of runtime objects into abstract objects, and measure several properties about the abstracted heaps. The first major difference is that, while their abstraction is sound, their underlying object graphs are extracted using dynamic analysis, and thus are unsound: some objects or relations may occur only in other executions, for other inputs or for other exercised testcases, so they will not be reflected in the abstractions. Not taking into account all objects and relations is likely to skew the results. The second difference is that the abstractions are automatically inferred rather than human-guided, and as a result, they identify specific types of ownership patterns such as strict domination. In particular, their system does not have the notion of logical containment, where an object is made only conceptually part of another object based on arbitrary design intent. In fact, the previous work was unable to classify some relationships that did not follow the expected ownership patterns and called for more flexible annotation systems.

3 Background: Object Graph Semantics

A static analysis extracts the OOG using an abstract interpretation of code to which annotations have been added. The annotations are checked to be consistent with the code and implement a type system, Ownership Domains [AC04]. So we briefly review the Ownership Domains type system and the OOG semantics in order to formally define our metrics.

Abstract syntax. We show a small portion of the abstract syntax for Ownership Domains (Fig. 2), focusing on class declarations, field declarations and object creation expressions. The metavariable C ranges over class names; T ranges over types; f ranges over fields; v ranges over values; d ranges over domain names; and p ranges over formal domain parameters, actual domains, or the special domain `SHARED`. As a shorthand, an overbar is used to represent a sequence that can be possibly empty.

A class is parameterized by a list of domain parameters, and extends another class that has a subsequence of its domain parameters. A type T is a class name and a set of actual domain parameters $C<\overline{p}>$. In the

$$\begin{array}{rcl}
\mathit{cdef} & ::= & \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>} \\
 & & \{\ \overline{\mathit{dom}};\ \ \overline{T}\ \overline{f};\ \ldots\} \\
\mathit{dom} & ::= & [\texttt{public}]\ \texttt{domain}\ d; \\
e & ::= & x\ \mid \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(\overline{e})\ \mid \ldots \\
n & ::= & x\ \mid\ v \\
p & ::= & \alpha\ \mid\ n.d\ \mid\ \texttt{SHARED} \\
T & ::= & C\texttt{<}\overline{p}\texttt{>} \\
v, \ell \in & \mathit{locations} &
\end{array}$$

Figure 2: Portions of the Ownership Domains abstract syntax [AC04].

$$\begin{array}{ll}
D \in \mathsf{ODomain} & ::= \langle\ \mathbf{Id} = D_{id}, \mathbf{Domain} = C{::}d\ \rangle \\
O \in \mathsf{OObject} & ::= \langle\ \mathbf{TypeDomain} = C\texttt{<}\overline{D}\texttt{>}\rangle \\
E \in \mathsf{OEdge} & ::= \langle\ \mathbf{From} = O_{src}, \mathbf{Field} = f,\ \mathbf{To} = O_{dst}\ \rangle
\end{array}$$

Figure 3: Key data type declarations for the OGraph.

formal system, we treat the first domain parameter of a class as its owning domain.

Data types. The internal representation of an OOG is an OGraph (Fig. 3). An OGraph is a graph with two types of nodes, OObjects referred to by the meta-variable O, and ODomains referred to by the meta-variable D. Edges between OObjects, OEdges referred to by the meta-variable E, correspond to points-to relations due to field references.

An OObject O is represented using the tuple $C\texttt{<}\overline{D}\texttt{>}$ to mean an abstract object of type C and actual ODomains $\overline{D}$, where $D1$ is the owning ODomain of O. Thus, in the object hierarchy, the abstract object $C\texttt{<}\overline{D}\texttt{>}$ is shown as the child of ODomain $D1$, which in turn is the child of some other OObject. The OGraph has a single root object. The full list of ODomains $\overline{D}$ identifies the OObject O; conceptually, these are the domains that the object has access to, i.e., it references objects from those domains.

By having abstract objects of the form $C\texttt{<}\overline{D}\texttt{>}$, the OOG can distinguish between different abstract objects of the same class C that are in different owning domains $D1$, even if created at the same object creation expression. In Ownership Domains, a creation expression is `new` $C\texttt{<}\overline{p}\texttt{>}$`()`, where `owner` is the owner domain and $\overline{p}$ are parameters, instead of `new` $C()$ in Java-like languages. Even if they have the same ODomains, two abstract objects can be distinguished if they have different ODomains. This situation occurs, for example, when two collections have the same owning domain, but have different domains for their elements. In addition, the analysis treats an instance of class C with actual parameters $\overline{p}$ differently from another instance that has actual parameters $\overline{p'}$. Hence, the data type of an OObject uses $C\texttt{<}\overline{D}\texttt{>}$, where the first element D_1 of $\overline{D}$ is the OObject's owning ODomain. By considering $C\texttt{<}\overline{D}\texttt{>}$ instead of just a class C and an owning ODomain, the OOG avoids merging objects excessively: it merges two objects of the same class only if all of their domains are the same.

Although a domain d is declared by class C, each instance of C gets its own runtime domain $\ell.d$. For example, if there are two distinct object locations ℓ and ℓ' of class C, then $\ell.d$ and $\ell'.d$ are distinct. Since an ODomain represents an abstraction of a runtime domain $\ell_i.d_i$, one domain declaration d in a class C can correspond to multiple ODomains D_i in the OGraph, where each D_i gets a fresh identifier D_{id}.

The static analysis computes an abstract object of type C in some domain D, based on mapping domain parameters in the code to domains that may be declared by other classes C_i. To measure the differences between the code structure and the abstract runtime structure, the metrics relate C to C_is.

Abstract interpretation. The OOG extraction analysis is a kind of a points-to analysis. A points-to analysis typically merges all the objects created at the same object creation expression into an equivalence class, attaching an object label h at *each* object creation expression `new` $A()$, that we show as a superscript:

$$\texttt{new}^{h}\ A()$$

$$\begin{aligned}
&\texttt{class}\ A < \texttt{owner}, \alpha_A >\ \texttt{extends}\ A'\ \{ && (1)\\
&\} && (2)\\
&[\ \texttt{this} \mapsto O_C\ ] && (3)\\
&\texttt{class}\ C < \texttt{owner}, \alpha_C >\ \texttt{extends}\ C'\ \{ && (4)\\
&\quad \texttt{domain}\ d_{local}; && (5)\\
&\qquad \rightarrow [\ D_{local} = \langle o.d, C :: d_{local} \rangle] && (6)\\
&\quad \texttt{void}\ m()\ \{ && (7)\\
&\qquad \texttt{new}^L\ A{<}p_{owner}, \alpha_C{>}(); && (8)\\
&\qquad\quad \rightarrow [\ O_A = \langle\ A{<}\overline{D_A}{>}\ ] && (9)\\
&\qquad\quad \rightarrow [\ (O_C, p_{owner}) \mapsto D_1,\ (O_C, \alpha_C) \mapsto D_2\ ] && (10)\\
&\qquad [\ \exists O_B, \exists d \text{ s.t. } (O_B, d) \mapsto D1 \text{ and } O_B = \langle B{<}\overline{D_B}{>} \rangle\ ]\}\} && (11)
\end{aligned}$$

The OOG extraction analysis interprets an object creation expression (line 8) as follows. The first parameter p_{owner} is the owning domain, and α_C is an additional domain parameter. Each formal domain parameter is bound to some other domain. During abstract interpretation, the analysis tracks the bindings of each domain in the code and maps it to an ODomain D_i in the OGraph. In particular, in some analysis context O_C, the analysis binds p_{owner} to some ODomain D (line 10), where D is the child domain of some OObject O_B of type B and some actual domains (line 11). In the OGraph, O_B is in the set of parent objects of O_C ($parentObj(O_C)$).

To recap, the analysis labels all the objects that may be created at this object creation expression with a label L where $L = O_A = \langle\ A{<}\overline{D_A}{>}\ \rangle$. Compared to the label h of a standard points-to analysis, the label L is different as follows. First, multiple object creation expressions of the type A can still be represented by the same L label, if the analysis context maps the domain parameters $\overline{p}$ to the same $\overline{D}$, whereas a basic points-to analysis will create multiple h labels. Second, if the analysis context maps p_{owner} and α_C to n different $\langle A, D \rangle$ combinations, then one object creation expression of type A may create n different abstract objects, and thus n different L labels.

Abstract edges. An OEdge in the OGraph is a directed edge from a source OObject O_{src} to a destination OObject O_{dst}. A points-to OEdge is due to some field declaration $T f$ in some class in the code, where $T = C'{<}\texttt{owner}, \alpha{>}$.

To add edges between objects, the OOG extraction analyzes a field declaration $T f$ in class C, in a given analysis context, the analysis maps the owning domain p'_1 to an ODomain D. It looks up in D each OObject O_t of type C_t, where C_t is a subtype of C'. It then creates multiple OEdges, where each edge has as its origin the OObject corresponding to the current object (the one for the `this` object), and as its destination each OObject O_t in D. As a result, the edges in an OOG are more precise than the edges that are added between objects on an object graph based on just *type* information. The latter approach adds an edge from any object of type C or a subtype thereof to any object of type C' or a subtype thereof, when class C has a field of type C'.

4 Metrics

In the following, an abstract runtime element is either an abstract object or an abstract edge in the abstract runtime structure. In this section, we define metrics that relate *one or more* code elements in the code structure to *one or more* abstract runtime element.

Research Questions. The metrics measure the differences between the code structure and the abstract runtime structure, based on the following research questions:

RQ1 How often does *one* code element map to *many* abstract runtime elements?

Rationale: in general, one class has many instances at runtime. If the object graph merged all those instances, it would not add much information over the code structure.

RQ2 How often do *many* code elements map to *one* abstract runtime element?
Rationale: if the code and the runtime structure mirrored each other closely, distinct code elements correspond to distinct runtime elements, but when they do not, it would be interesting to understand why not.

RQ3 How often does the location of an abstract runtime element mismatch the location of its corresponding code element?
Rationale: if the syntactic location always matched the semantic location of an element, it would not be that helpful to have an object graph.

RQ4 How often does the abstract runtime structure have more precision than the code structure?
Rationale: runtime type information is more precise than static type information. The abstract object graph is extracted statically, so does it still achieve precision, e.g., due to the domain annotations, or to the fact that the abstract interpretation resolves formals to actuals?

For each research question, we define several metrics. We informally describe each metric and illustrate it using the pedagogical framework Minidraw (MD). We also formally define each metric. For some metrics that use types, we define two versions, one that considers by default the full genericized type, e.g., Which-A of `List<String>`, and one that considers only the raw type, e.g., Which-A raw of `List`. This distinction is useful to measure when the full generic type (with the argument) can be used to distinguish between different instantiations of the same parameterized generic type. Finally, a type A is *instantiated* if an OObject of type A exists in the OGraph. A type is *instantiatable* if it is neither an abstract class nor an interface.

4.1 RQ1: How often does one code element map to many abstract runtime elements?

In RQ1, we consider the *one-to-many* mapping from code elements to elements in the runtime structure.

The code structure such as a class diagram will show just one box for the class A. On the other hand, an abstraction of the runtime structure such as an OGraph can show different OObjects that have the same type A. Typically, different abstract objects of the same type serve different roles in the design.

4.1.1 Which-A (WA).

This metric measures how many OObjects of type A exist in the OGraph. More formally, for each instantiated type A, *WA(A)* computes the following set:

$$\{A_i \text{ where } O_i = \langle A_i{<}\overline{D_i}{>}\rangle\}$$

Ranges. *WA* ranges from 1 to the size of the OGraph. For values of 2 or higher *WA* indicates that at least two OObjects of the same type exist in the OGraph.

4.1.2 Which-A-in-B (WAB).

An OGraph can express design intent using multiple sibling domains per object to distinguish between the different parts of an object's substructure. For example, the substructure of the `bd:BDrawing` has two domains and one object of type `HashMap` in each (Fig. 1). *WAB* measures how frequently different OObjects of the same type A are in the same parent OObject B, but in different domains of B.

More formally, *WAB* finds the unordered pairs of objects that satisfy the condition:

$$\begin{aligned}
&\{O_i = \langle A_i{<}\overline{D_i}{>}\rangle, O_j = \langle A_j{<}\overline{D_j}{>}\rangle\}\\
&\exists O_{B_i} = \langle B_i{<}\overline{D_{B_i}}{>}\rangle, \text{ s.t. } (O_{B_i}, d_i) \mapsto D_{i_1}, d_i \in \mathit{domains}(B_i)\\
&\exists O_{B_j} = \langle B_j{<}\overline{D_{B_j}}{>}\rangle, \text{ s.t. } (O_{B_j}, d_j) \mapsto D_{j_1}, d_j \in \mathit{domains}(B_j)\\
&\text{where } A_i = A_j \text{ and } B_i = B_j \text{ and } d_i \neq d_j
\end{aligned}$$

Inheritance Type Hierarchy

```
+- Object
  +-AFigure
    +-IFigure
      +-BFigure
    +-CFigure
       +-StDrawing
         +-BDrawing
```

Hierarchy of objects

```
+- main:Breakthrough
  +- MODEL
    +- bd:BDrawing
      +- OWNED
        +- e:FCEvent
    +- bf:BFigure
      +- OWNED
        +- e:FCEvent
    +- p:Position
```

```
class AFigure implements Figure {
  FCEvent<OWNED> e = new FCEvent(this);
}
class FCEvent { Figure f; }
class BPFactory<OWNER,M> {
  Position<M> p = new Position(...);
}
class MCommand<OWNER,M> {
  Position<M> from = new Position(...);
  Position<M> to = new Position(...);
}
class Breakthrough {
  domain MODEL,CTRL;
  BPFactory<CTRL,MODEL> bpf =...;
  MCommand<CTRL,MODEL> mc =...;
}
```

HMO(`new FCEvent`) = 2   1FnE(`Figure f`) = 4
TMO(`bf:BFigure`) = 4
TOS(`main:Breakthrough`) = 3
HMN(`p:Position`) = 3

Figure 4: One-to-many and many-to-one differences: One object creation expression in the base class `AFigure`, corresponds to two `FCEvent` objects that have different roles. Many distinct object creation expressions `new Position(...)` correspond to one object of type `Position` in OGraph.

4.1.3 Which-A-in-Which-B (WAWB).

WAWB measures how frequently different OObjects of the same type A are in different parents OObjects of different types. For example, the objects `bd:BDrawing` and `bf:BFigure` each have in their substructure distinct objects of type `HashMap` (Fig. 1).

More formally, *WAWB* finds the unordered pairs of objects that satisfy the condition:

$$\begin{array}{l}\{O_i = \langle A_i{<}\overline{D_i}{>}\rangle, O_j = \langle A_j{<}\overline{D_j}{>}\rangle\}\\ \exists O_{B_i} \in parentObj(O_i) \text{ and } \exists O_{B_j} \in parentObj(O_j)\\ \text{where } A_i = A_j \text{ and } B_i \neq B_j\end{array}$$

Ranges. For the metrics *WAB* and *WAWB*, we are interested in the number of unordered pairs of objects. The larger the number is, the more often objects of the same type are used in different contexts (domains or enclosing types) and with different roles.

4.1.4 Same New Expression Different Objects (HMO).

This metric measures how many distinct OObjects in the OGraph correspond to the same object creation expression `new` $C()$ in the code. For example, for the object creation expression `new FCEvent(this)` in the base class `AFigure`, the OGraph shows two objects of type `FCEvent` in the domains `OWNED` of the `BDrawing` object and the `BFigure` object, respectively (Fig. 4).

From an OObject O_A, $traceToCode(O_A)$ is the set of nodes from the AST that O_A can be traced to. An OObject is traced to a set of object creation expressions. Formally, *HMO* computes:

$$\{O_i\} \text{ where } \texttt{new } A{<}...{>}(...) \in traceToCode(O_i)$$

4.1.5 One Field Declaration Many Edges (1FnE).

This metric measures how many edges in the OGraph are due to the same field declaration in the code. For the field declaration `Figure f` in the type declaration `FCEvent`, the OGraph shows 4 distinct points-to edges, 2 for each `FCEvent` object (Fig. 4). For a points-to OEdge E, $traceToCode(E)$ is the set of field declarations that E can be traced to.

Formally, *1FnE* computes the set:

$$\{E_i\} \text{ where } (T\ f) \in traceToCode(E_i)$$

Ranges. The values of *HMO* and *1FnE* range from 1 to the number of objects and edges, where values greater than 2 indicate that many abstract runtime elements map to one code element.

4.2 RQ2: How often do many code elements map to one abstract runtime element?

In RQ2, we consider the *many-to-one* mapping from code elements to elements of the runtime structure.

4.2.1 Different New Expressions Same Object (HMN).

This metric measures how many distinct object creation expressions `new` $A()$ are abstracted by the same OObject O_A. For example, there are three object creation expressions `new Position` that are mapped to the same OObject of type `Position` that represents a position on the board (Fig. 4). Formally, for each OObject O_A, *HMN* is defined as the set $traceToCode(O_A)$.

4.2.2 Types Merged by Object (TMO).

This metric measures the number of distinct types, excluding interfaces, that are merged by an OObject. *TMO* measures the effect of collapsing the inheritance hierarchy. For example, *TMO* of the `BFigure` object is 4 because the object merges the types: `BFigure`, `IFigure`, `AFigure`, and `Figure` (Fig. 4).

Formally, for each OObject O, *TMO* is defined as the set:

$$\{A'_i\} \text{ where } A_i <: A'_i \text{ and } O_i = \langle A_i{<}\overline{D_i}{>}\rangle$$

4.2.3 Types in Object Substructure (TOS).

This metric measures the number of distinct types that are in the sub-structure of an OObject. It uses the transitive *descendants* of the OObject O, which includes O, the children of O and their children, recursively.

For example (Fig. 4), for the `main:Breakthrough` OObject, *TOS* is 3 and includes the types `BDrawing`, `BFigure`, and `FCEvent`, where the type `FCEvent` of the descendants is counted once.

Formally, *TOS* is defined as the set:

$$\{A_i\} \text{ where } O_i \in descendants(O) \text{ and } O_i = \langle A_i{<}\overline{D_i}{>}\rangle$$

Ranges. *TMO* ranges between 1 and the number types excluding interfaces, and is greater than 2 for all objects of a type different from `Object`. *TOS* ranges between 1 and the number of instantiated types.

```
class BDrawing<OWNER,Df> {
  domain OWNED;
  Figure<Df> bf = new BFigure<Df>();
}
class Breakthrough {
 domain MODEL,CTRL;
 BDrawing<CTRL,MODEL> bd = new BDrawing<CTRL,MODEL>();
}
```
Figure 5: The object of type `BFigure` is created in class `BDrawing` using the formal domain parameter `Df`. In the hierarchy of objects (Fig. 1), the object `BFigure` appears in the domain `MODEL` because `Df` is bound to `MODEL`.

4.3 RQ3: How often does the location of an abstract runtime element mismatch the location of its corresponding code element?

4.3.1 Same Package Different Domains (1PnD).

This metric measures how often objects are in different domains, but are of types that are in the same package. For example, the package `minidraw.standard.handlers` has 5 types. The objects of these types are in 3 distinct domains such as `TRACKERS` of the `SelectionTool` object, `CTRL` of the `Breakthrough` object and `HANDLERS` of `BDrawing` (Fig. 1).

More formally, for all instantiatable types A_i in that system qualified by their packages names P_i, denoted by the set $\{P_i.A_i\}$, the metric finds the unordered pairs S_i such that:

$$S_i = \{\langle P_i.A_i{<}\overline{D_i}{>}\rangle, \langle P_j.A_j{<}\overline{D_j}{>}\}$$
$$\text{where } P_i = P_j, A_i = A_j, D_i \neq D_j$$

4.3.2 Same Domain Different Packages (1DnP).

This metric measures how often objects are in the same domain, but have types that are declared in different packages. For example, the objects of type `SelectionTool` and `SRBSStrategy` are in the same domain `CTRL`, but in different packages (Fig. 1).

More formally, for each instantiatable type A_i in the system qualified by its package names P_i, denoted by the set $\{P_i.A_i\}$, the metric finds the unordered pairs S_i such that:

$$S_i = \{\langle P_i.A_i{<}\overline{D_i}{>}\rangle, \langle P_j.A_j{<}\overline{D_j}{>}\}$$
$$\text{where } P_i \neq P_j, A_i \neq A_j, D_i = D_j$$

Ranges. The values of *1DnP* and *1PnD* range between 0 and the number of domains and packages. The greater the values, the more mismatched is the hierarchy of classes from the hierarchy of objects.

4.3.3 Pulled Objects (PO).

In the OGraph, each OObject that is assigned to a domain must appear where that domain is declared. An object of type A can appear in a domain D that is inside some parent object of type B. In the code, however, the class B may not directly create an object of type A. An object of type A is pulled in a domain D of B if the object creation expression of A is in a class C, the actual domain parameter p_{owner} is a formal domain parameter of C, and the analysis binds p_{owner} to the actual domain D declared in B. For example, the object of type `BFigure` is pulled to the domain `MODEL` in the object `Breakthrough`, although its object creation expression is in the class `BDrawing` (Fig. 5).

Formally, let $params(C)$ be the list of formal domain parameters declared on a class C. Let the domain d be in the list of locally declared domains on a class B ($domains(B)$). And let O_A be an OObject of type A

that is created by the analysis in the context O_X. If an object creation expression of type A is in a class C, then C is in the $declaringTypes(O)$ (Fig. 5), of which there can be several. Let the first domain provided to the object creation expression, i.e., the owner, be a domain d_f in scope. d_f can be either a formal domain parameter or a local domain of the class C. In general, based on the context O_X, the OGraph can map d_f to an ODomain D_1, and D_1 can be the same as some ODomain that came from a locally declared domain d on a class B. In some cases, B can be the same as C. When B is not the same as C, the object of type A, O_A, represents an object that was "pulled from" the domain parameter d_f to the domain D_1 in an object of type B. The metric *PO_P* measures the percentage of pulled objects compared to all OObjects in the OGraph.

$$\begin{aligned}
&O = \langle A\texttt{<}\overline{D}\texttt{>}\rangle, O_B = \langle B\texttt{<}\overline{D_B}\texttt{>}\rangle, O_B \in parentObj(O)\\
&\text{where } C \in declaringTypes(O) \text{ and } B \neq C\\
&\texttt{new } A\texttt{<}d_f, ...\texttt{>}(...) \in traceToCode(O), \text{ where } d_f \in params(C)\\
&(O_X, d_f) \mapsto D_1 \text{ and } (O_B, d) \mapsto D_1, \text{ where } d \in domains(B)
\end{aligned}$$

4.3.4 Scattered Objects (SO).

Multiple elements in the code structure, e.g., multiple object creation expressions, may correspond to the same element in the OGraph. For one OObject O of type A in the OGraph, the code may have multiple object creation expressions `new A()` scattered across multiple type declarations.

For an OObject O, *scattering(O)* is the set of distinct classes that contain object creation expressions in *traceToCode(O)*, as the set $declaringTypes(O)$. If a class has an inner class and the object creation expression is in the body of the inner class, $declaringTypes(O)$ includes only the inner class. We also define the *scatteringFactor (SO_F)* for OObjects that trace to more than two object creation expressions.

$$\begin{aligned}
&scattering(O) = |declaringTypes(O)|\\
&SO\_F(O) = scatteringFactor(O) = 1 - \frac{1}{scattering(O)}\\
&SO\_F\_Mean = \frac{1}{n} \cdot \sum_{i=1}^{n} scatteringFactor(O_i)\\
&n = |\ \{O_i,\ \text{where } |traceToCode(O_i)| \geq 2\}\ |
\end{aligned}$$

Ranges. *SO_F* ranges between 0 and 1, where 0 means an object is created in exactly one type declaration. The closer is *SO_F* to 1.0, the more declaring types exist. We use 0.5 as the *SO_F* threshold to indicate at least two declarations.

4.3.5 Inherited Domains (InhD).

An object of type A that is not created in a class B may appear in a parent object of type B in an ODomain D_1, but the domain declaration d can be in another class C, where the type C is a super-type of B and the domain D is inherited from C to B. The metric *Inherited Domains Percentage (InhD_P)* measures the percentage of inherited domains, compared to all domains, in the OGraph.

$$\begin{aligned}
&O = \langle A\texttt{<}\overline{D}\texttt{>}\rangle, \text{ where } C \in declaringTypes(O)\\
&(O_B, d) \mapsto D_1, O_B = \langle B\texttt{<}\overline{D_B}\texttt{>}\rangle\\
&O \text{ is in } D_1, \text{ where } d \in domains(C) \text{ and } B <: C
\end{aligned}$$

4.3.6 Inherited Points-To Edges (InhE).

An OEdge from $O_{src} = \langle A_{src}{<}\overline{D_{src}}{>}\rangle$ to a destination OObject $O_{dst} = \langle A_{dst}{<}\overline{D_{dst}}{>}\rangle$ may be due to a field declaration $C{<}\overline{p}{>}f$ in the class A_{src} or one of its super-classes, in which case the points-to edge is due to an inherited field. Also, A_{dst} must be a subtype of C.

When looking at an object in the watch window of a debugger, one can see all the fields on an object including inherited ones. But when looking at a class declaration one at a time, inherited fields that lead to additional points-to edges may not be obvious. The field declaration corresponding to an edge may not be in the field declarations of A_{src} ($fields(A_{src})$). Instead, the field declaration can be in another class A', where A' is a super-class of A_{src}. The metric *Inherited Edges Percentage (InhE_P)* is the percentage of points-to edges that are due to inherited fields, compared to all points-to edges in the OGraph.

$$\begin{aligned}
&\langle\ \textbf{From} = O_{src},\ \textbf{To} = O_{dst}, \textbf{Field} = f\ \rangle \in \mathsf{OEdge}\\
&O_{src} = \langle A_{src}{<}\overline{D_{src}}{>}\rangle \text{ and } O_{dst} = \langle A_{dst}{<}\overline{D_{dst}}{>}\rangle\\
&\ \text{where } C{<}\overline{p}{>}\ f \notin fields(A_{src}) \text{ and } C{<}\overline{p}{>}\ f \in fields(A')\\
&\ \text{such that } A_{src} <: A'
\end{aligned}$$

Ranges. A value of 0 for *InhE_P* indicates no mismatch between the type of source of the OEdges and the declaring type of the field.

4.4 RQ4: How often does the abstract runtime structure have more precision than the code structure?

4.4.1 Points-To Edge Precision (PTEP).

A field type can be an interface C that can be part of a deep inheritance hierarchy with several abstract classes and concrete subclasses. A field is also a reference to some OObject. A code structure such as a type hierarchy will show all possible subclasses of C. The OGraph, on the other hand, can have more precise information. In a points-to edge, A_{dst} must be a subtype of C, the declared class of the field f. *PTEP* captures the points-to edges on the OGraph, where the number of all possible subclasses of C is greater than the number of subclasses of C that are actually instantiated in the reachable domain D_{dst}. A points-to edge may exist from an object A_{src} to an object A_{dst} such that the class A_{dst} is a subclass of C and that is in a reachable ODomain D_{dst}, after binding formal to actual domains. Thus, the OGraph shows points-to edges to only a subset of all possible objects of type C.

We define *precisionRatio* as the ratio between the two sets associated with a field declaration. Since a lower ratio indicates a higher precision, we also define the points-to edge precision factor (*PTEP_F*). In MD, for a field declaration `TOOL<TRACKERS> fChild` in `SelectionTool`, the OGraph shows 3 points-to edges, which is more precise than if it were to consider all 7 classes that implement the `Tool` interface. Thus, the *PTEP_F* is $1 - \frac{3}{7} = 0.57$.

$$\begin{aligned}
&D_{dst} = mapFtoA(O_X, p)\ //\ map\ domain\ in\ the\ code\ to\ ODomain\\
&precisionRatio(C{<}\overline{p}{>}\ f) = \frac{|OOGPossibleSubTypes(C, D_{dst})|}{|AllPossibleSubClasses(C)|}\\
&PTEP\_F(C{<}\overline{p}{>}\ f) = 1 - precisionRatio(C{<}\overline{p}{>}\ f)
\end{aligned}$$

Ranges. *PTEP_F* is 0 if the information in the OGraph is not more precise than using the type hierarchy and finding instances of all the possible subtypes. If *PTEP_F* is above a certain threshold, the abstract runtime structure shows more precise points-to edges than the code structure. We use the threshold of 0.5, which means that the runtime structure shows points-to edges for less than half of all possible subclasses. A positive *PTEP_F*, but smaller than the threshold, indicates that more precision is needed than the OGraph provides.

Table 1: List of systems: *Inst. Types*, *Abs. Cls*, *Itfs* are the number of instantiatable types, abstract classes and interfaces.

Abbr.	**KLOC**	**Notes**	**All Types**	**Inst. Types**	**Abs. Cls**	**Itfs**	**Objects**	**Edges**
MD	1.4	MiniDraw	68	47	2	19	46	124
CDB	2.3	CrytoDB	47	38	2	7	48	104
AFS	14.4	Apache FtpServer	173	112	12	49	140	429
DL	8.8	DrawLets	165	111	18	36	610	2528
PX	36	Pathway-Express	300	238	22	40	422	3485
JHD	18.0	JHotDraw	306	241	21	44	410	6422
HC	15.6	HillClimber	206	171	6	29	353	4494
APD	8.2	Aphyds	70	55	4	11	50	70
Total	104.7							

5 Results

We compute the metrics on a corpus of object-oriented code totaling over 100 KLOC and consisting of eight subject systems (Table 1) that already have annotations and extracted `OGraph`s [VAA13]. Additional information about these systems and their code structure such as the number of classes and the depth of the inheritance tree is also available [VAA13, Table 2].

The implementation generates sets of code elements or runtime elements that satisfy each metric. Due to space limits, the paper presents only a few interesting examples, but the generated datasets are in the online appendix [Abi14]. The generated sets are then used as input to the statistical analysis.

In Table 2, we compute the *p-value* based on the one-sample Wilcoxon non-parametric test to test if the difference between the code and runtime structure is statistically significant. We also estimate the magnitude of the difference using Cliff's Delta D, a non-parametric effect size for ordinal data. Some statistics tests require a control value that indicates no difference between the code and runtime structure. For each metric, the control value is the first column in Table 3. We chose non-parametric statistics because the set sizes are non-normally distributed. D ranges between $+1$ if all selected values of the metrics are higher than the control value and -1 if the reverse is true. In this work, all the D values are positive. The effect size D is considered negligible for $0 \leq D < 0.147$, small for $0.147 \leq D < 0.333$, medium for $0.333 \leq D < 0.474$ and large for $D \geq 0.474$ [Coh88]. In Table 2, significant values for p are in bold, and medium or large values for D are in italics. Finally, Δ estimates the mean difference between the metric and the minimum range. For example, for *Types-Merged-By-Object (TMO)*, the control value is 2 since every object merges the object type and the class `Object`. A *p-value* lower than 0.05, a positive D and Δ greater than 1 indicate that, on average, an object merges 3 types.

In Table 3, we show as rows several descriptive statistics such as Median, Mean or Maximum, for each metric and for each system, e.g., *SO_F_Median*, *SO_F_Mean* and *SO_F_Max*. We also show as columns several descriptive statistics such as the Minimum, Maximum, Standard Deviation and Mean, for each metric, but across all the systems.

5.1 RQ1: How often does one code element map to many abstract runtime elements?

5.1.1 Which-A (WA) and raw type version (WAR).

WA_Max ranges between 2 (MD) and 31 (JHD). The results are statistically significant for all systems. For most systems, however, D indicates a negligible or small effect, except DL. For DL, D indicates a medium effect size. Indeed, there are many objects of the same type for 39 of the 102 instantiated types, or 36 out of 92 if raw types are considered. The maximum is reached for the `Vector` class, and only 8 out of 39 types and 5 out of the 36 raw types are from the Java library. The results imply that many objects of the same

Table 2: Statistical analysis for the set metrics: p-value (p), Cliff's Delta (D) and Cliff's Delta size (Δ). Highlighted are values of $p < \mathbf{0.05}$ and $\mathbf{0.333} \leq D$ (medium).

Metric	MD			CDB			AFS			DL			PX			JHD			HC			
	p	D	Δ	p	D	Δ	p	D	Δ	p	D	Δ	p	D	Δ	p	D	Δ	p	D	Δ	
WA	**0.00**	0.12	0	**0.00**	0.31	0	**0.00**	0.14	0	**0.00**	*0.69*	3	**0.00**	0.27	1	**0.00**	0.09	0	**0.00**	0.24	1	**0**
WAR	**0.00**	0.11	0	**0.00**	0.29	1	**0.00**	0.17	0	**0.00**	*0.65*	3	**0.00**	0.23	1	**0.00**	0.07	1	**0.00**	0.22	1	**0**
WAB	0.50		0	**0.04**	0.12	0	0.05		0	**0.00**	*0.43*	2	**0.00**	0.04	1	1.00		0	**0.01**	0.04	1	0
WABR	0.19		0	0.09		0	0.05		0	**0.00**	*0.48*	2	**0.02**	0.04	3	1.00		0	**0.01**	0.05	1	0
WAWB	**0.02**	0.12	0	0.09		0	**0.01**	0.08	0	**0.00**	0.10	13	**0.00**	0.12	3	**0.00**	0.07	5	**0.00**	0.07	3	0
WAWBR	0.09		1	0.09		1	**0.00**	0.09	1	**0.03**	0.05	49	**0.00**	0.11	23	**0.00**	0.05	23	**0.00**	0.07	4	0
HMO	**0.00**	0.06	0	**0.00**	0.21	0	**0.00**	0.08	0	**0.00**	*0.54*	2	**0.00**	0.05	0	**0.00**	0.05	0	**0.00**	0.11	0	**0**
1FnE	**0.00**	0.17	0	**0.00**	0.29	1	**0.00**	0.09	2	**0.00**	*0.91*	37	**0.00**	*0.48*	3	**0.00**	*0.59*	22	**0.00**	0.25	1	**0**
HMN	**0.00**	0.22	1	**0.00**	0.25	1	**0.00**	0.27	1	**0.00**	0.22	1	**0.00**	*0.37*	2	**0.00**	0.25	1	**0.00**	0.28	1	**0**
TMO	**0.00**	0.17	0	**0.00**	*0.48*	1	**0.00**	0.27	1	**0.00**	0.29	1	**0.00**	-0.00	0	**0.00**	*0.39*	1	**0.00**	0.13	0	**0**
TOS	**0.00**	0.20	1	**0.00**	0.28	1	**0.00**	0.19	1	**0.00**	0.17	1	**0.00**	0.21	1	**0.00**	*0.41*	1	**0.00**	0.24	1	**0**
1PnD	**0.03**	*0.42*	1	**0.02**	*0.67*	3	**0.00**	0.31	1	**0.00**	*0.80*	5	**0.00**	*0.42*	3	**0.00**	0.24	1	**0.02**	*0.67*	10	**0**
1DnP	0.05	*0.40*	2	**0.03**	0.25	1	**0.03**	0.19	2	0.05		1	**0.03**	0.06	1	**0.00**	*0.48*	3	**0.03**	0.07	0	0

Table 3: All metrics across all systems: Min, Max, Standard Deviation and Mean.

		Control	MD	CDB	AFS	DL	PX	JHD	HC	APD	Min	Max	Std Dev	Mean
RQ1	WA_Median	1	1	1	1	3	1	1	1	1	1	3	1	1
	WA_Max	1	2	6	9	28	23	31	27	3	2	31	12	16
	WAR_Median	1	1	1	1	3	1	1	1	1	1	3	1	1
	WAR_Max	1	7	8	9	90	71	45	27	5	5	90	33	33
	WAB_Median	0	0	0	0	0	0	0	0	0	0	0	0	0
	WAB_Max	0	1	3	15	54	105	0	28	1	0	105	37	26
	WABR_Median	0	0	0	0	0	0	0	0	0	0	0	0	0
	WABR_Max	0	5	4	15	96	503	0	28	1	0	503	173	82
	WAWB_Median	0	0	0	0	0	0	0	0	0	0	0	0	0
	WAWB_Max	0	1	12	13	356	213	495	204	2	1	495	189	162
	WAWBR_Median	0	0	0	0	0	0	0	0	0	0	0	0	0
	WAWBR_Max	0	18	24	13	3827	1842	4029	204	5	5	4029	1769	1245
	HMO_Median	1	1	1	1	3	1	1	1	1	1	3	1	1
	HMO_Max	1	2	3	6	24	31	31	33	2	2	33	14	16
	1FnE_N_Median	1	1	1	1	3	2	2	1	1	1	3	1	2
	1FnE_N_Max	1	12	4	76	924	474	961	87	2	2	961	416	318
RQ2	HMN_Median	1	1	1	1	1	1	1	1	1	1	1	0	1
	HMN_Max	1	13	9	13	23	122	94	64	9	9	122	44	43
	TMO_N_Median	2	2	2	2	2	2	2	2	2	2	2	0	2
	TMO_N_Max	2	5	4	5	9	5	6	4	5	4	9	2	5
	TOS_N_Median	1	1	1	1	1	1	1	1	1	1	1	0	1
	TOS_N_Max	1	11	9	36	10	33	160	14	10	9	160	52	35
RQ3	1PnD_Median	0	0	2	0	3	0	0	5	1	0	5	2	1
	1PnD_Max	0	4	10	8	23	66	16	34	5	4	66	21	21
	1DnP_Median	0	0	0	0	0	0	0	0	0	0	0	0	0
	1DnP_Max	0	6	5	21	20	23	31	9	9	5	31	10	16
	PO_P%	0	41	32	50	73	28	44	40	5	5	73	20	39
	SO_F_Median%	0	50	50	0	50	0	0	50	0	0	50	27	25
	SO_F_Mean%	0	35	33	26	57	31	30	48	12	12	57	13	34
	SO_F_Max%	0	75	80	83	93	95	97	95	75	75	97	9	87
RQ4	PTEP_F_Median%	0	50	0	0	0	0	0	0	0	0	50	18	6
	PTEP_F_Mean%	0	43	0	23	25	11	12	12	2	0	43	14	16
	PTEP_F_Max%	0	92	0	82	99	100	79	76	50	0	100	33	72

type exist for a few of the types. This is unsurprising since the *WA_Median* is 1 for all systems except DL. Further inspection indicates that *WA* and *WAR* are greater than 1 for data structures such as `HashMap` or `ArrayList` or types from the Java library such as `File`.

5.1.2 Which-A-in-B (WAB) and raw type version (WABR).

WAB_Max ranges between 0 in most systems and 15 in AFS. The results are not statistically significant, except for DL, and indicate that rarely do objects of the same type have different roles in the substructure of the same parent object. In other words, multiple domains inside one object are rarely used to distinguish between instances of the same type within the same parent object. For DL, there are 24 pairs of objects of same type such as `Line` and `PropertyChangeEvent` in different domains and the difference is statistically significant.

5.1.3 Which-A-in-Which-B (WAWB) and raw type version (WAWBR).

WAWB_Max ranges between 1 in MD and 495 in JHD, indicating that objects of the same type often serve different roles. The results are statistically significant for all systems, except APD and CDB. Similarly to *WA* and *WAB*, D indicates a negligible or small effect for most systems. *WAWB_Max* is higher for DL, PX, JHD, and HC, and tends to involve objects that are collections, like instances of `Vector<Figure>` and `Hashtable<String,String>`. In most cases, the raw type version *WAWBR_Max* is greater than the generic type version *WAWB_Max*. This is unsurprising since generic types express design intent, and can thus distinguish between objects that have different roles. For some systems that use many generic collections, *WAWB_Max* is also high, indicating that objects can have the same generic type but can still have different roles, even when generic types are used.

5.1.4 Same New Expression Different Objects (HMO).

HMO_Max ranges between 2 (MD, APD) and 31 (PX, JHD) and 33 (HC). For example, in DL, in the body of the class `AFigure`, the expression `new Vector<PropertyChangeListener>()` generates 7 different objects. The results are statistically significant for all systems. The negligible or small effect size for most systems indicates that the extraction analysis does not create a spurious number of abstract objects for one object creation expression.

5.1.5 One Field Declaration Many Edges (1FnE).

1FnE_Max ranges between 2 (APD) and 961 (JHD). The results are statistically significant for all systems and D indicates a large effect size for DL and JHD, a medium effect size for PX, and a small effect size for the remaining systems. Δ is also higher for DL and JHD. The higher D values for edges indicates more prominent differences between code and runtime structure when it comes to abstract edges (relations) between objects than when it comes to objects. The difference is higher for DL and JHD, for which Δ is 37 and 22, respectively. The higher values occur for field declarations in abstract classes where the field type is an interface or an abstract class. For example, the data point for *1FnE_Max* in JHD is for the field declaration `Command myObservedCommand` in the inner class `EventDispatcher` of the abstract class `AbstractCommand`. The `Command` interface is implemented by 20 classes.

5.2 RQ2: How often do many code elements map to one abstract runtime element?

5.2.1 Different New Expressions Same Object (HMN).

HMN_Max ranges between 9 (CDB, APD) and 122 (PX). The results are statistically significant for all systems. The effect size is medium for PX and small for the others. A high value for *HMN_Max* occurs for object creation expressions of Java library types such as `String` and `JLabel`. Most of these objects are in the `SHARED` domain. In PX, an interesting example is a `MultiValuedSetHashtable` object that traces to 10 different object creation expressions in the code. These objects are used to store temporary data being read from database tables and have the same role.

5.2.2 Types Merged by Object (TMO).

TMO_Max ranges between 4 (CDB) and 9 (DL). Unsurprisingly, the values are higher for systems that use inheritance. For all systems, the results are statistically significant with a medium effect size for CDB and JHD and a negligible or small effect size for MD, AFS, PX, HC and APD. Δ is highest for DL, where 12 types are merged by an object of type `ConnectingLine` including abstract classes such as `AbstractShape` and `AFigure`, and interfaces such as `Paintable`, `Duplicatable` and `Figure`.

5.2.3 Types in Object Substructure (TOS).

While *TMO* measures differences due to inheritance, *TOS* focuses on composition. *TOS_Max* ranges between 9 (APD) and 160 (JHD). Unsurprisingly, the maximum is reached for the root of the object hierarchy, but high values occur for other objects as well: for example, the object `PEInputFrame` in PX merges 25 types in its substructure. The results indicate a small or medium—but still statistically significant—difference for most systems.

5.3 RQ3: How often does the location of an abstract runtime element mismatch the location of its corresponding code element?

5.3.1 Same Package Different Domains (1PnD).

1PnD_Max ranges between 4 and 138 and indicates that objects of types in the same package are often in different domains. The results are statistically significant for all systems except APD where all the classes are in one package. The effect size is large for CDB, DL and HC, medium for AFS and PX and small for JHD (which has a very precise package organization), while Δ ranges between 1 in JHD and 5 in HC.

5.3.2 Same Domain Different Packages (1DnP).

1DnP_Max ranges between 5 and 31 and indicates that objects that are in the same domain are often of types from different packages. The results are statistically significant for all systems except APD. The effect size is large for AFS, DL and JHD, medium for MD and PX, and small for CDB and HC, while Δ ranges between 1 and 3 in JHD.

5.3.3 Inherited Domains Percentage (InhD_P).

InhD_P ranges between 0% in CDB and 54% in JHD, and is lower for systems that do not use much inheritance such as CDB, HC and PX, and higher for systems that use inheritance and interfaces more heavily such as DL and JHD. In JHD, 30 domains inherit the same domain declaration `owned` in `AbstractCommand`. In CDB, the only inherited domain is `ITERS`, a domain that contains iterator objects: `ITERS` is declared in the class `ArrayList` and inherited by `ArrayList<LocalKey>`.

5.3.4 Inherited Points-To Edges Percentage (InhE_P).

InhE_P ranges between 0% in CDB and 12% in MD. In JHD, 30 of the 362 inherited edges trace to the field declaration `DrawingEditor myDrawingEditor` in `AbstractCommand`.

5.3.5 Pulled Objects Percentage (PO_P).

PO_P ranges between 5% in APD and 73% in DL. The mean of 39% across all systems indicates that the mismatch occurs for almost half of the abstract objects. Most of these pulled objects are in domains at the top level of the hierarchy or in public domains. This is unsurprising since objects are pulled as a result of refining an OOG. A common operation during refinement is to push an architecturally relevant object to the top of the object hierarchy, or to push some object into the public domain of some other object of which it is conceptually part.

5.3.6 Scattering Object Factor (SO_F).

SO_F_Max ranges between 75% and 97%, which means that for every system, there is at least one object that is created in many type declarations. Some objects in DL, PX, JHD, and HC are scattered across more than 10 type declarations. For DL and PX, objects with maximum scattering factor are in the `SHARED` domain and are of types from the Java library such as `String` or `Rectangle`. In JHD, an interesting example is

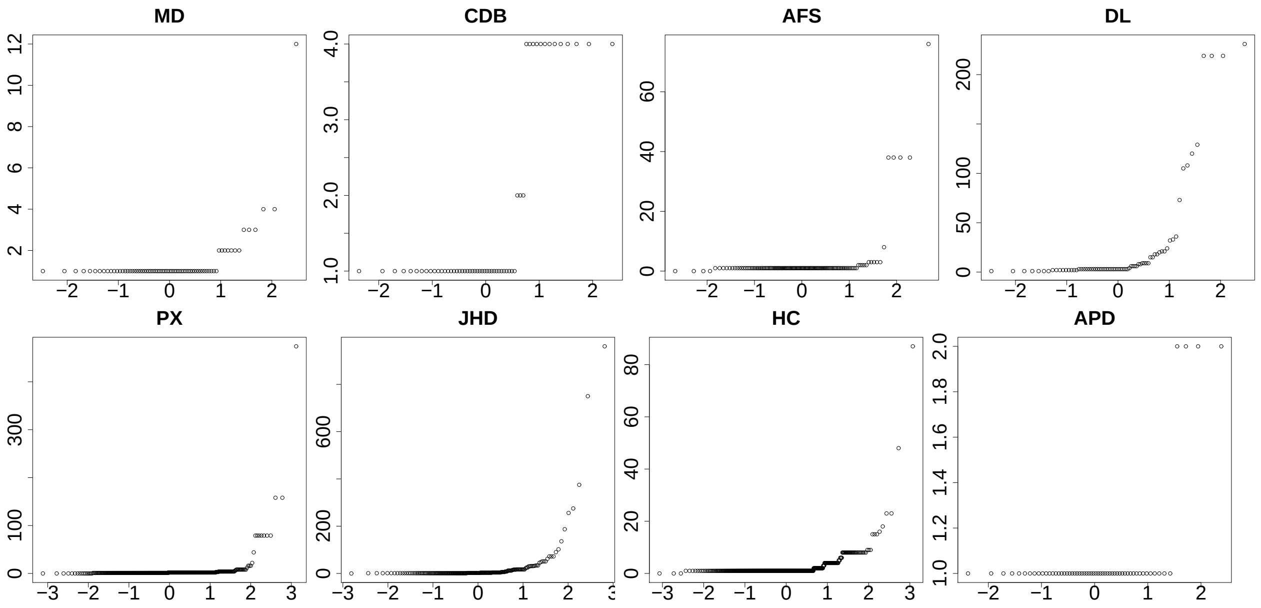

Figure 6: QQ probability plots of *1FnE* for each system show exponential distributions with high values for the upper quintile.

`UndoActivity`, which traces to code to 5 different type declarations such as `ScribbleTool`, `PasteCommand`, `PolygonTool`, `CreationTool`, and `DuplicateCommand` that implement the undo concept.

5.4 RQ4: How often does the abstract runtime structure have more precision than the code structure?

5.4.1 Points-To Edge Precision Factor (PTEP_F).

PTEP_F_Max ranges between 0 in CDB and 100% in PX, with most other systems exhibiting values over 70%. In DL, only one edge maps to the field declaration `Object<owned> observerList` where `owned` is a locally declared domain, and the destination object is of type `Vector`. But the field declaration `Object<M> model` in the class `ValueAdapter` leads to 146 edges: `M` is a domain parameter that is bound to `MODEL`, so the edges have as destinations all the objects in the `MODEL` domain.

6 Research Questions Revisited

We revisit the research questions based on the results.

6.1 RQ1: How often does one code element map to many abstract runtime elements?

The metrics to answer RQ1 (*WA*, *WAB*, *WAWB*, *HMO*, *1FnE*) indicate that one type declaration or one object creation expression in the code can lead to many objects or edges. It is rare, however, to find objects of the same type that are in sibling domains of the same parent.

For the abstract objects, the magnitude of the difference is small; for the abstract edges, the magnitude of the difference is medium or large. For *1FnE*, the probability plot shows that the metric likely follows an exponential distribution (Fig. 6). The sharp slope for the last quartile indicates that about 20% of field declarations generate a large number of points-to edges, and the difference is higher.

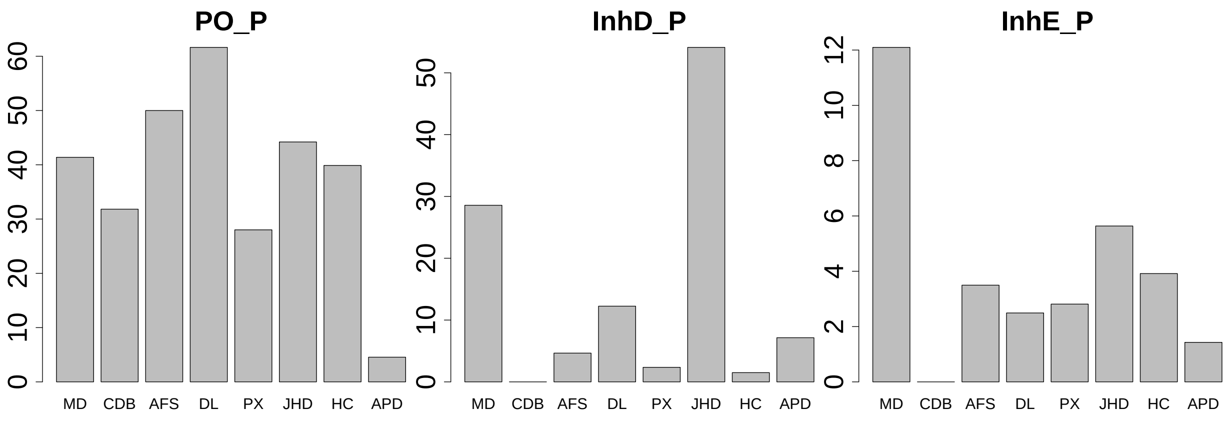

Figure 7: Percentage of objects domains and edges that mismatch code elements

6.2 RQ2: How often do many code elements map to one abstract runtime element?

The metrics to answer RQ2 (*HMN*, *TMO*, *TOS*) indicate that many code elements correspond to one object. The metrics focus on objects and show that many object creation expressions, or many types are merged by one abstract object, or appear in the substructure of one abstract object.

Overall, the RQ1 and RQ2 metrics indicate a many-to-many relation between code elements and abstract runtime elements, where the difference is small but statistically significant. Systems with a medium or large effect size for at least two metrics pose additional challenges for code exploration tools.

6.3 RQ3: How often does the location of an abstract runtime element mismatch the location of its corresponding code element?

The metrics to answer RQ3 (*1PnD*, *1DnP*, *InhD_P*, *InhE_P*, *PO_P*, *SO_F*) indicate that a mismatch often exists. The results of *1PnD* and *1DnP* indicate that packages and tiers often do not align. The way classes are grouped in packages is often different from how abstract objects of those classes are grouped into domains. For most systems, the *1PnD_Max* is reached for `java.util`, while *1DnP_Max* is reached for the `SHARED` domain or for domains at the top level of the object hierarchy. Therefore, reverse engineering approaches should not treat packages as tiers in a runtime architecture.

The high percentage of inherited domains (*InhD_P*) and edges (*InhE_P*) indicates that the mismatch is higher for systems that use inheritance more heavily, such as DL and JHD (Fig. 7).

Less than half of the objects are scattered. However, *SO_F_Median* and a high *SO_F_Max* indicate that some of these objects can be very scattered across different type declarations. We think such cases cause particular program comprehension difficulties, as maintainers must find and modify all the scattered places in the code, instead of making localized changes.

6.4 RQ4: How often does the runtime structure have more precision than the code structure?

The metric to answer RQ4 (*PTEP_F*) indicates that for all systems except CDB, the OGraph has more precision than a global type hierarchy such as the Eclipse Type Hierarchy that shows all the possible subclasses in the inheritance hierarchy. *PTEP_F_Median* is 0 for most systems and indicates that for more than half of the field declarations, it is sufficient to use the precision of the code structure. However, when the imprecision exists, it is high. In such cases, the OGraph provides up to one order of magnitude more precision as the values of *PTEP_F_Max* greater than 90% indicate.

As an internal threat to validity, an OGraph may have edges that are false positives since it is a sound approximation of the runtime structure. We categorize field declarations according to the *PTEP_F* range

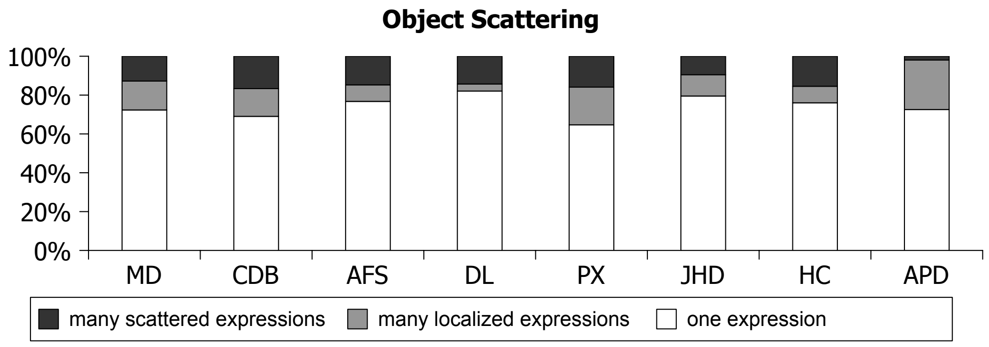

Figure 8: Percentage of scattered objects (top categories). *SO_F* threshold 50%.

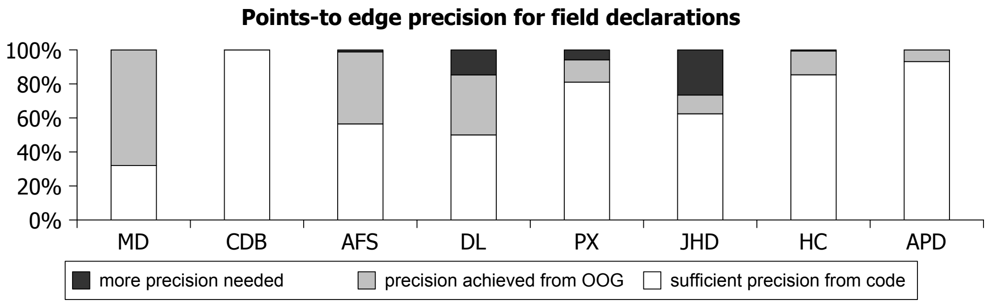

Figure 9: Percentage of field declarations that require precision of runtime structure (top two categories). *PTEP_F* threshold 50%.

and observe that more precision is needed for 5% of the field declarations in PX, 15% in DL, and 25% in JHD (Fig. 9). These results are consistent with the largest values of *1FnE_Max*. The imprecision is due to field declarations in abstract classes where the field type is an interface that is part of a deep inheritance hierarchy. The OOG relies on domain annotations to achieve precision for some of these fields. The precision is higher for fields in locally declared domains than for fields in domain parameters. *PTEP_F* highlights where refining the OOG can perhaps achieve more precision.

6.5 Limitations

Threats to Validity. An external threat to validity is that the number of subject systems in our study is lower than is typically seen in empirical studies of the code structure [BFN+06] or studies of runtime heaps using dynamic analysis [PNB04]. Those studies consist of running a fully automated analysis on a large number of systems and comparing the results across systems. Running our metrics on many more systems requires first adding annotations to the code, which is currently done manually, then extracting OOGs.

7 Qualitative Analysis of Outliers

Next, we analyzed the outliers identified by the metrics and noticed some code patterns that lead to greater differences between the code structure and the abstract runtime structure.

We broadly classified the outliers as follows: 1. Collection of elements of a general type; 2. Field of a general type; 3. Inheritance; 4. Composition; 5. Framework layering.

Collections. Typically, a collection object is in one domain, and the elements referenced by the collection are in a different domain. When the declared type of the collection elements is a general type such as an interface, this collection contributes many abstract objects and edges.

```
class C1 {
  domain OWNED;
  public domain DATA;
  Collection<OWNED, C2<DATA> > f;
}
class C2 { // C2 can be a general type
}
// E: generic type parameter
// ELTS: domain parameter for collection elements
class Collection< E<ELTS> > {
   E<ELTS> obj;
}
```

Fields of a general type. A field that is declared to be of a general type may contribute many abstract edges in the OOG. This can happen due to inheritance or due to composition, as we discuss below.

Inheritance. Many outliers are due to having a field in a base class that has many subclasses, and the subclasses are instantiated in different domains. A field `f` in class `C1` can lead to many abstract edges when a class `C2` (that inherits from `C1`) is instantiated in different domains.

```
class C1 {
  C f; // C is a general type
}
class C2 extends C1 {
}
```

Composition. Many outliers are due to having a field in a class that is composed in other classes that are instantiated in different domains. A field `f` in class `C1` can lead to many abstract edges when a class `C2` (that uses composition and declares a field of type `C1`) is instantiated in different domains.

```
class C1 {
  C f; // C is a general type
}
class C2 {
  C1 c1;
}
```

Framework layering. Frameworks try to balance reuse of design and reuse of implementation, so they organize interfaces and classes into several packages that represent layers in the code structure. A typical framework consists of a core package, a default package, and a kit package [BG97]. The core package consists of mostly interfaces and no implementation, and is considered to be the most stable. The default package supports reuse of design and code and thus has more implementation, mainly abstract classes that implement the interfaces in the core package, at least partially. The kit package supports mostly reuse of implementation through concrete classes and is the least stable. We observed this framework layering in several of the subject systems that use frameworks such as MD, JHD, and DL.

When an application uses an external framework or internally follows a framework design, fields tend to have more general declared types, namely types from the core package, so the rest of the code can still work with different concrete classes from the default package or from the kit package, as well as custom implementations of those interfaces.

8 Qualitative Analysis of Transcripts

Next, we analyzed transcripts of code modification tasks performed by 10 participants on one of the 8 subject systems on which we computed metrics, MiniDraw (MD). In previous work, we had conducted a controlled experiment [AAA12] where we asked participants to implement some tasks on MD The participants were divided into a Control group (numbered C1. . . C5), who received class diagrams, and an Experimental group (numbered E1. . . E5), who received an OOG as a diagram of the runtime structure.

Many of our participants were graduate students, several had extensive—even industrial—programming experience, as shown in [AAA12, Table III]. We had also performed an Analysis Of Covariance (ANCOVA), selecting the Java and industry experience as our covariates. The ANCOVA analysis showed that neither Java nor industry experience had a significant effect on the results [AAA12, Table XVI].

When the controlled experiment was conducted, none of the metrics discussed in this paper were designed or computed. Having now computed metrics and outliers on MD, we re-analyzed the participant transcripts, this time connecting the parts of the code identified by the outliers to areas where the participants from either group seemed to struggle. A struggle area was noted for example, where participants indicated they were unsure what to do, or where they seemed to erratically browse the code. We connected the following metrics to some program comprehension difficulties:

8.0.1 HMN.

While validating piece movements on the board, several participants struggled looking for classes and methods initializing the positions of pieces on the board. Some of the quotes by the participants during the task clearly indicate that struggle. "I don't really have an idea how to actually go about to figure out the position" [C1], "So where do we have the boardfigure [from] we have the positions [to]?" [E5], "Need to look for `toX` and `toY`, I do not know where I can find those" [E4]. Indeed, the object `p: Position` is an outlier in the HMN metric.

8.0.2 WA.

While trying to understand what constitute pieces of the board, the participants struggled locating that portion of code in the application. Here are some of the quotes by a participant [E4]. "Does board figure mean that they are the pieces? I want to make sure if it actually contains these guys [boardfigure objects], oh but there are instances of these guys inside board figure." The above quote clearly indicates that the participant struggled. Indeed, the field `fFigures: ArrayList<BoardFigure>` is an outlier in the WA metric.

8.0.3 1FnE.

Participants implementing validating piece movements on the board in the application tried to locate classes or methods that implement the movement of board figures. One quote from a participant "and where do they actually perform the movement of pieces?" [C1] clearly indicates that the he struggled to associate `BoardActionTool` that has protocols for movement of pieces and those protocols are applied on pieces of board. Indeed, the outliers of the metric 1FnE have two edges from `BoardActionTool` to `BoardDrawing` and `BoardFigure`.

8.0.4 HMO.

The participants trying to implement capture of pieces on a board game application found it hard to locate the classes or methods that initialize the black or white pieces on the board. Some quotes: "I want to figure out is there any field in boardfigure related to black or white kind of thing but where "[E1], "so if we find a piece we can check what is a piece like black or white but where do I find it?" [C3]. Indeed, the field `game: GameStub` is an outlier of the HMO metric.

9 Broader Implications

The code structure and the runtime structure are of dual nature and inherently different. We posit that code where the abstract runtime structure is significantly different from the code structure is harder to understand.

This work has implications for the following. (i) Program comprehension; (ii) Language design; (iii) Architectural extraction; (iv) Program analysis.

Program comprehension. The factoring of design into interfaces, abstract base classes, intermediate subclasses and many concrete classes tends to increase the number of types and clutter the code structure, while the runtime structure stays largely the same.

Based on the metrics (particularly, TMO), such design factoring tends to lower the comprehensibility of the code. This could be one of the contributing factors as to why frameworks have a steep learning curve [SLB00, KRW06].

Language design. Many of the outliers involve collections that are used to implement one-to-many or many-to-many relationships between objects. Programming languages that express object relationships as first-class code constructs, e.g., [BW05], instead of using collections of general types, seem to be in the right direction to improve code comprehension.

Architectural extraction. Many papers often implicitly use the term "component" to mean a "package", "module" or a collection of classes [TCL02], without recognizing that a code architecture is just one of several architecture views. This observation is corroborated by a survey of architecture extraction techniques: "Because it is complex to extract architectural components from source code, those are often simply mapped to packages or files. Even if this practice is understandable, [. . .] it limits and overloads the term component" [DP09, p. 587].

If one were to consider an abstract object with sub-structure to be an approximation of a component in a runtime architecture, (related to the metrics TMO, TOS), then architectural components are far from being groups of classes. Based on the metrics 1DnP and 1PnD, packages are not good approximations of runtime tiers either. In particular, architectural extraction approaches that group classes into components tend to not handle interfaces or collections very elegantly. For example, collections are placed in a utility module, when in fact collections contribute many interesting relations between components.

Program analysis. Several other approaches extract object graphs using a static analysis. For an object-sensitive analysis, an abstract object corresponds to a sequence of object allocation expressions that lead to the creation of the object [MRR05]. An object-sensitive analysis is parameterized by the size k of the sequence that also determines the precision of the analysis. Implementations are available for $k = 1$ [LBLH11], but for $k >= 2$ the analysis do not scale for large number of variables and objects [SBL11]. For $k = 1$, one abstract object corresponds to one object allocation expression. Based on HMO, more precision is needed in practice. At the same time, HMN and SO_F indicate that distinct object allocation expressions are abstracted by the same abstract object. This abstraction may reduce the number of abstract objects and increase the scalability of the analysis.

10 Related Work

Previous paper. This paper revises and extends an earlier workshop paper [AVA13]. This paper refines some metric definitions (e.g., for WAB and WAWB, we moved from sets of sets to more precise sets of pairs), defines new metrics, implements the metrics (the earlier paper had a partial implementation for a few metrics and a paper-design for the others), evaluates the metrics on eight systems (the earlier paper had a partial evaluation on just one of the eight systems, MD), and adds the quantitative analysis across systems as well as the qualitative analysis (they were completely missing in the earlier paper).

Metrics on statically extracted abstract object graphs. Vanciu and Abi-Antoun [VAA13] also computed other metrics on the same eight subject systems discussed in this paper. Their metrics, however, were at the level of the annotations only, or at the level of the OOG only. For example, the metrics counted the percentage of globally aliased abstract objects, or the average number of domains per abstract object and did not attempt to explicitly relate nodes or edges of the OOG back to code elements.

Other static analyses also use annotations to extract abstract object graphs. For example, the static analysis proposed by Lam and Rinard [LR03] is guided by developer-specified token annotations and extracts an object model by merging objects based on tokens. The focus there was not to compare the token graphs to the code structure.

Metrics on dynamically extracted abstract object graphs. There are other ways of extracting an abstraction of the runtime structure. For example, Marron et al. [MSSF13] abstract a dynamically extracted runtime heap by collapsing objects into conceptual components based on structural indistinguishability. They also compute some metrics like the number of types merged by an object (similar to our TMO), but did not relate the metrics back to the code structure.

Barr et al. [BBM13] do not directly compare the code structure to an abstract object graph, but their work is related in several ways. They automatically abstract runtime objects from a heap into abstract objects, and measure several properties about the abstracted heaps. The first major difference is that, while their abstraction is sound, their underlying object graphs are extracted using dynamic analysis, and thus are unsound. Not taking into account all objects and relations is likely to skew the results. The second difference is that the abstractions are automatically inferred rather than human-guided, and as a result, they identify specific types of ownership patterns such as strict domination. In particular, their system does not have the notion of logical containment, where an object is made to be part of another object only conceptually, based on arbitrary design intent. In fact, the previous work was unable to classify some relationships that did not follow the expected ownership patterns and called for more flexible annotation systems.

Metrics on dynamically extracted raw object graphs. Potanin et al. computed metrics on raw object graphs extracted from snapshots of the heap, such as the in-degree of an object [PNB04]. The dynamic heap can grow arbitrarily large, so additional post-processing is needed. Our metrics compare a finite representation, a statically extracted abstract object graph, to the code structure, another finite representation.

Metrics of the code structure or of the runtime structure. Researchers proposed a plethora of metrics [Kit10] specific to object-oriented code, such as the depth of the inheritance tree, coupling at the level of a class, method, or object [ABF04]. Such metrics are used to predict the quality of the design [Mar05] or the error-proneness of classes [BBM96]. Most metrics focused either on the code structure or on the runtime structure, and little work was done to quantify the difference between code and runtime structure. Also, more than half of the metrics are method-level rather than class or object-level [Kit10]. For example, Bavota et al. compared sets of coupling links between methods based on code structure and runtime structure and found an 84% overlap where only 13% of the code structure coupling links were covered [BDO$^+$13]. However, these results inherently depend on the quality of the test suite used to collect the metrics of the runtime structure using dynamic analysis. Our metrics are based on a sound abstraction to avoid such a limitation.

Empirical studies of language features. Many empirical studies focus on specific features of object-oriented code like inheritance [TYN13] or generics in the wild. Our metrics are more cross-cutting, focusing on measuring properties object-oriented *designs* that may combine several design patterns and several language features, rather than measuring the use of language features in isolation. For example, some metrics bring out the use of composition (TOS) in object-oriented programs, while others also deal with inheritance and subtyping (TMO).

11 Conclusion

We propose a range of measures of the inherent differences between the code and the runtime structure of object-oriented code. The results of analyzing 100 KLOC of Java code lead us to conclude that the difference is small—but still statistically significant—for objects, and medium or large for edges. Also, for most of the systems considered, a large and statistically significant mismatch exists between packages and groups of objects such that packages cannot be considered tiers. The measures are also new software design metrics that have clear relations to the underlying object-oriented language.

Finally, our results contribute a better understanding of the abstract shapes of object-oriented systems; a unique characteristic of our work is that these shapes are guided by developer design intent, using annotations, and the graphs are suitable for human consumption. Moreover, these shapes are finite representations and sound abstractions that handle key object-oriented features such as inheritance, interfaces, recursion and aliasing. Finally, our work nicely complements other empirical results measuring abstract shapes derived from dynamic heaps.

References

[AA09] Marwan Abi-Antoun and Jonathan Aldrich. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *OOPSLA*, 2009.

[AAA12] Nariman Ammar and Marwan Abi-Antoun. Empirical Evaluation of Diagrams of the Run-time Structure for Coding Tasks. In *WCRE*, pages 367–376, 2012.

[ABF04] Erik Arisholm, Lionel C. Briand, and Audun Foyen. Dynamic Coupling Measurement for Object-Oriented Software. *TSE*, 30(8), 2004.

[Abi14] Online appendix. `www.cs.wayne.edu/~mabianto/arch_metrics/`, 2014.

[AC04] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, pages 1–25, 2004.

[AVA13] Marwan Abi-Antoun, Radu Vanciu, and Nariman Ammar. Metrics to Identify Where Object-Oriented Program Comprehension Benefits from the Runtime Structure. In *Workshop on Emerging Trends in Software Metrics*, 2013.

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo. A validation of object-oriented design metrics as quality indicators. *TSE*, 22(10), 1996.

[BBM13] Earl T. Barr, Christian Bird, and Mark Marron. Collecting a heap of shapes. In *ISSTA*, 2013.

[BDO$^+$13] Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimilano Di Penta, Denys Poshynanyk, and Andrea De Lucia. An empirical study on the developers perception of software coupling. In *ICSE*, 2013.

[BFN$^+$06] 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*, pages 311–324, 2006.

[BG97] Kent Beck and Erich Gamma. JHotDraw – Patterns Applied (Tutorial). In *OOPSLA*, 1997.

[BW05] Gavin Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *ECOOP*, 2005.

[Coh88] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 2nd edition, 1988.

[DP09] Stéphane Ducasse and Damien Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *TSE*, 35(4):573–591, 2009.

[JW01] Daniel Jackson and Allison Waingold. Lightweight Extraction of Object Models from Bytecode. *TSE*, 27(2):156–169, 2001.

[Kit10] Barbara A. Kitchenham. What's up with software metrics? – A preliminary mapping study. *J. Systems & Software*, 83(1), 2010.

[KRW06] D. Kirk, M. Roper, and M. Wood. Identifying and Addressing Problems in Object-Oriented Framework Reuse. *Empirical Software Engineering*, 12(3):243–274, 2006.

[LBLH11] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, 2011.

[LR03] Patrick Lam and Martin Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP*, 2003.

[Mar05] R. Marinescu. Measurement and quality in object-oriented design. In *ICSM*, 2005.

[Mit06] Nick Mitchell. The Runtime Structure of Object Ownership. In *ECOOP*, pages 57–64, 2006.

[MRR05] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-To Analysis for Java. *TOSEM*, 14(1):1–41, 2005.

[MSSF13] Mark Marron, Cesar Sanchez, Zhendong Su, and Manuel Fähndrich. Abstracting runtime heaps for program understanding. *TSE*, 39(6):774–786, 2013.

[PNB04] Alex Potanin, James Noble, and Robert Biddle. Checking Ownership and Confinement. *Concurrency and Computation: Practice and Experience*, 16(7):671–687, April 2004.

[SBL11] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL*, 2011.

[SLB00] Forrest Shull, Filippo Lanubile, and Victor R. Basili. Investigating Reading Techniques for Object-Oriented Framework Learning. *TSE*, 26(11):1101–1118, 2000.

[TCL02] Roseanne Tesoriero Tvedt, Patricia Costa, and Mikael Lindvall. Does the Code Match the Design? A Process for Architecture Evaluation. In *ICSM*, pages 393–401, 2002.

[TYN13] Ewan Tempero, HongYul Yang, and James Noble. What Programmers Do with Inheritance in Java. In *ECOOP*, 2013.

[VAA13] Radu Vanciu and Marwan Abi-Antoun. Object Graphs with Ownership Domains: an Empirical Study. In *State-of-the-art Survey on Aliasing in Object-Oriented Programming*, LNCS 7850. Springer-Verlag, 2013.