# Metrics to Identify Where Object-Oriented Program Comprehension Benefits from the Runtime Structure

Marwan Abi-Antoun    Radu Vanciu    Nariman Ammar

Department of Computer Science, Wayne State University

***Abstract*—To evolve object-oriented code, developers often need to understand both the code structure in terms of classes and packages, as well as the runtime structure in terms of abstractions of objects.**

**Recent empirical studies have shown that for some code modifications tasks, developers do benefit from having access to information about the runtime structure. However, there is no good sense of when object-oriented program comprehension clearly depends on information about the runtime structure.**

**We propose using metrics to identify cases in object-oriented program comprehension that benefit from information about the runtime structure. The metrics relate properties observed on a statically extracted hierarchical object graph to the type structures declared in the code and highlight key differences between the runtime structure and the code structure.**

***Index Terms*—object-oriented runtime structure; metrics**

## I. Introduction

Software maintenance accounts for 50% to 90% of the costs over the life-cycle of a software system. One major activity during maintenance, program comprehension, absorbs around half of the costs [1]. For object-oriented comprehension, it is believed that it is at least as important—possibly more important—that developers understand the runtime structure in terms of abstractions of objects and their relations, as to understand the code structure in terms of classes and packages. Yet, there is no good sense of when the distinction between the code structure and the runtime structure is crucial.

To help with program comprehension, there is no shortage of tools of the code structure [2]. On the other hand, the runtime structure is notoriously difficult to reverse-engineer from object-oriented code [3]. Several tools to help developers explore and understand the runtime structure have been proposed, but there is no good sense of when these tools offer developers a huge improvement over widely used tools of the code structure such as class diagrams.

The evaluation of tools of the runtime structure is still immature. A few controlled experiments have measured their impact on comprehension, using the usual variables of time and effort. There is no good sense, however, why information about the runtime structure helps for some tasks but not for others [4], or for some subject systems but not for others [5]. One major confound with experiments is that some participants do not like to use novel exploration tools or diagrams that may use a different notation than widely used tools of the code structure. Moreover, some of these tools have a steep learning curve, which makes them less suitable for time-limited experiments.

In this work, we define metrics to quantify the differences between the runtime structure and the code structure, across a range of systems. In general, one element in the code structure (e.g., one class) can generate multiple entities in the runtime structure (e.g., many objects of the same class). Moreover, many elements in the code structure (e.g., many references) can refer to the same entity in the runtime structure (e.g., the same object) [6]. Admittedly, if a system does not use inheritance or subtyping, or is designed such that there is only one instance of each class at runtime, then the differences between the code and the runtime structure will be small.

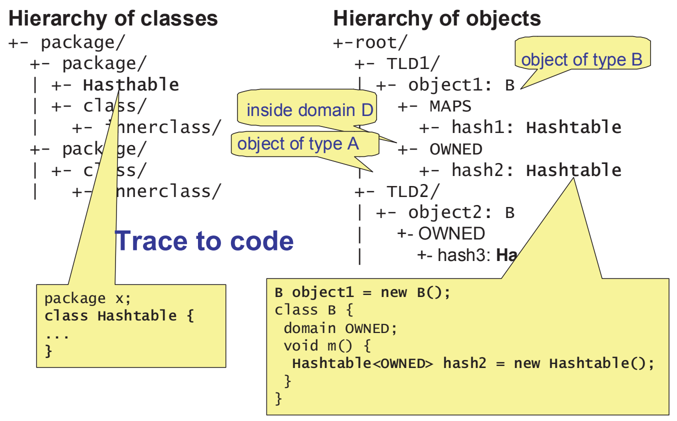
**Outline.** The rest of this paper is structured as follows. Section II describes our program comprehension theory focused on the runtime structure. Section III places our theory on formal foundations. Section IV presents some of our proposed metrics. Section V shows our preliminary evaluation. Next, we discuss related work (Section VI) and conclude.

## II. Theory of Program Comprehension

**Instances matter in object-oriented code.** When evolving object-oriented code, developers must understand both the code structure and the runtime structure. In object-oriented design patterns, much of the functionality is determined by what instances point to what other instances. For example, in the Observer design pattern [7], understanding "what" gets notified during a change notification is crucial for understanding the behavior of the system, but "what" does not usually mean a type, "what" means an instance.

**Do specific instances really matter for program comprehension?** When developers need very specific instances, they can use a debugger. For program comprehension, however, it seems that *specific* instances do not matter. Indeed, many approaches merge related objects in an object graph [8], or collapse objects underneath other objects [9]. What seems to matter then is describing an object's *role* and merging objects that have the same role.

**Code vs. Runtime Structure.** Overall, tools of the code structure allow developers to explore a *hierarchy of classes*, where classes are organized by packages. In contrast, tools of the runtime structure allow developers to explore a *hierarchy of objects* (Fig. 1). One tool to explore the runtime structure, previously proposed by Abi-Antoun and Aldrich [10], statically extracts from a program with annotations a hierarchical object graph, the Ownership Object Graph (OOG), where



```
+- package/
  +- package/
  | +- Hasthable
  | +- class/
  |   +- innerclass/
  +- package/
  | +- class/
  |   +- innerclass/
```

```
+-root/
  +- TLD1/
  | +- object1: B
      +- MAPS
        +- hash1: Hashtable
      +- OWNED
        +- hash2: Hashtable
  +- TLD2/
  | +- object2: B
  |   +- OWNED
  |     +- hash3: Ha
```

```
package x;
class Hashtable {
...
}
```

```
B object1 = new B();
class B {
 domain OWNED;
 void m() {
  Hashtable<OWNED> hash2 = new Hashtable();
 }
}
```

Fig. 1. Hierarchy of classes vs. Hierarchy of objects.

architecturally significant objects appear near the top of the hierarchy and data structures are further down. This hierarchy enables both high-level understanding and detail, compared to flat object graphs extracted by other tools that do not rely on annotations [11], [12]. The OOG imposes a hierarchy on objects and groups objects that are related into named, conceptual groups called *domains*.

**The OOG merges objects of the same** $type$ **that are in the same** $group$ **at the same** $hierarchy$ **level.** If despite merging objects, an OOG still holds enough precision and is useful for program comprehension as our preliminary evidence suggests [4], an instance may not matter in terms of "the particular object". It seems enough to pin things down just to objects of a $type$ that are within a $group$. This leads us to revisit the question regarding if specific instances matter.

The OOG can express *"an object of type A in domain D in an object of type B"*, that we show as the triplet $\prec A, D, B \succ$. So the question becomes: How often does it matter to distinguish the role of an object not just by $type$ (as in a hierarchy of classes), but by named $groups$ (domains) and by position in an object $hierarchy$ that dictates parent-child relationships between objects?

We believe that the OOG was useful for two out of three coding tasks in our controlled experiment [4], precisely because there are situations where the $type$ is not enough, and information about $type + hierarchy + group$ can describe the role of an object, or relations between objects, more precisely than $type$ alone. The following is an excerpt from the think-aloud transcript of one participant in our study:

> *I mean any of these are really a possibility of where it might have all the positions of all the pieces. I guess I should be looking for some sort of a data structure.*
> Participant C5 (control group) [4]

Indeed, the code uses many `HashMap` instances, and the one that the developer really needed is the `HashMap` in the `MAPS` domain in the object of type `BoardDrawing`, i.e., the triplet $\prec$`HashMap, MAPS, BoardDrawing`$\succ$ on the OOG. By analyzing a corpus of object-oriented code, we will measure how often the $type$ is not enough to identify an object, or to identify relations between objects.

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

Fig. 2. Portions of the Ownership Domains abstract syntax [13].

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

Fig. 3. Core data type declarations for the OGraph.

## III. Formal Foundations

The OOG is extracted with a static analysis that uses abstract interpretation of the code with annotations that implement the Ownership Domains type system [13]. So we base the theory of program comprehension in the Ownership Domains type system and the OOG semantics.

**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.

Classes are parameterized by a list of domain parameters, and extend 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 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 and ODomains. Edges between OObjects, OEdges, correspond to field reference points-to relations.

With $C{<}\overline{D}{>}$, the OOG can distinguish between different instances of the same class $C$ that are in different domains, even if created at the same object creation expression (`new`). 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 $C{<}\overline{D}{>}$ instead of just a class $C$ and an owning ODomain. We consider an OObject's owning ODomain as the first element $D_1$ of $\overline{D}$. As a result of the aliasing precision provided by the annotations, the OOG avoids merging objects excessively. It merges two objects of the same class only if all 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 $\ell$ and $\ell'$ 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 the code can correspond to multiple ODomains $D_i$ in the OGraph and the fresh identifier $D_{id}$ ensures that multiple $D_i$ can be created for one domain $d$ declared in the code.

**Points-to analysis.** The OOG extraction analysis is a kind of a points-to analysis, a fundamental static analysis to determine the set of objects whose addresses may be stored in variables or fields of objects.

A common idea in points-to analysis is to merge all the objects created at the same object creation expression into an equivalence class. A basic points-to analysis attaches an object label, $h$, at *each* object creation expression, `new` $A()$ (line 1).

$$\texttt{new}^h \ A() \qquad (1)$$
$$\texttt{new}^{L_A} \ A{<}p_{owner}, p_{params}...{>}() \qquad (2)$$
$$(O_C, p_{owner}) \mapsto D \qquad (3)$$
$$\exists O_B \text{ s.t. } (O_B, d) \mapsto D \text{ and } O_B = \langle B{<}\overline{D_B}{>}\rangle \qquad (4)$$

The OOG extraction analysis analyzes an object creation expression attaching to it the label $L_A$ (line 2). The first parameter, $p_{owner}$, is the owning domain, and $p_{params}$ is a (possibly empty) sequence of additional domain parameters. Each formal domain parameter $p_i$ is bound to some actual domain. During abstract interpretation, the static analysis tracks the bindings of each $p_i$ and maps $p_i$ to an ODomain $D_i$ in some analysis context. In particular, in some analysis context $O_C$, the analysis binds $p_{owner}$ to some ODomain $D$ (line 3), where $D$ is the child domain of some OObject $O_B$ of type $B$ and actual domains $\overline{D_B}$ (line 4).

For the object creation expression, the analysis creates an OObject $O_A = \langle\ A{<}\overline{D_A}{>}\ \rangle$, which is labeled with $L_A$. We can represent the label $L_A$ as the triplet $L_A = \prec A, d, B \succ$, which serves as the basis of our metrics.

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 to the same $\overline{D}$, whereas a basic points-to analysis will create multiple $h$ labels. Second, if the analysis context maps the domain parameters to $n$ different $\overline{D}$, then one object creation expression of type $A$ may create $n$ different $L$ labels.

For program comprehension, we believe that developers often do not want to see multiple instances that are equivalent or have the same role. Instead, we believe that it is useful to assign different roles to objects that have different $L$ labels.

The analysis then adds edges between objects. When analyzing a field declaration $T f$ in class $C$ where $T = C'{<}\overline{p'}{>}$, in a given analysis context, the analysis maps the owning domain $p'_1$ to an ODomain $D$. It then looks up in $D$ each OObject $O_t$ of type $C_t$, where $C_t$ is a subtype of $C'$. It then creates an OEdge with the source being the OObject corresponding to the current object (the one for the `this` object), and the destination being 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, just because class $C$ has a field of type $C'$ [12].

## IV. Metrics

In this section, we define metrics to measure interesting facts about the $\prec A, D, B \succ$ triplets obtained from an OOG. The metrics relate properties observed on the OOG to the type structures declared in the code and highlight the differences between the runtime structure and the code structure.

### A. Metrics Related to Objects

Every OObject $O = \langle A{<}\overline{D}{>}\rangle$ has a unique associated triplet $\prec A, D, B \succ$. For brevity, we refer to an O as "object $A$" or "an object of type $A$", with the meaning an object of type $A$, where $D$ is the owning ODomain of $O$. Also, for an ODomain $D = \langle D_{id}, d\rangle$, we use $D$ to refer to the domain declaration $d$ or to the ODomain, interchangeably.

**Metric: *Which-A-in-Which-B*.** A diagram of the runtime structure such as an OOG can show different objects of the same type $A$. On the other hand, a diagram of the code structure such as a class diagram will show just one box for the class $A$. Typically, different instances of the same type serve often different conceptual purposes in a design. *Which-A-in-Which-B* measures how frequently different objects of the same type $A$ are in different parent objects. For brevity, if a metric measures the size of a set, we define only the corresponding set. In this case, we measure the size of the set of distinct triplets that satisfy the following condition:

$$\prec A_i, D_i, B_i \succ, \prec A_j, D_j, B_j \succ$$
$$\text{where } A_i = A_j \text{ and } B_i \neq B_j$$

**Metric: *Which-A-in-B*.** An OOG can also express design intent using multiple domains per object to distinguish between the different parts of an object's substructure. *Which-A-in-B* measures how frequently different objects of the same type $A$ are in different domains, but in the same parent object $B$.

$$\prec A_i, D_i, B_i \succ, \prec A_j, D_j, B_j \succ$$
$$\text{where } A_i = A_j \text{ and } B_i = B_j \text{ and } D_i \neq D_j$$

**Ranges.** For the metrics *Which-A-in-Which-B* and *Which-A-in-B*, we are interested in sets that have at least two elements. The larger the set is, the more often instances of the same type are used in different contexts and with different roles.

**Object that appear in parent objects of types that do not directly create them.** On the OOG, each object that is assigned to a domain must appear where that domain is declared. An object $A$ can appear in a domain $D$ that is inside some object of type $B$. In the code, however, the class $B$ may not directly create an object of type $A$. Instead, an object creation expression of the object $A$ may be in a class $C$ that is different than $B$. This can be due to either pulled objects or inherited domains, as discussed below.

```
class ArrayList<OWNER, T<ELTS>> {//generic type T
  T<ELTS> obj;
}
class Board<OWNER,Df> {
  domain OWNED;
  ArrayList<OWNED,Figure<Df>>  fgrs = ...;
  Figure<Df> f = new Figure<Df>();
}
class Main{
 domain DATA,CTRL;
 Board<CTRL,DATA> b = new Board<CTRL,DATA>();
}
```

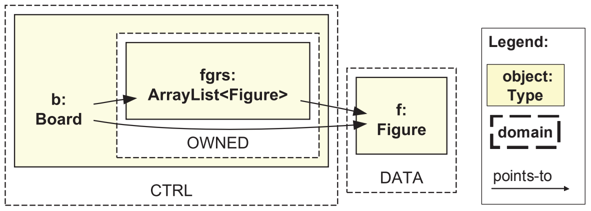


Fig. 4. The object of type `Figure` is created in class `Board`. The class `Figure` is instantiated in a formal domain parameter `Df` of class `Board`, but the object `Figure` appears in the domain `DATA` because `Df` is bound to `DATA`.

The next two metrics, *PulledObjects* and *InheritedDomains*, measure how the runtime structure and the code structure can differ. The OOG places object based on containment, ownership, and type structures, rather than according to where the objects are syntactically declared in the code, some naming convention or a graph clustering algorithm.

**Metric: *PulledObjects*.** 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 $B$.

More formally, let $params(C)$ be the list of formal 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 $O$ as $\prec A, D, B \succ$ be an OObject of type $A$. If an object creation expression of $A$ is in a class $C$ in a formal domain parameter $D_f$, then $C$ is in the $declaringTypes(O)$ (Fig. 4), of which there can be several. If $D_f$ is later bound to a local domain $D$ on a class $B$, then the object of type $A$ represents an object that was pulled from the domain $Df$ to the domain $D$ in an object of type $B$.

$$\begin{aligned}&O \text{ as } \prec A, D, B \succ \text{, where } C \in declaringTypes(O) \text{ and } B \neq C\\ &A \text{ is declared in } D_f\text{, where } D_f \in params(C)\\ &\text{and } D_f \text{ is bound to } D\text{, where } D \in domains(B)\end{aligned}$$

The same domain parameter can be bound to different domains. For example, consider a simplified version of the `ArrayList<T>` class, which uses a generic type parameter `T` (Fig. 4). The elements of the `ArrayList` may be of any type, where the element type is specified at the array creation expression. To express that the elements can be in a domain other than a locally declared domain, `ArrayList` takes a domain parameter `ELTS`. When the `ArrayList<T>` is instantiated, `T` is bound to the type `Figure`, and `ELTS` is bound to the domain parameter `Df` of `Board`, which in turn is bound

```
class C {
 domain D;
 A<D> f = new A<D>();
}
class B extends C {  // domain D is inherited
}
```

Fig. 5. The domain $D$ is inherited from $C$ to $B$. The OOG shows an object of $A$ in the domain $D$ of the parent object $B$ rather than $C$.

to the actual domain `DATA` at the object creation expression of `Board` in class `Main`.

The object of type `Figure` is pulled to the domain `DATA` in the object `Main`, although the object `Figure` is instantiated in a domain parameter in class `Board`. On the other hand, the object of type `ArrayList` is not pulled because is instantiated in a locally declared domain in class `Board`.

**Metric: *InheritedDomains*.** Inheritance makes object-oriented programs harder to understand, since developers must flatten the inheritance hierarchy to understand the program, e.g., to include any inherited fields.

An object of type $A$ that is not created in a class $B$ may appear in an object of type $B$ in an ODomain $D$, 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 $B$ (Fig. 5).

$$\begin{aligned}&O \text{ as } \prec A, D, B \succ \text{, where } C \in declaringTypes(O)\\ &O \text{ is in } D\text{, where } D \in domains(C) \text{ and } B <: C\end{aligned}$$

**Metric: *ScatteringFactor*.** Multiple elements in the code structure, e.g., multiple object creation expressions, may correspond to the same entity in a diagram of the runtime structure that employs abstraction. For one object $O$ represented by the triplet $\prec A, D, B \succ$, the code may have multiple object creation expressions `new A()` scattered across multiple files.

*Scattering* indicates the number of distinct classes that contain object creation expressions for the same type, as the size of 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. This metric is defined on a finer granularity than the *Average Scattering of Objects* metric in related work [14], which is defined as the set of files that contain an object creation expression of $A$. We also define the *ScatteringFactor* for an object.

$$scattering(O) = |declaringTypes(O)|$$
$$scatteringFactor(O) = 1 - \frac{1}{scattering(O)}$$

**Ranges.** If the *scatteringFactor* of an object is high, developers have to understand and modify many different places in the code in order to handle the same situation. The *scatteringFactor* is zero if a developer has to visit only one type declaration to understand how an object is created.

## *B. Metrics Related to Edges*

The next set of metrics are computed on points-to edges in the OGraph. A points-to edge in the OOG, an OEdge, is from a source object $O_{src}$ to a destination object $O_{dst}$.

The source and destination objects can be represented as two triplets, so a points-to edge is from an object of type $A_{src}$ to an object of type $A_{dst}$, where the source corresponds to the triplet $\prec A_{src}, D_{src}, B_{src} \succ$, and the destination corresponds to the triplet $\prec A_{dst}, D_{dst}, B_{dst} \succ$.

An OEdge corresponds to a field declaration $C\langle\overline{p}\rangle f$ in some class in the code. The field declaration may be in the class $A_{src}$, or one of its super-classes, in which case the points-to edge is due to inherited fields. Also, $A_{dst}$ must be a subtype of $C$. Since not all the concrete subtypes of $C$ correspond to $A_{dst}$, we measure the precision of a points-to edge.

**Metric: *EdgeInheritance.*** Points-to edges that involve inherited fields may be harder to understand from looking only at the code because a developer typically inspects one class at a time, and also has to understand the inheritance hierarchy. When using a watch window in a debugger, the developer sees all the fields on an object including the inherited ones.

The field declaration corresponding to an edge may not be in the set of 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 $A_{src}$. *EdgeInheritance* is the size of the set of points-to edges that involve inherited fields.

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

**Ranges.** A value 0 indicates that developers can understand all field references by inspecting one class at a time. A higher value corresponds to a system that requires additional effort to understand the field references from looking only at the code.

**Metric: *EdgePrecision.*** In object-oriented code, a declared field can be of an interface type $C$. The interface in turn can be implemented by an abstract base class and several concrete subclasses which can have a deep inheritance tree. A field is ultimately a reference to some object, and the developer needs to locate where that object is created and what is the concrete type of that object. When searching through the code, the developer may find a new expression that is used to initialize a field in the field declaration, but the field may be reassigned elsewhere in the code. A diagram of the code structure or a type hierarchy may show all possible concrete subclasses of $C$, and the developer may need to inspect all of them.

In a points-to edge, $A_{dst}$ must be a subtype of $C$, the class of $f$ according to the field declaration. This metric captures the points-to edges on the OOG, where the number of all possible concrete subclasses of $C$ ($|AllPossibleSubClasses(C)|$) is greater than the number of concrete 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 $D_{dst}$ ($OOGPossibleSubTypes(C, D_{dst})$), after binding formal to actual domains. Thus, the OOG shows points-to edges to only a subset of all possible objects of type $C$ (Fig. 6).

```
class Asrc {
  domain DDST;
  C<DDST> f;
}
class Adst extends C{
}
class Main{
  domain DATA;
  Asrc<DATA> a = new Asrc<DATA>()
  C<DATA> c = new C<DATA>();
  a.f = new Adst<a.DDST>()
}
```

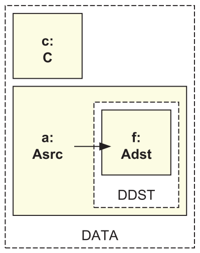


Fig. 6. The OOG shows a points-to edge ($E$) from the object `a:Asrc` to `f:Adst` in the `DDST` domain, but not to `c:C` in the `DATA` domain ($precision(E, C) = 0.5$).

We define $precisionRatio$ as the ratio between the two sets associated with an OEdge $E$. Since a lower ratio indicates a higher precision, we also define the $precision$ of $E$.

$$\begin{aligned}
&E\ \text{as}\ \langle \prec A_{src}, D_{src}, B_{src} \succ, \prec A_{dst}, D_{dst}, B_{dst} \succ, f \rangle\\
&C\langle\overline{p}\rangle\ f \in fields(A'_{src})\ \text{s.t.}\ A_{src} <: A'_{src}\ \text{and}\ C <: A_{dst}\\
&precisionRatio(E) = \frac{|OOGPossibleSubTypes(C, D_{dst})|}{|AllPossibleSubClasses(C)|}\\
&precision(E) = 1 - precisionRatio(E)
\end{aligned}$$

**Ranges.** The *precision* is zero if a developer can use only the type hierarchy to find all the possible references. We are interested in cases where *precision* is above a certain threshold value. For example, the *precision* is 0.5 if the developer has to consider only one of the two existing subclasses (Fig. 6). For those cases, developers may benefit from using information about the runtime structure.

**Metric: *EdgeLifting.*** As a hierarchical representation, the OOG supports collapsing and expanding the substructure of objects. When objects are collapsed, some edges may be lifted to the nearest visible ancestors. These lifted edges help developers get a high-level understanding of the relations between the objects. Developers can also get a more detailed understanding, by expanding the substructure of an object to find the direct points-to edge that is responsible for the lifted edge. Lifted edges are commonly used with objects that represent collections such as `LinkedList` or `HashMap`. The implementation details of the collection (e.g., nodes or entries) are often not shown, although these objects have a direct field reference to the elements stored in the collection. Instead, a lifted points-to edge is shown from the collection object to an object representing the elements of the collection.

At the top level, some of the edges between objects represent direct field references, while others are lifted edges. A developer selecting a lifted edge and tracing to the code may be taken to a field declaration in a class different from the one she expected. Edge lifting is defined as follows: If node $r$ has an edge to node $q$, and $r$ is a descendant of $o$ and $q$ is a descendant of $o'$, then we lift the edge $(r, q)$ to $(o, o')$ only if $o$ and $o'$ are distinct nodes and $o$ is not a descendant or ancestor of $o'$. Consider a lifted edge labeled $s$ between $o$ and $o'$ (Fig. 7). The metric *EdgeLifting* measures the number of lifted edges shown when the objects that appear near the top of the hierarchy have their substructure collapsed.

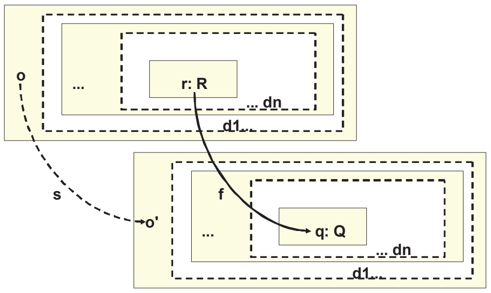


Fig. 7. Lifted edge, used in a nested-box visualization of an OOG.

$\langle r, q, f \rangle \triangleright \langle \prec A_{src}, D_{src}, B_{src} \succ, \prec A_{dst}, D_{dst}, B_{dst} \succ, s \rangle$
iff
$\nexists C$ s.t. $C{<}\overline{p}{>}\ s \in fields(A'_{src})$ s.t. $A_{src} <: A'_{src}$ and $C <: A_{dst}$
and
$(r : R \in descendants(o : A_{src})$ s.t. $C{<}\overline{p}{>}\ f \in fields(R')$
s.t. $R <: R'$ and $A_{dst} <: C$
or
$q : Q \in descendants(o' : A_{dst})$ s.t. $C{<}\overline{p}{>}\ f \in fields(A'_{src})$
s.t. $A_{src} <: A'_{src}$ and $Q <: C)$
$\langle r, q, f \rangle \triangleright^* \langle o, o', s \rangle$
$r \in descendants(o)$ and $q \in descendants(o')$
$o \neq o'$ and $o \notin descendants(o')$ and $o' \notin descendants(o)$

**Ranges.** The higher the number of lifted edges is, the more significant is the difference between the code structure and the runtime structure, because the field declarations that contribute these edges are not where they seem to be.

## V. Preliminary Results

In our preliminary evaluation, we computed these metrics on one subject system, MiniDraw [15], [16], the same system we used in our controlled experiment [4]. MiniDraw is a pedagogical object-oriented framework that consists of around 1,400 lines of Java code, 31 classes and 17 interfaces.

For example, the results for *Which-A-in-B* in Table I highlight multiple instances of the types `HashMap` and `ArrayList` that are in different domains, and a developer often needs to distinguish between them.

The results for *EdgePrecision* in Table II show an example where 7 concrete classes implement the interface `Tool`. The `SelectionTool` declares a field `fChild` of `Tool` in the `TRACKERS` domain. This same field declaration corresponds to 3 points-to edges, which means that the precision of these edges is 0.57, and the developer may benefit from the runtime information, when her task involves understanding the relations of a `SelectionTool` instance.

TABLE I
$\prec A, D, B \succ$ triplets from MiniDraw that satisfy the metric *Which-A-in-B*, grouped by the raw type A.

| A | D | B |
|---|---|---|
| **ArrayList** | | |
| `ArrayList<FigureChangeListener>` | `owned` | `BoardDrawing` |
| `ArrayList<Figure>` | `owned` | `BoardDrawing` |
| `ArrayList<BoardFigure>` | `MAPS` | `BoardDrawing` |
| **HashMap** | | |
| `HashMap<Position,List<BoardFigure>>` | `MAPS` | `BoardDrawing` |
| `HashMap<String,BoardFigure>` | `owned` | `BoardDrawing` |

TABLE II
Example of points-to edges that are more precise in the MiniDraw OOG ($precision = 0.57$).

| **Field Declaration** ($C{<}\overline{p}{>}f$) | $\prec A_{src}, D_{src}, B_{src} \succ$ |
|---|---|
| `Tool<TRACKERS> fChild` | $\prec$`SelectionTool`, `CTRL`, `BreakThrough`$\succ$ |

| $AllPossibleSubClasses$ **(Tool) (7)** | $OOGPossibleSubTypes$ **(Tool, TRACKERS) (3)** |
|---|---|
| `NullTool`, `SelectAreaTracker`, `BoardActionTool`, `SelectionTool`, `DragTracker`, `SelectionTool`, `DragTracker` | `NullTool`, `SelectAreaTracker`, `DragTracker` |

**Future work.** We will next compute the metrics on a corpus of object-oriented code. We already have 7 subject systems totaling over 100 KLOC [14] to which we have added annotations and from which we extracted OOGs. We believe that our empirical data will show that occurrences of interesting cases of our metrics are reasonably common, and that such cases correspond to hard program comprehension questions with which developers struggle during code modification tasks.

**Will we find mostly data structures?** The *Which-A-in-B* metric we computed on MiniDraw highlighted mostly data structures such as `ArrayList` and `HashMap`. In another system, the Apache FtpServer [14], we found 4 different instances of `FileInputStream` that are used in different contexts.

**Will the generic type be enough?** On MiniDraw, we had to use the "raw type", rather than the full generic type, to find more interesting cases of *Which-A-in-B* and *Which-A-in-Which-B*. This is unsurprising since generic types also convey design intent. Other research explored the relations between generic types and ownership types such as Ownership Domains. In general, generics and ownership are orthogonal. We will measure how often objects of the same generic type are in different domains.

## VI. Related Work

**Theories of program comprehension.** Several papers discuss questions that programmers ask and state that some questions have inadequate tool support. For example, LaToza and Myers [17] conducted a survey asking developers to report on the "hard-to-answer questions" about code. Some of the questions seem related to what the metrics are trying to identify, such as "How is this object different from that object?"

In this work, we focus on the challenges in understanding the runtime structure of object-oriented code. In particular, we are interested in measuring how often the runtime structure contains key facts that cannot be easily learned from information sources that are based on the code structure.

**Defining an object's role.** Several theories have defined their own notion of an object *role*. Reenskaug [18] was among the first to suggest the conceptual grouping of objects, role modeling. A role diagram shows how a group of collaborating objects, each playing one or more roles, achieve a common goal. Based on the role models suggested by Reenskaug, Riehle [19] enriched class diagrams with information about design patterns using "collaboration roles" and found in several case studies that the role modeling adds more information to the existing documentation.

Demsky and Rinard [20] used dynamic analysis and the aliasing properties of objects to define a set of roles, where each role represents an abstract object state that can be interesting to developers. Nodes on their diagrams represent abstract object states, and edges represent method executions that take these objects as parameters or methods that change the roles of objects.

**Merging objects with the same role.** Many researchers share our view that it is often unnecessary to pin things down to the level of individual objects. For example, Marron et al. [21] abstract a dynamically extracted concrete runtime heap structure. The approach collapses objects into conceptual components based on structural indistinguishability.

**Metrics on statically extracted object structures.** In previous work, Vanciu and Abi-Antoun [14] computed various metrics on extracted OOGs from seven subject systems totalling over 100 KLOC. The metrics were at the level of the annotations only, or at the level of the OOG only, such as the average number of public domains per object. The metrics focused on the expressiveness of the Ownership Domains type system (i.e., which features were used and how heavily), the quality of the annotations, and the quality of an extracted OOG, namely, that the OOG does not show many objects at the top level, compared to a flat object graph for the same system. In this work, we tie each property we measure on the OOG back to the code. For example, how many edges on the OOG can be traced back to the same field declaration in the code.

**Metrics on dynamically extracted object graphs.** Previous work has computed metrics on raw object graphs extracted from snapshots of the heap, such as the in-degree of an object. Heap snapshots do not convey much design intent. In this work, we compute metrics on abstracted object graphs that convey design intent.

**Tool design.** Together with our own tools, there is a growing trend of tools that expose information about the runtime structure to developers [22]. Our metric-based approach can inform the design of such tools by identifying the key areas where developers struggle the most with the information content provided by the current tools that still emphasize the code structure, or where information about the runtime structure is most helpful while they perform code modification tasks.

## VII. Conclusion

In our previous experiment, developers who had access to information of the runtime structure, that included information about $type + hierarchy + group$ spent less time and explored fewer code elements than developers who used information from the code structure that provided only $type$ information. We propose to use metrics to better understand how often this distinction between the code structure and the runtime structure matters

Our results will identify information gaps in the current tools that focus solely on the code structure, and identify where developers will benefit the most from using information about the runtime structure. Given the considerable costs of software maintenance, giving developers tools that go beyond simply exposing information from the code structure will have a significant impact on software engineering practice.

## Acknowledgements

The authors thank Jonathan Aldrich for insightful comments on an earlier version of this work. Vanciu and Ammar were supported by Wayne State University. Abi-Antoun was supported by his faculty startup fund at Wayne State University.

## References

[1] K. H. Bennett, V. Rajlich, and N. Wilde, "Software evolution and the staged model of the software lifecycle," *Advances in Computers*, vol. 56, pp. 3–55, 2002.

[2] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zundorf, "A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering," in *WCRE*, 2002, pp. 22–32.

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

[4] N. Ammar and M. Abi-Antoun, "Empirical Evaluation of Diagrams of the Run-time Structure for Coding Tasks," in *WCRE*, 2012, pp. 367–376.

[5] J. Quante, "Do dynamic object process graphs support program understanding? - a controlled experiment," in *ICPC*, 2008, pp. 73–82.

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

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

[8] E. Barr, C. Bird, and M. Marron, "Collecting a heap of shapes," Tech. Rep., 2012.

[9] T. Hill, J. Noble, and J. Potter, "Scalable Visualizations of Object-Oriented Systems with Ownership Trees," *Journal of Visual Languages and Computing*, vol. 13, no. 3, pp. 319–339, 2002.

[10] M. Abi-Antoun and J. Aldrich, "Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations," in *OOPSLA*, 2009, pp. 321–340.

[11] D. Jackson and A. Waingold, "Lightweight Extraction of Object Models from Bytecode," *TSE*, vol. 27, no. 2, pp. 156–169, 2001.

[12] A. Spiegel, "Automatic Distribution of Object-Oriented Programs," Ph.D. dissertation, FU Berlin, 2002.

[13] J. Aldrich and C. Chambers, "Ownership Domains: Separating Aliasing Policy from Mechanism," in *ECOOP*, 2004, pp. 1–25.

[14] R. Vanciu and M. Abi-Antoun, "Object Graphs with Ownership Domains: an Empirical Study," in *State-of-the-art Survey on Aliasing in Object-Oriented Programming*, ser. LNCS 7850, D. Clarke, J. Noble, and T. Wrigstad, Eds. Springer-Verlag, 2013, pp. 109–155.

[15] "MiniDraw," www.baerbak.com/.

[16] H. B. Christensen, *Flexible, Reliable Software Using Patterns and Agile Development*. Chapman and Hall/CRC, 2010.

[17] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Workshop on the Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2010.

[18] T. Reenskaug, *Working with objects: the OOram Software Engineering Method*. Manning/Prentice Hall, 1996.

[19] D. Riehle, "Framework Design: a Role Modeling Approach," Ph.D. dissertation, Federal Institute of Technology Zurich, 2000.

[20] B. Demsky and M. Rinard, "Role-based exploration of object-oriented programs," in *ICSE*, 2002, pp. 313–324.

[21] M. Marron, C. Sanchez, Z. Su, and M. Fähndrich, "Abstracting runtime heaps for program understanding," *TSE*, 2013.

[22] M. Taeumel, B. Steinert, and R. Hirschfeld, "The vivide programming environment: connecting run-time information with programmers' system knowledge," in *Onward!*, 2012.