2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation

Are Object Graphs Extracted Using Abstract Interpretation Significantly Different from the Code?

Marwan Abi-Antoun Sumukhi Chandrashekar Radu Vanciu Andrew Giang
Department of Computer Science, Wayne State University, Detroit, Michigan, USA
Email: {mabiantoun, sumukhic, radu, andrewgiang}@wayne.edu

***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.**

I. 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 [1], [2], by analyzing a finite number of executions, or static analysis by analyzing the code only [3]. 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 [4] 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 [5]. 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 III);
- 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 IV) then qualitatively (Section V);
- A qualitative analysis of the outliers to identify language features, designs or code patterns that contribute to larger differences (Section VI); 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 VII).

II. Background: Object Graph Semantics

This section reviews the semantics of the object graph. The representation handles inheritance, aliasing, and recursion [4].

978-1-4799-6148-1/14 $31.00 © 2014 IEEE
DOI 10.1109/SCAM.2014.42

$D \in$ ODomain ::=$\langle$**Id** $= D_{id}$, **Domain** $= C$::$d\rangle$
$O \in$ OObject ::=$\langle$**Type** $= C$, **OwningDomain** $= D1$, **OtherDomain** $= D2\rangle$
$E \in$ OEdge ::=$\langle$**From** $= O_{src}$, **Field** $= f$, **To** $= O_{dst}\rangle$

Fig. 1. Key data type declarations for the OGraph.

Due to space limits, the details are not repeated here. We discuss briefly inheritance.

Data types. The internal representation of an OOG is an OGraph (Fig. 1). 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 $\langle A, D1, D2\rangle$ to mean an abstract object of type A where $D1$ is the owning ODomain of O. Thus, in the object hierarchy, the abstract object $\langle C, D1, D2\rangle$ is shown as the child of the OwningDomain $D1$, which in turn is the child of some other OObject. The OGraph has a single root object. The OtherDomain, $D2$, identifies the OObject O further, and conceptually is a domain that the object has access to, i.e., it references objects from that domain. In this paper, we use a simplified definition of an abstract object, compared to the technical report [6].

In Ownership Domains, a class takes the parameters `owner` (the owning domain) and α (an additional parameter to pass through additional domains). Thus, a creation expression is `new` $C{<}p_1, p_2{>}()$. instead of `new` $C()$.

$$\texttt{class}\ A < \texttt{owner}, \alpha > \ \texttt{extends Object} < \texttt{owner} > \ \{\} \quad (1)$$

By having abstract objects of the form $\langle C, D1, D2\rangle$, the OOG can distinguish between different abstract objects of the same class C that are in different OwningDomains, even if created at the same object creation expression. Also, two abstract objects can have the same OwningDomain but different OtherDomains. This situation occurs, for example, when two collections have the same owning domain, but have different domains for their elements.

Since an ODomain represents an abstraction of a runtime domain, 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 one equivalence class, attaching an object label h to *each* object creation expression `new` $A()$ (line 12).

The OOG extraction analysis interprets an object creation expression (line 7) 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.

$[\ \texttt{this} \mapsto O_C\ ]$ // *1. Track receiver, analysis context* (2)
`class` $C <$ `owner`$, \alpha_C >$ `implements` $I <$ `owner` $>\ \{$ (3)
  `domain` d_{local}; (4)
  // *2. Create* ODomain $D_{local} = \langle o.d, C :: d_{local}\rangle$ (5)
  `void` $m()\ \{$ (6)
    `new`^{L} $A{<}p_{owner}, \alpha_C{>}()$; (7)
    // *3. Create* OObject $O_A = \langle A, D_1, D_2\rangle$ (8)
    such that $(O_C, p_{owner}) \mapsto D_1,\ (O_C, \alpha_C) \mapsto D_2$ (9)
    and $\exists O_B, \exists d$ s.t. $(O_B, d) \mapsto D1$ and $O_B = \langle B, \ldots\rangle$ (10)
    // *vs. Regular points-to analysis* (11)
    `new`^{h} $A()$ $\ldots\}\}$ (12)

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_1 (line 9), where D_1 is the child domain of some OObject O_B of type B and some actual domains (line 10).

In effect, the analysis labels all the objects that may be created at this object creation expression with one or more labels L where $L = O_A = \langle A, D_1, D_2\rangle$. Compared to the label h of a standard points-to analysis, there are two differences. First, distinct object creation expressions of type A can still have the same L label, if the analysis context maps the domain parameters p_{owner} and α_C to the same D_1 and D_2, but a basic points-to analysis will always create distinct h labels. Second, if the analysis context maps p_{owner} and α_C to n different $\langle A, D_1, D_2\rangle$ combinations, then one object creation expression of type A may create n different abstract objects, and thus n different L labels due to different combinations of domains.

Handling inheritance. The abstract interpretation handles inheritance. Domain parameters are bound between a class and its superclasses. When analyzing a class, the analysis recursively traverses all the superclasses, creating additional objects, domains or edges in the process.

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 Tf in some class in the code, where $T = C'{<}\texttt{owner}, \alpha{>}$.

To add edges between objects, the OOG extraction analyzes a field declaration Tf 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'.

III. Metrics

In the following, an abstract runtime element is either an abstract object or an abstract edge in the abstract runtime

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
```

Fig. 2. MiniDraw: hierarchy of classes vs. hierarchy of abstract objects.

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). The formal definitions are in a technical report [6]. 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.

A. RQ1

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.

Which-A (WA). This metric measures how many OObjects of type A exist in the OGraph.

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.

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. 2). *WAB* measures how frequently different OObjects of the same type A are in the same parent OObject B, but in different domains of B.

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. 2).

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.

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. 3).

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.

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. 3).

Inheritance Type Hierarchy

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

Hierarchy of objects

```
+- main:Brea
  +- MODEL
    +- bd:BD
      +- OWN
        +- e
    +- bf:BF
      +- OWN
        +- e
    +- p:Position
```

```
class AFigu
  FCEvent<OW
}
class FCEver
class BPFact
  Position<M
}
class MComma
  Position<M
  Position<M
}
class Breakt
  domain MOD
  BPFactory<
  MCommand<C
}
```

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

Fig. 3. 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.

For a points-to OEdge E, $traceToCode(E)$ is the set of field declarations that E can be traced to.

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.

B. RQ2

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

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. 3).

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. 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.

```
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>();
}
```

Fig. 4. The object of type `BFigure` is created in class `BDrawing` using the formal domain parameter `Df`. In the hierarchy of objects (Fig. 2), the object `BFigure` appears in the domain `MODEL` because `Df` is bound to `MODEL`.

For example (Fig. 3), 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.

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.

C. RQ3

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. 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. 2).

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.

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. 4).

The metric *PO_P* measures the percentage of pulled objects compared to all OObjects in the OGraph.

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.

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.

D. RQ4

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 $1 - \frac{3}{7} = 0.57$.

$$D_{dst} = mapFtoA(O_X, p) \text{ // map domain in the code to ODomain}$$

$$precisionRatio(C\text{<}\overline{p}\text{>}\ f) = \frac{|OOGPossibleSubTypes(C, D_{dst})|}{|AllPossibleSubClasses(C)|}$$

$$PTEP\_F(C\text{<}\overline{p}\text{>}\ f) = 1 - precisionRatio(C\text{<}\overline{p}\text{>}\ f)$$

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.

IV. Results

We compute the metrics on a corpus of object-oriented code totaling over 100 KLOC and consisting of eight subject systems (Table I) that already have annotations and extracted OGraphs [7]. 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 [7, Table 2].

TABLE I
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							

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 [6]. The generated sets are then used as input to the statistical analysis.

In Table II, 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 III. 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 $D \geq 0.474$ [8]. In Table II, 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 III, 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.

A. RQ1

Which-A (WA) and raw type version (WAR). The results imply that many objects of the same 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`.

Which-A-in-B (WAB) and raw type version (WABR). The results are not statistically significant, except for DL, and

TABLE II
STATISTICAL ANALYSIS FOR THE SET METRICS: P-VALUE (p), CLIFF'S DELTA (D) AND CLIFF'S DELTA SIZE (Δ). HIGHLIGHTED ARE VALUES OF $p <$ **0.05** AND **0.333** $\leq D$ (MEDIUM).

Metric	MD			CDB			AFS			DL			PX			JHD			HC			APD		
	p	D	Δ	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	**0.00**	0.04	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	**0.00**	0.07	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	0.50		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	0.50		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	0.50		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	0.50		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	**0.00**	0.01	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	**0.00**	0.07	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	**0.00**	0.12	1
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	**0.00**	0.29	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	**0.00**	0.14	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	**0.03**	*0.50*	2
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	0.19		1

TABLE III
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

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.

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

Same New Expression Different Objects (HMO). 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.

One Field Declaration Many Edges (1FnE). 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.

B. RQ2

Different New Expressions Same Object (HMN). 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.

Types Merged by Object (TMO). 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`.

Types in Object Substructure (TOS). While *TMO* measures differences due to inheritance, *TOS* focuses on composition. The results indicate a small or medium—but still statistically significant—difference for most systems.

C. RQ3

Same Package Different Domains (1PnD). The results are statistically significant for all systems except APD where all the classes are in one package.

Same Domain Different Packages (1DnP). The results are statistically significant for all systems except APD.

Pulled Objects Percentage (PO_P). 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.

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 `UndoActivity`, which traces to code to 5 different type declarations that implement the undo concept.

D. RQ4

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.

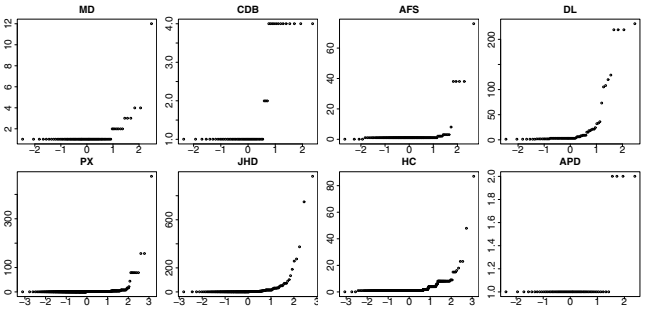

Fig. 5. QQ probability plots of *1FnE* for each system show exponential distributions with high values for the upper quintile.

V. Research Questions Revisited

A. RQ1

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. 5). 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.

B. RQ2

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.

C. RQ3

The metrics to answer RQ3 (*1PnD*, *1DnP*, *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.

Less than half of the objects are scattered. However, *SO_F_Median* and a high *SO_F_Max* indicate that some

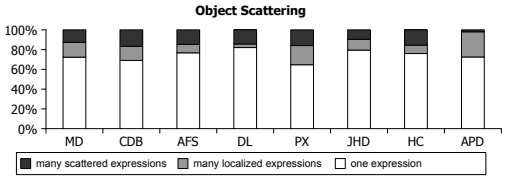

Fig. 6. Percentage of scattered objects (top categories). *SO_F* threshold 50%.

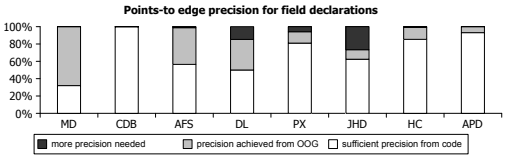

Fig. 7. Percentage of field declarations that require precision of runtime structure (top two categories). *PTEP_F* threshold 50%.

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.

D. RQ4

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 and observe that more precision is needed for 5% of the field declarations in PX, 15% in DL, and 25% in JHD (Fig. 7). 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.

E. 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 [9] or studies of runtime heaps using dynamic analysis [10]. 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.

VI. 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 [11]. 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.

VII. 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 [12] 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 [12, 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 [12, 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 reanalyzed 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:

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.

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.

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`.

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.

VIII. Related Work

Previous paper. This paper revises and extends an earlier workshop paper [13]. 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 [7] 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 [14] 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. [15] 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. [2] 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 [10]. 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 [16] specific to object-oriented code, such as the depth of the inheritance tree, coupling at the level of a class, method, or object [17]. Such metrics are used to predict the quality of the design [18] or the error-proneness of classes [19]. 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 [16]. 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 [20]. 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 [21] 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).

IX. 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. 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.

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

References

[1] N. Mitchell, "The Runtime Structure of Object Ownership," in *ECOOP*, 2006, pp. 57–64.
[2] E. T. Barr, C. Bird, and M. Marron, "Collecting a heap of shapes," in *ISSTA*, 2013.
[3] D. Jackson and A. Waingold, "Lightweight Extraction of Object Models from Bytecode," *TSE*, vol. 27, no. 2, pp. 156–169, 2001.
[4] M. Abi-Antoun and J. Aldrich, "Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations," in *OOPSLA*, 2009.
[5] J. Aldrich and C. Chambers, "Ownership Domains: Separating Aliasing Policy from Mechanism," in *ECOOP*, 2004, pp. 1–25.
[6] "Online appendix," www.cs.wayne.edu/~mabianto/arch_metrics/, 2014.
[7] 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. Springer-Verlag, 2013.
[8] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Routledge, 1988.
[9] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero, "Understanding the Shape of Java Software," in *OOPSLA*, 2006, pp. 311–324.
[10] A. Potanin, J. Noble, and R. Biddle, "Checking Ownership and Confinement," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 7, pp. 671–687, April 2004.
[11] K. Beck and E. Gamma, "JHotDraw – Patterns Applied (Tutorial)," in *OOPSLA*, 1997.
[12] N. Ammar and M. Abi-Antoun, "Empirical Evaluation of Diagrams of the Run-time Structure for Coding Tasks," in *WCRE*, 2012, pp. 367–376.
[13] M. Abi-Antoun, R. Vanciu, and N. Ammar, "Metrics to Identify Where Object-Oriented Program Comprehension Benefits from the Runtime Structure." in *Workshop on Emerging Trends in Software Metrics*, 2013.
[14] P. Lam and M. Rinard, "A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information," in *ECOOP*, 2003.
[15] M. Marron, C. Sanchez, Z. Su, and M. Fähndrich, "Abstracting runtime heaps for program understanding," *TSE*, vol. 39, no. 6, pp. 774–786, 2013.
[16] B. A. Kitchenham, "What's up with software metrics? – A preliminary mapping study," *J. Systems & Software*, vol. 83, no. 1, 2010.
[17] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic Coupling Measurement for Object-Oriented Software," *TSE*, vol. 30, no. 8, 2004.
[18] R. Marinescu, "Measurement and quality in object-oriented design," in *ICSM*, 2005.
[19] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *TSE*, vol. 22, no. 10, 1996.
[20] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshynanyk, and A. De Lucia, "An empirical study on the developers perception of software coupling," in *ICSE*, 2013.
[21] E. Tempero, H. Yang, and J. Noble, "What Programmers Do with Inheritance in Java," in *ECOOP*, 2013.