

**QUANTITATIVE AND QUALITATIVE EVALUATION OF METRICS  
ON OBJECT GRAPHS EXTRACTED BY ABSTRACT  
INTERPRETATION**

by

**SUMUKHI CHANDRASHEKAR**

**THESIS**

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

**MASTER OF SCIENCE**

2015

MAJOR: COMPUTER SCIENCE

Approved By:

---

Advisor

Date

## DEDICATION

*To my family for their endless support and encouragement.*

## ACKNOWLEDGEMENTS

This thesis is a summation of my work at the SoftwarE Visualization and Evolution REsearch (SEVERE) lab at Wayne State University, Detroit. I would like to take a moment to thank all of those who supported me during this expedition. First and foremost, I wish to express my gratitude to my advisor, Prof. Marwan Abi-Antoun for giving the opportunity to work on the thesis under his supervision, without whose guidance, it would had been never possible to complete this work. I would like to thank him for his patience and his criticism that helped me understand my flaws and mend my thinking to be more methodological. This lead me to complete the work with as much perfection as I can. His enthusiasm and motivation for research pushes me to keep going. Any remaining faults in this thesis are mine alone.

Deepest gratitude is given to Prof. Reynolds and Prof. Rajlich who were kind enough to accept my invitation to be part of the committee on a short notice despite their own busy academic schedules. I thank them for their valuable comments and discussions on this work.

I would also like to thank all my friends at Wayne State. I especially thank the SEVERE group members: Mohammad Anamul Haque, who read through my thesis with a new pair of eyes; Mohammad Ebrahim Khalaj, for being a supportive lab mate and for criticizing my paper drafts; Yibin Wang, for spending his weekends on reading my drafts and providing my constructive feedback; Laurentiu Radu Vanciu, for encouraging and motivating me at the start of my career in this lab. I would like to specially thank my friends in other labs Faria Mahnaz, Bhanukiran Vinzamuri and Diana Mabel Diaz for always being supportive.

The cooperation and support of the members of Department of Computer Science for providing the financial means and laboratory facilities for conducting the study is also very much appreciated. I would like to thank all the graduate committee members who took time off and answered all the questions patiently regarding various

process of submitting the thesis. I would also like to thank other faculty members who supported me. Prof. Goel has always been a constant source of encouragement and support during my graduate study.

Without the love, encouragement of my husband, inspiration of my mother and father, this accomplishment could have never been imagined. I also wish to express my gratitude to my brother and parents in law for their understanding through the duration of two years of my study.

## TABLE OF CONTENTS

Dedication . . . . .	ii
Acknowledgements . . . . .	iii
List of Tables . . . . .	viii
List of Figures . . . . .	ix
Chapter 1: Introduction . . . . .	1
1.1 Problem and Solution . . . . .	2
1.2 Thesis Statement . . . . .	3
1.3 Contributions . . . . .	3
1.4 Outline . . . . .	4
Chapter 2: Background . . . . .	5
2.1 Overview . . . . .	5
2.2 Object Graph Semantics . . . . .	6
Chapter 3: DiffMetrics . . . . .	11
3.1 DiffMetrics . . . . .	11
3.1.1 Category: One-To-Many . . . . .	11
Subcategory: DFMetrics . . . . .	11
3.1.2 Category: Many-To-One . . . . .	12
3.1.3 Category: MismatchedLocation . . . . .	13
3.1.4 Category: Precision . . . . .	13
3.2 Formalization of DiffMetrics . . . . .	14
3.2.1 DiffMetrics in One-To-Many Category . . . . .	14
3.2.2 DiffMetrics in Many-To-One Category . . . . .	23
3.2.3 DiffMetrics in MismatchedLocation Category . . . . .	23
3.2.4 DiffMetrics in Precision Category . . . . .	23
3.3 Quantitative Analysis of DiffMetrics . . . . .	24
Chapter 4: Approach Overview . . . . .	27

4.1	A Principled Approach . . . . .	27
4.2	Research Hypotheses . . . . .	28
4.2.1	Testing H1 . . . . .	28
4.2.2	Testing H2 . . . . .	29
Chapter 5:	Code Patterns as Predictors . . . . .	32
5.1	Visitors and Classifications of DiffMetrics . . . . .	33
5.1.1	Classifiers of Outliers . . . . .	33
	Classification of Outliers in One-To-Many Category . . . . .	34
	Classification of Outliers in Many-To-One Category . . . . .	37
	Classification of Outliers in Precision Category . . . . .	38
5.2	No-Annotation Visitors . . . . .	39
5.3	Using H1 as Predictor . . . . .	40
Chapter 6:	Simple Metrics as Predictors . . . . .	48
6.1	Subset of Code Metrics . . . . .	48
6.2	Correlation between DiffMetrics and Code metrics . . . . .	50
6.2.1	Correlation with DiffMetrics in One-To-Many Category . . . . .	50
6.2.2	Correlation with DiffMetrics in Many-To-One Category . . . . .	51
6.2.3	Correlation with DiffMetrics in Precision Category . . . . .	51
6.3	Using H2 as Predictor . . . . .	52
Chapter 7:	Evaluation Overview . . . . .	56
Chapter 8:	Evaluation . . . . .	59
8.1	Muspy . . . . .	59
8.1.1	Testing the Hypotheses . . . . .	59
8.1.2	Annotations . . . . .	61
8.1.3	Results from the DiffMetrics . . . . .	62
8.1.4	Results from the Wilcoxon test . . . . .	64
8.2	Ermete SMS . . . . .	65

8.2.1	Testing the Hypotheses . . . . .	65
8.2.2	Annotations . . . . .	67
8.2.3	Results from the DiffMetrics . . . . .	68
8.2.4	Results from the Wilcoxon test . . . . .	69
Chapter 9:	Related Work . . . . .	71
9.1	Metrics as Predictors . . . . .	71
9.1.1	Predictors of Maintainability and Defects . . . . .	71
9.1.2	Predictors of Program Comprehension . . . . .	72
9.1.3	Predictors of Runtime State or Properties . . . . .	73
9.2	Correlations and Code patterns as predictors . . . . .	75
Chapter 10:	Discussion and Conclusion . . . . .	77
10.1	Threats to Validity . . . . .	77
10.2	Limitations . . . . .	78
10.3	Global Discussion . . . . .	78
10.4	Future Work . . . . .	80
10.5	Conclusion . . . . .	80
References	. . . . .	86
Abstract	. . . . .	90
Autobiographical Statement	. . . . .	92

## LIST OF TABLES

Table. 4.1	Research Hypotheses. . . . .	28
Table. 4.2	Code Patterns identified from MD. . . . .	31
Table. 5.1	List of Training set Systems. . . . .	32
Table. 5.2	WABW Outlier. . . . .	34
Table. 5.3	1MInE_RecType Outlier. . . . .	37
Table. 5.4	DFEP Outlier. . . . .	39
Table. 5.5	Code Patterns and Classification. . . . .	40
Table. 6.1	Code Metrics for Training set Systems. . . . .	49
Table. 6.2	Average of Code Metrics. . . . .	49
Table. 6.3	Correlation of Code Metrics with DiffMetrics. . . . .	52
Table. 6.4	Average of Code Metrics. . . . .	53
Table. 7.1	Hypotheses tested on four Test Systems. . . . .	56
Table. 7.2	Classification count using No-Annotation Visitors. . . . .	57
Table. 7.3	Average of Code Metrics from Muspy and Ermete SMS. . . . .	57
Table. 7.4	Test Systems Selected for Evaluation. . . . .	57
Table. 7.5	Statistical analysis of the DiffMetrics. . . . .	58
Table. 8.1	Classification using No-Annotation Visitors. . . . .	60
Table. 8.2	WABW Outliers from Muspy. . . . .	63
Table. 8.3	DEEP Outliers from Muspy. . . . .	64
Table. 8.4	Count of outliers of DiffMetrics. . . . .	64
Table. 8.5	Classification using No-Annotation Visitors. . . . .	66
Table. 8.6	1MInE_ArgType Outlier. . . . .	69



Table. 8.7	Count of outliers of DiffMetrics. . . . .	69
Table. 9.1	Correspondence between DiffMetrics and Metrics from [9]. . . . .	75
Table. 1	Number of Outliers in each Classification. . . . .	85

## LIST OF FIGURES

Figure. 2.1	Hierarchy of classes vs. Hierarchy of objects. . . . .	6
Figure. 2.2	Portions of the Ownership Domains abstract syntax [3]. . . . .	7
Figure. 2.3	Key data type declarations for the <b>OGraph</b> . . . . .	8
Figure. 2.4	Example illustrating dataflow import and export edges . . . . .	9
Figure. 3.1	Formal definition of WAWB. . . . .	14
Figure. 3.2	1 method invocation may have N edges in the OGraph. . . . .	16
Figure. 3.3	Illustrating DFMetrics associated with Method Invocation. . . . .	17
Figure. 3.4	Formal definition of 1MInE_RecType. . . . .	18
Figure. 3.5	Formal definition of 1MInE_ArgType. . . . .	18
Figure. 3.6	Formal definition of 1MInE_RetType. . . . .	18
Figure. 3.7	1 field read may have N edges in the OGraph. . . . .	19
Figure. 3.8	Formal definition of 1FRnE. . . . .	20
Figure. 3.9	1 field write may have N edges in the OGraph. . . . .	21
Figure. 3.10	Formal definition of 1FwnE. . . . .	21
Figure. 3.11	Illustrating DFMetrics associated with field read and write. . . . .	22
Figure. 3.12	Formal definition of TMO. . . . .	23
Figure. 3.13	Formal definition of PO. . . . .	23
Figure. 3.14	Formal definition of PTEP. . . . .	24
Figure. 3.15	Formal definition of DFEP. . . . .	24
Figure. 3.16	Examples illustrating PTEP, DFEP. . . . .	25
Figure. 3.17	Quantitative Analysis. . . . .	26
Figure. 4.1	No-Annotation Visitors. . . . .	29

Figure. 4.2	Test H1. . . . .	29
Figure. 4.3	Small portion of metrics_map from MD. . . . .	30
Figure. 4.4	Compute Code Metrics. . . . .	30
Figure. 4.5	Test H2. . . . .	30
Figure. 5.1	Visit Outliers of DiffMetrics. . . . .	33
Figure. 5.2	Pseudo-code of the Visitors. . . . .	42
Figure. 5.3	Classification of OObjects. . . . .	43
Figure. 5.4	Hierarchy of OObjects. . . . .	43
Figure. 5.5	Classification of DiffMetrics under DFMetrics. . . . .	44
Figure. 5.6	Classification of fields and variables of DFEP. . . . .	45
Figure. 5.7	Classification of Fields and Variables from No-Annotation Visitors. . . . .	45
Figure. 5.8	Pseudo-code of the No-Annotation Visitor. . . . .	46
Figure. 5.9	Illustrating Code Patterns and Classification. . . . .	47
Figure. 6.1	CBO computed using STAN. . . . .	48
Figure. 6.2	Sample src.code from MD. . . . .	54
Figure. 6.3	TMO and DIT. . . . .	55
Figure. 6.4	RMA and PTEP, DFEP. . . . .	55
Figure. 8.1	Annotation of Muspy. . . . .	62
Figure. 8.2	Inheritance hierarchy of AAct. . . . .	63
Figure. 8.3	Annotation of Ermete SMS. . . . .	68

# Chapter 1: Introduction

Evaluating programming language techniques is done using case studies, controlled experiments, corpus analysis, or surveys [14]. Corpus analysis has been extensively used to evaluate techniques that promote structural typing [15], express protocols [6] and prevent security vulnerabilities [18]. For evaluation approaches that involve case studies or controlled experiments with participants, it may not be possible to convincingly demonstrate the usefulness of the technique as there are confounds in using participants such as differing skills or experience levels, and how amenable they are to learning new tools, notations. Any evaluation approach that may be used, may indicate that the technique is beneficial for some systems but not for other systems or tasks [12, 19].

Often times, to evaluate a heavyweight technique that requires specifications or annotations be added to the code, a lightweight proxy is used to select systems on which using the technique is rewarding and evaluation can be conducted. The proxy may consists of a visitor that identify methods with specific substrings in their names, or a specific signature. However, such a proxy is often determined *a priori* and may be unreliable. The proxy may predict a larger benefit than can be achieved in practice. Also, finding a system for which the proposed heavyweight technique shines is often a process of trial and error. Even if such a system is found, there is little understanding of what types of similar systems these findings can be generalized to.

This work proposes a principled approach to derive a proxy that predicts if using a heavyweight technique may be beneficial on a system. We use the approach to derive a proxy for a heavyweight technique that our research group has been working on. The technique extracts the abstract runtime structure of a system based on the annotations that are added manually to the code, thus making it *heavyweight*. As a first step, we compute the DiffMetrics that measure the differences between the

system representation extracted by the heavyweight technique (the abstract runtime structure) and a baseline representation (the code structure) to investigate how different the two representations are for subject systems. We then identify data points of the DiffMetrics that are higher than a predefined threshold, the outliers. We identify code patterns that are associated with the outliers. We then implement visitors that visit the code of a system to identify code patterns and to classify the outliers into several classifications based on the identified code patterns. We generalize the visitors to identify the same code patterns on systems that do not have any annotations added to the code. Independently, we determine other code metrics that correlate with the DiffMetrics. The proxy runs the visitors to identify the code patterns and computes the code metrics on new systems that have not been analyzed using our heavyweight technique. Based on the code patterns and the values of the code metrics computed on the systems, the proxy predicts if its abstract runtime structure may be significantly different from the code structure as the outliers of the DiffMetrics indicate.

## 1.1 Problem and Solution

**The Problem.** To evaluate a heavyweight technique, a lightweight proxy is used to select systems on which the technique may be useful. Often, the proxy is defined arbitrarily and not specifically for a heavyweight technique. Such a proxy may be unreliable. Thus, we need a more principled approach to define a proxy that predicts if a heavyweight technique may work for a system without applying the technique on the system.

**Solution.** We propose a principled approach to derive a proxy to predict if the heavyweight technique that extracts the abstract runtime structure may prove to be useful on a system. The proxy uses the DiffMetrics that are computed on the training

set to build a model that predicts if the abstract runtime structure of a new system is significantly different from its code structure. Based on the prediction, we decide if using the heavyweight technique on this system may be beneficial or not. As the proxy uses the DiffMetrics that are computed on the training set to build a model, it is data-driven. The proxy is defined to work specifically for a heavyweight technique and thus is more reliable.

## 1.2 Thesis Statement

Two tests predict if a system’s abstract runtime structure is different from the corresponding code structure, as indicated by DiffMetrics that measure the differences between the two representations. The first test consists of running visitors that look for code patterns identified from the outliers. The second test consists of computing code metrics that are strongly correlated with the DiffMetrics.

## 1.3 Contributions

The contributions of this work are as follow:

- I. Formal definition of some DiffMetrics that measure the differences between the abstract runtime structure and the code structure of systems based on dataflow edges;
- II. A principled approach to derive a lightweight proxy for a heavyweight technique;
- III. A proxy derived using the approach for a heavyweight technique that extracts the abstract runtime structure for a system;
- IV. An evaluation of the proxy’s predictions on systems that were not part of the evaluation corpus of our heavyweight technique;

## 1.4 Outline

The rest of this thesis is organized as follows: Chapter 2 discusses the background of the heavyweight technique for which we propose a lightweight proxy. Next, we discuss the DiffMetrics and the formalization of the DiffMetrics in Chapter 3. We briefly discuss the proposed data-driven approach to derive a lightweight proxy and discuss the research hypotheses in Chapter 4. In Chapter 5 and Chapter 6 we discuss in detail the proxy, which consists of two tests. We evaluate the proposed proxy on four test systems. A brief outline of the evaluation is discussed in Chapter 7. The proxy predicts that using the heavyweight technique may prove to be useful for developers during program comprehension, code modification tasks, for only two out of the four test systems. We discuss the results of the proxy and the DiffMetrics after applying the heavyweight technique on these test systems. Chapter 9 discusses research work in same line as the work in this thesis such as the use code metrics and code patterns to predict maintainability, defects and testability. Chapter 10 concludes the thesis work and discusses some potential future work.

## Chapter 2: Background

We give some background for the heavyweight technique for which we define the proxy.

### 2.1 Overview

We have been investigating a global hierarchy of abstract objects as a first-class view of an object-oriented system at design time, one that conveys architectural abstraction by showing *abstract objects*, rather than specific instances. This view promotes a new design-time thinking in terms of an abstract runtime structure consisting of abstract objects and abstract edges between them.

**Why Abstraction.** Much of the functionality is determined by what instances point to what other instances in object-oriented systems. Tasks that require knowing the number of instances of a type that are created, or how one instance is related to how many other instances, a debugger may be useful. But for program comprehension tasks, understanding the system just in terms of specific instances or concrete objects may not be effective. Instead, it may be useful to abstract objects and merge conceptually similar objects or those that play the same role into one abstract object. Our heavyweight technique that extracts a hierarchy of abstract objects in the form of a global, hierarchical object graph, the Ownership Object Graph (OOG) [1], presents to the developers a manageable number of abstract objects.

With the help of the OOG, the developers may be able to distinguish the role that an object plays not just by *type* but also by named *group* referred as the domain and by position in an object *hierarchy* that dictates parent-child relationships between objects. Using *type+ierarchy+group*, developers can construct an abstract runtime structure in the form of an OOG that soundly approximates all possible objects and relations.



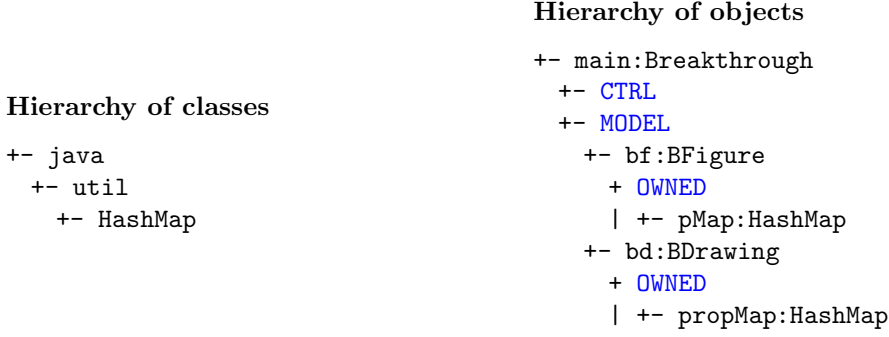


FIGURE 2.1: A hierarchy of classes has one type `HashMap`. A hierarchy of objects has multiple abstract objects of type `HashMap` distinguished based on their parent domain and hierarchy.

**Hierarchy of Classes vs. Hierarchy of Objects.** Current tools present the code structure of a system in terms of *hierarchy of classes* where the classes are organized by packages. In contrast, the analysis that extracts the abstract runtime structure presents a *hierarchy of objects* to the developers. We illustrate an example with a subject system to distinguish the hierarchy of classes from the hierarchy of objects that may have multiple abstract objects of the same type (Figure 2.1).

## 2.2 Object Graph Semantics

A static analysis extracts the OOG using an abstract interpretation of code to which manual annotations have been added. The annotations are checked to be consistent with the code and implement a type system, Ownership Domains [3]. We briefly review the Ownership Domains type system and the OOG semantics in order to formally define our DiffMetrics.

**Abstract syntax.** A portion of the abstract syntax for Ownership Domains is presented (Figure 2.2), focusing on class declarations, field declarations, expressions e.g., method invocations, field reads and field writes. The meta-variable  $C$  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 domain `SHARED`. An overbar over the meta-variable represents a sequence.

$cdef$	$::=$	$\text{class } C\langle\bar{\alpha}, \bar{\beta}\rangle \text{ extends } C'\langle\bar{\alpha}\rangle$ $\{ \bar{dom}; \bar{T} \bar{f}; \bar{md} \}$
$dom$	$::=$	$[\text{public}] \text{ domain } d;$
$md$	$::=$	$T_R m(\bar{T} \bar{x}) T_{this} \{ret = e_R; \text{return } e_R; \}$
$e$	$::=$	$x = \text{new } C\langle\bar{p}\rangle(\bar{e}) \mid x = r.m(\bar{e})$ $\mid x = y.f \mid x.f = y \dots$
$n$	$::=$	$x \mid v \mid r \mid y$
$p$	$::=$	$\alpha \mid n.d \mid \text{SHARED}$
$T$	$::=$	$C\langle\bar{p}\rangle$
$v, \ell \in \text{locations}$		

FIGURE 2.2: Portions of the Ownership Domains abstract syntax [3].

A class is parameterized by a list of domain parameters, and extends another class that has a subsequence of its domain parameters. A type  $T$  is a class name and a set of actual domain parameters  $C\langle\bar{p}\rangle$ . The first domain parameter of a class is its owning domain followed by other domain parameters.

**Data types.** The internal representation of an OOG is an **OGraph**. The **OGraph** has two types of nodes: the **OObjects** referred to by the meta-variable  $O$  and the **ODomains** referred to by the meta-variable  $D$ . Two **OObjects** may be connected by **OEdges** referred to by the meta-variable  $E$  and can represent *points-to*, or *dataflow* relations.

An **OObject** is represented using the tuple  $\langle A, D1, D2 \rangle$ . The tuple represents an abstract object of type  $A$  whose owning domain is  $D1$ ;  $D2$  is a domain that the object has access to, i.e., it references objects from that domain. By having abstract objects of the form  $\langle C, D1, D2 \rangle$ , the OOG distinguishes different abstract objects of the same type  $C$  that are in different owning domains or that have the same owning domain but different other domains that the object has access to.

The **ODomain** represents an abstraction of a runtime domain, one domain declaration  $D$  in a type  $C$  can correspond to multiple **ODomains**  $D_i$  in the **OGraph**. The static analysis computes an abstract object of type  $C$  in some domain  $D$  based on mapping the formal domain parameters in the code to domains that may be declared by other types  $C_i$ .

$D \in \text{ODomain}$	$::=\langle \mathbf{Id} = D_{id}, \mathbf{Domain} = C::d \rangle$
$O \in \text{OObject}$	$::=\langle \mathbf{Type} = C, \mathbf{OwningDomain} = D1, \mathbf{OtherDomain} = D2 \rangle$
$E \in \text{OEdge}$	$::=\langle \mathbf{From} = O_{src}, \mathbf{Field} = f, \mathbf{To} = O_{dst} \rangle$
$E \in \text{ODFEdge}$	$::=\langle \mathbf{From} = O_{src}, \mathbf{To} = O_{dst}, \mathbf{Lable} = O_{label}, \mathbf{Flag} = Imp Exp \rangle$

FIGURE 2.3: Key data type declarations for the OGraph.

**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 a field declaration  $Tf$  in a class in the code, where  $T = C' < \mathbf{owner}, \alpha >$ .

While points-to OEdges are useful, the relationships between abstract objects due to field usages and dataflow relations are also crucial. Data-flow OEdges represent such relations in the OGraph. They express references propagating between the OObjects and are referred to as the flow OObjects. We identify two different types of dataflow scenarios, Import and Export. The OGraph shows import and export dataflow edges for the two scenarios respectively.

- I. Export scenario: In a scenario where an object  $a$  of type **A** owns reference of an object  $c$  of type **C** and passes that to an object  $b$  of type **B**, the OGraph has an export edge from source object  $a: A$  to destination object  $b: B$  with the flow object  $c: C$  propagating between them. Such an edge is extracted due to a method invocation or a field write in the OGraph.
- II. Import scenario: In a scenario where  $a$  of type **A** owns reference to  $b$  of type **B** from which it receives a reference to another object  $c$  of type **C**, the OGraph has an import edge from source object  $b: B$  to destination object  $a: A$  with the flow object  $c: C$  propagating between them. Such an edge is extracted due to a field read or a method return statement.

A type has method declarations; each method declaration has method body; the method body includes method invocations.

```

class A<OWNER> {
  domain D1, D2
  B<D1,D2> oB;
  D<D1,D2> oD;
  void ma(){
    // Method Invocation
    oB.mb(oD);
    // Field Read
    C<D1,D2> c = oB.oC;
  }
}

class B<OWNER,BDOM> {
  domain DOM
  // Field
  C<DOM,BDOM> oC;
  // Method declaration
  void mb(D<DOM,BDOM> oD) {
    ...
  }
}

```

FIGURE 2.4: Example illustrating dataflow import and export edges

Figure 2.4 illustrates dataflow OEdges in the OGraph. The type A has a method declaration `void ma()` that has a method invocation, `oB.mb(D oD)` that is declared in type B. So the analysis adds an export edge from the abstract object  $a: A$  to the abstract object  $b: B$  propagating the reference of the argument of the abstract object  $d: D$  in the OGraph. The type A reads a field of type C declared in B. The OGraph has an import edge from  $b: B$  to  $a: A$  propagating the reference of the abstract object that is type of the field associated in the read ( $c: C$ ).

To add an OEdge between OObjects, the OOG extraction analyzes a field declaration  $Tf$ , method invocation `recv.method(List<Arguments>)`, field write `field1 = recv.field`, field read `recv.field` in a type  $C$ , in a given analysis context  $O_{this}$ . 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 and as its destination each OObject  $O_t$  in  $D$ .

Extracting an OOG for a system is a two-step process. Developers add annotations that express their design intent. Then they use a static analysis that extracts the abstract runtime structure represented by its OOG from the annotated code. The details of the steps are discussed below.

**Add annotations.** To extract an OOG that is hierarchical and that conveys design intent, we assign each object to a *domain*. The developers understand the structure

of the system and decide the top-level architectural tiers that are also the top-level domains in the OOG. Each object is assigned to a single domain that does not change at runtime. The developer assigns a domain to an object by annotating each of its references. The annotations define two kinds of object hierarchies. Public domain provides *logical containment* by making an object *part of* another object. Private domain provides *strict encapsulation* by making an object *owned by* its parent object. The developers iterate the process until all types in the system are annotated.

**Run a static analysis.** A static analysis extracts the abstract runtime structure from the annotated code. The developers choose a root class as the starting point. The analysis maps the formal domain parameters to the actual domains to which they are bound. The OOG is both abstract and sound. It is abstract in the sense that a canonical abstract object may correspond to many objects at runtime and sound in the sense that the abstract runtime structure considers all possible runtime objects and edges. The OOG is hierarchical where an abstract object can have one or more nested domains that contain other abstract objects. The abstract objects that are architecturally significant are at the top of the hierarchy and the ones that represent data structures or other implementation details are hidden in the lower levels of the hierarchy.

To decide if it is worthwhile extracting the OOG for a system and be able to understand if an OOG may be useful for program comprehension or code modification tasks, it is instructive to measure how much the OOG, derived from the usage of objects, differs from the code structure derived from syntactic declarations in the Abstract Syntax Tree (AST) of the program, using the DiffMetrics that we discuss in the next chapter.

## Chapter 3: DiffMetrics

We illustrate the DiffMetrics with examples. We define the DiffMetrics more formally and analyze the DiffMetrics quantitatively.

### 3.1 DiffMetrics

The DiffMetrics relate one or more code elements in the code structure to one or more abstract runtime elements (OObjects and OEdges) in the OGraph. They are grouped into several categories. The DiffMetrics that relate the points-to edges and OObjects to elements in the AST are discussed in a previous work [2]. However, the DiffMetrics that relate the dataflow edges to elements in the AST are discussed here.

#### 3.1.1 Category: One-To-Many

This category measures how often *one* code element maps to *many* abstract runtime elements. A type in the system may have many instances at runtime.

- I. *Which-A-in-Which-B (WAWB)* measures how frequently different OObjects of the same type are assigned to different parent OObjects of different types in the OGraph based on the roles the objects play.

We define other DiffMetrics that measure how often one expression e.g., method invocation, field read, or field write in the code may map to many abstract runtime dataflow edges in the OGraph. Such DiffMetrics are grouped into a subcategory of One-To-Many Category.

#### Subcategory: DFMetrics

- I. *One Method Invocation Many Edges – Receiver Type (1MinE\_RecType)* measures how many OEdges in the OGraph are due to the same method invocation

in the code such that the types of the destination `OObjects` of the `OEdges` are subtype compatible with the receiver type of the invocation.

- II. *One Method Invocation Many Edges – Argument Type (1MInE\_ArgType)* measures how many edges in the `OGraph` are due to the same method invocation in the code such that the types of the flow `OObjects` between the source and destination `OObjects` are subtype compatible with the actual argument types of the invocation.
- III. *One Method Invocation Many Edges – Return Type (1MInE\_RetType)* measures how many edges in the `OGraph` are due to the same method invocation in the code such that the types of the flow `OObjects` between the source and destination `OObjects` are subtype compatible with the return type of the corresponding method declaration.
- IV. *One Field Read Many Edges (1FRnE)* measures how many edges in the `OGraph` are due to the same field read expression in the code such that the types of the source `OObjects` are subtype compatible with the receiver type of the expression.
- V. *One Field Write Many Edges (1FWnE)* measures how many edges in the `OGraph` are due to the same field write such that the types of the destination `OObjects` of the edges are subtype compatible with the receiver type of the expression.

### 3.1.2 Category: Many-To-One

This category measures how *many* code elements map to *one* abstract runtime element. When the code and the abstract runtime structure are very closely aligned, distinct code elements correspond to distinct runtime elements. However, when the two structures are different, different types in the program may correspond to the same runtime elements.

- I. *Types Merged by Object (TMO)* measures the number of distinct types, excluding interfaces, that are merged by an `OObject` in the `OGraph`.

### 3.1.3 Category: MismatchedLocation

Each `OObject` assigned to a domain appears where that domain is declared in the `OGraph`. For example, an object of type `A` that is assigned to a domain `D` that may be declared in the object of type `B`, will appear in the hierarchy of the parent object (`B`), inside the domain `D` in the `OGraph`. However, in the code, the type `B` may not directly create an object of type `A`. The `OObject` is mapped to a domain of a parent `OObject` after the analysis maps formal domain parameters to the actual domains to which they are bound to. This category of `DiffMetrics` measures how often the location of an `OObject` appears in the hierarchy of another object that does not directly create this object in the code.

- I. *Pulled Objects (PO)* measures the percentage of `OObjects` assigned to domains of parent `OObjects` that may not directly create the object compared to all `OObjects` in the `OGraph`.

### 3.1.4 Category: Precision

This category measures the precision of the abstract runtime structure compared to the code structure. As the `OGraph` is extracted by an analysis that uses abstract interpretation, it resolves some information compared to a visitor that traverses the AST of a system. If the abstract runtime structure contains significantly more precision than the code structure, then the `OGraph` may help in program comprehension.

- I. *Points-To Edge Precision (PTEP)* measures how precisely the `OGraph` resolves the possible types of all the objects that a field may reference, compared to all the possible subtypes of the field's type.



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

FIGURE 3.1: Formal definition of WAWB.

We derive another DiffMetrics that generalizes the precision based on abstract interpretation to other variables and fields that may be the receivers of method invocations, field reads, field writes, method parameters and method returns.

- II. *Data-Flow Edge Precision (DFEP)* measures how precisely the **OGraph** resolves the possible types of all the objects that such a field or variable may reference, compared to all possible subtypes of the field's or variable's type.

## 3.2 Formalization of DiffMetrics

The DiffMetrics under each category are discussed more formally here. The formalization uses the Featherweight Domain Java (FDJ) syntax. The details of the syntax are discussed in [3]. We illustrate the DiffMetrics using a subject system MiniDraw (MD).

### 3.2.1 DiffMetrics in One-To-Many Category

**WAWB**<sup>1</sup> collects the unordered pairs of **OObjects** that satisfy the condition in Figure 3.1.

To discuss DiffMetrics that relate a method invocation to many abstract dataflow edges, we first establish that one method invocation (MI) may be associated with N different dataflow import or export edges in the **OGraph**.

---

<sup>1</sup>The definition of WAWB is included here for completeness. It includes contributions by co-authors [2].

**One Method Invocation and N edges.** The **OGraph** may have many export dataflow edges from the **OObject** of the enclosing type of the expression to the **OObjects** of the receiver or subtypes of the receiver type that are in reachable domains after the formal domain parameters are bound to the actual domains propagating **OObjects** of the argument types or subtypes of argument types between the abstract objects. Similarly, the **OGraph** may have many import edges when the reference **OObjects** propagating between the abstract objects may be subtypes of the return type of the corresponding method declaration that are in reachable domains after the formal domain parameters are bound to the actual domains.

Formally, the **OGraph** has  $N$  import or export dataflow edges (Figure 3.2). The edges trace to the same method invocation (1MI) in the code. The method is invoked in class  $C$  through the receiver of type  $C_{rec}$ . This indicates that the method may be declared in  $C_{rec}$  or in a type that may be subtype compatible with  $C_{rec}$ . The invocation has sequence of arguments and return an **OObject** of type  $C_{ret}$  or a subtype of  $C_{ret}$ .

Once we have established that one method invocation in the code may relate to many abstract dataflow edges in the **OGraph**, we discuss that **DiffMetrics**, **1MInE\_RecType**, **1MInE\_ArgType** and **1MInE\_RetType**, that count number of edges due to the same method invocation in the code satisfying different conditions.

**1MInE\_RecType** collects the sets of **OEdges** such that the type of the destination **OObjects** are subtypes of the receiver of a method invocation (Figure 3.4).

For example, the method invocation `figure.addFigChangeListener(this)` (Figure 3.3) is associated with two distinct dataflow export edges  $E1$  and  $E2$  such that the destination **OObjects** of types **BDrawing** and **BFigure** are subtype compatible with the receiver type **Figure** in the **OGraph**. The **OObjects** are in domain **MODEL** and the owning domain **M** of **Figure** maps to **MODEL**.

**1MInE\_ArgType** collects the sets of **OEdges** such that the type of the flow **OObjects**

$E_i = \langle Osrc_i, Odst_i, Oflow_i, EXP|IMP \rangle // 1. \text{ Edge } 1$   
 $E_j = \langle Osrc_j, Odst_j, Oflow_j, EXP|IMP \rangle // 2. \text{ Edge } 2$   
*s.t*  
 $E_i \neq E_j$   
 Edges due to the same method invocation  
 $x = r.m(\overline{y}) \in traceToCode(E_i) \cap traceToCode(E_j)$   
 Receiver of the method invocation  
 $r : T_{rec}, T_{rec} = C_{rec} \langle \overline{p_{rec}} \rangle$   
 Variable or a field  
 $x : T_x, T_x = C_x \langle \overline{p_x} \rangle$   
 Actual arguments of the method invocation  
 $\overline{y} : \overline{T_{arg}}, T_{arg} = C_{arg} \langle \overline{p_{arg}} \rangle$   
 The method m is declared in  $C_{declr}$   
 $mbody(m, C_{declr} \langle \overline{p_{declr}} \rangle) = (\overline{arg} : \overline{T}, ret)$   
 Return type the method  
 $ret : T_{ret}, T_{ret} = C_{ret} \langle \overline{p_{ret}} \rangle$   
 Receiver may be subtype of type that declares m  
 $C_{rec} <: C_{declr}$   
 x may be subtype of return type  
 $C_x <: C_{ret}$

FIGURE 3.2: 1 method invocation may have N edges in the OGraph.

are subtypes of the argument types of a method invocation (Figure 3.5).

For example, the OGraph has two distinct dataflow edges E3 and E4 propagating the flow OObjects of types BDrawing and BFigure due to the method invocation `fFigure.remove(figure)` (Figure 3.3). The owning domain M of the actual argument of the invocation maps to the actual domain MODEL and both the flow OObjects are in the domain MODEL.

**1MInE\_RetType** collects the sets of OEdges such that the type of the flow OObjects are subtypes of the return type of the corresponding method declaration of a method invocation (Figure 3.6).

For example, the method invocation `figure = fFigure.get(index)` (Figure 3.3)

```

class BDrawing<OWNER, M, C> implements Figure<OWNER, M, C>{
  Figure<M, M, C> figure;
  List<OWNED, <Figure><M, M, C>> fFigure;
  buildPropMap() {
    // Example: 1MInE_RecType
    figure.addFigChangeListener(this)
    // Edges due to the above method invocation
    E1 = <bdrawing, bdrawing, bdrawing, EXP>
    E2 = <bdrawing, bfigure, bdrawing, EXP>
  }
  pieceMoveEvent() {
    // Example: 1MInE_RetType
    int index;
    figure = fFigure.get(index);
    // Edges due to the above method invocation
    E5 = <fFigure, bdrawing, bdrawing, IMP>
    E6 = <fFigure, bdrawing, bfigure, IMP>
    // Example: 1MInE_ArgType
    fFigure.remove(figure);
    // Edges due to the above method invocation
    E3 = <bdrawing, fFigure, bdrawing, EXP>
    E4 = <bdrawing, fFigure, bfigure, EXP>
  }
}
class BFigure<OWNER, M, C> implements Figure<OWNER, M, C> { ... }
class Breakthrough {
  domain MODEL, CTRL;
  // Subtypes of Figure in reachable domains
  // M maps to MODEL
  // C maps to CTRL
  BDrawing<MODEL, MODEL, CTRL> bdrawing = ...;
  BFigure<MODEL, MODEL, CTRL> bfigure = ...;
}

1MInE_RecType: figure.addFigChangeListener(this) = 2
1MInE_ArgType: fFigure.remove(Figure) = 2
1MInE_RetType: figure = fFigure.get(index) = 2

```

FIGURE 3.3: Examples illustrating 1MInE\_RecType, 1MInE\_ArgType, 1MInE\_RetType.

is associated with two distinct import dataflow edges E5 and E6 in the **OGraph** propagating the flow **OObjects** of types **BDrawing** and **BFigure** that are subtype compatible with return type.

To discuss the **DiffMetrics** that relate a field read or field write expression to many abstract dataflow edges, we first formally indicate that one field read (FR) and one field write (FW) may be associated with  $N$  dataflow import or export edges respectively in the **OGraph**.

$\{E_i\}$  where  $(r.m(\overline{y})) \in traceToCode(E_i)$

*s.t*

$$r : T_{rec}, T_{rec} = C_{rec} \langle \overline{p_{rec}} \rangle$$

$$O_{dst_i} = \langle A_{rec_i} \langle \overline{D_{rec_i}} \rangle \rangle$$

Destination OObject types and receiver types are subtype compatible

$$A_{rec_i} <: C_{rec} \mid C_{rec} <: A_{rec_i}$$

and  $A_{rec_i} \neq C_{rec}$  and  $flag = EXP$

FIGURE 3.4: Formal definition of 1MinE\_RecType.

$\{E_i\}$  where  $(r.m(\overline{y})) \in traceToCode(E_i)$

*s.t*

$$\overline{y} : \overline{T_{arg}}, T_{arg} = C_{arg} \langle \overline{p_{arg}} \rangle$$

$$O_{flow_i} = \langle A_{flow_i} \langle \overline{D_{flow_i}} \rangle \rangle$$

Flow OObject types and argument types are subtype compatible

$$\forall C_{arg} \langle \overline{p_{arg}} \rangle$$

$$A_{flow_i} <: C_{arg_k} \mid C_{arg_k} <: A_{flow_i}$$

and  $A_{flow_i} \neq C_{arg_k}$  and  $flag = EXP$

FIGURE 3.5: Formal definition of 1MinE\_ArgType.

$\{E_i\}$  where  $(x = r(\overline{y})) \in traceToCode(E_i)$

*s.t*

$$x : T_{ret}, T_{ret} = C_{ret} \langle \overline{p_{ret}} \rangle$$

$$O_{flow_i} = \langle A_{flow_i} \langle \overline{D_{flow_i}} \rangle \rangle$$

Flow OObject types and return types are subtype compatible

$$A_{flow_i} <: C_{ret} \mid C_{ret} <: A_{flow_i}$$

and  $A_{flow_i} \neq C_{ret}$  and  $flag = IMP$

FIGURE 3.6: Formal definition of 1MinE\_RetType.

**One Field Read and N edges.** One field read  $x = r.f$  in class  $C$  may be associated with N dataflow import edges in the OGraph from the OObject of receiver or OObjects

$E_i = \langle Osrc_i, Odst_i, Oflow_i, IMP \rangle$   
 $E_j = \langle Osrc_j, Odst_j, Oflow_j, IMP \rangle$   
*s.t*  
 $E_i \neq E_j$   
 Edges due to the same field read expression  
 $x = r.f \in traceToCode(E_i) \cap traceToCode(E_j)$   
 Receiver of the field read  
 $r : T_{rec}, T_{rec} = C_{rec} \langle \overline{p_{rec}} \rangle$   
 Variable or field  
 $x : T_x, T_x = C_x \langle \overline{p_x} \rangle$   
 Field type  
 $f : T_f, T_f = C_f \langle \overline{p_f} \rangle$   
 The field f is declared in  $C_{declr}$   
 $(T_f f) \in fields(T_{declr}), T_{declr} = C_{declr} \langle \overline{p_{declr}} \rangle$   
 Receiver may be subtypes with type that declares the field  
 $C_{rec} <: C_{fdeclr}$   
 x may be subtype of field type  
 $C_x <: C_f$

FIGURE 3.7: 1 field read may have N edges in the OGraph.

of subtypes of the receiver that are in reachable domains after the formal domain parameters are bound to the actual domains to the **OObject** of the enclosing type of the expression.

Formally **OGraph** has N import dataflow edges that trace to a single field read expression (1FR) in the code (Figure 3.7). The field is read through the receiver of type  $C_{rec}$ . The field of the type  $C_f$  is declared in  $C_{rec}$  or in subtypes of receiver type.

Once we have established that one field read in the code may relate to many abstract dataflow edges in the **OGraph**, we discuss that DiffMetrics, 1FRnE, that count number of edges due to the same field read in the code satisfying a specific conditions.

**1FRnE** collects the sets of **OEdges** such that the type of the source **OObjects** are

$$\begin{aligned}
& \{E_i\} \text{ where } (x = r.f) \in \text{traceToCode}(E_i) \\
& s.t \\
& r : T_{rec}, T_{rec} = C_{rec} \langle \overline{p_{rec}} \rangle \\
& O_{src_i} = \langle A_{rec_i} \langle \overline{D_{rec_i}} \rangle \rangle \\
& \text{Source OObject types and receiver types are subtype compatible} \\
& A_{rec_i} <: C_{rec} \mid C_{rec} <: A_{rec_i} \\
& \text{and } A_{rec_i} \neq C_{rec} \text{ and } flag = IMP
\end{aligned}$$

FIGURE 3.8: Formal definition of 1FRnE.

subtypes of the receiver type of a field read expression (Figure 3.8).

For example, the field read `x = md.editor` in the enclosing type `ATool` is associated with four distinct import dataflow edges in the `OGraph` E1, E2, E3 and E4 (Figure 3.11). As the field is declared and read in `ATool`, the sources and destinations of the edges are `OObjects` of types `SelTool`, `BoardATool`, `DragTrack` and `SelTrack` that are subtypes of `ATool`.

**One Field Write and N edges.** One field write `r.f = x` in a class `C` may be associated with N dataflow export edges in the `OGraph` from the `OObject` of the enclosing type of the expression to the `OObjects` of the type of receiver or subtypes of the receiver type that are in reachable domains after the formal domain parameters are bound to the actual domains.

The `OGraph` has N export dataflow edges that trace to the same field write expression in the code. The receiver of the expression is of type the  $C_{rec}$ . The field of type the  $C_f$  associated with the expression may be declared in  $C_{rec}$  or types that are subtype compatible with  $C_{rec}$  (Figure 3.9).

Once we have established that one field write in the code may relate to many abstract dataflow edges in the `OGraph`, we discuss that DiffMetrics, 1FWnE, that count number of edges due to the same field write in the code satisfying a specific conditions.

$E_i = \langle Osrc_i, Odst_i, Oflow_i, EXP \rangle$   
 $E_j = \langle Osrc_j, Odst_j, Oflow_j, EXP \rangle$   
*s.t*  
 $E_i \neq E_j$   
 Edges due to the same field write expression  
 $r.f = x \in traceToCode(E_i) \cap traceToCode(E_j)$   
 Receiver of the field write  
 $r : T_{rec}, T_{rec} = C_{rec} \langle \overline{p_{rec}} \rangle$   
 Variable or field  
 $x : T_x, T_x = C_x \langle \overline{p_x} \rangle$   
 Field Type  
 $f : T_f, T_f = C_f \langle \overline{p_f} \rangle$   
 The field f is declared in  $C_{declr}$   
 $(T_f, f) \in fields(T_{declr}), T_{declr} = C_{declr} \langle \overline{p_{declr}} \rangle$   
 Receiver may be subtype of type that declares the field  
 $C_{rec} <: C_{declr}$   
 x may be subtype of field type  
 $C_x <: C_f$

FIGURE 3.9: 1 field write may have N edges in the OGraph.

$\{E_i\}$  where  $(r.f = x) \in traceToCode(E_i)$   
*s.t*  
 $r : T_{rec}, T_{rec} = C_{rec} \langle \overline{p_{rec}} \rangle$   
 $Odst_i = \langle A_{rec_i} \langle \overline{D_{rec_i}} \rangle \rangle$   
 Destination OObject types and receiver types are subtype compatible  
 $A_{rec_i} <: C_{rec} \mid C_{rec} <: A_{rec_i}$   
 and  $A_{rec_i} \neq C_{rec}$  and  $flag = EXP$

FIGURE 3.10: Formal definition of 1FwnE.

**1FWnE** collects the sets of OEdges such that the type of the source OObjects are subtype of the receiver type of a field write expression (Figure 3.10).

An object of type **Figure** is written into the field **draggedFig** in **SelTool**. The



```

abstract class ATool<OWNER, M, C> implements Tool<OWNER, M, C> {
  MiniDraw<M, C> md = new MiniDraw();
  mouseDown() {
    // Example: 1FRnE
    x = md.editor;
    // Edges due to the above field read expression
    E1 = <dragTrack, dragTrack, window, IMP>
    E2 = <selTool, selTool, window, IMP>
    E3 = <baTool, baTool, window, IMP>
    E4 = <selTrack, selTrack, window, IMP>
  }
}

class SelTool<OWNER, M, C> extends ATool<OWNER, M, C> {
  Figure<M, M, C> draggedFig;
  void mouseDown(...){
    Drawing<M, C> model = editor().drawing();
    // Example: 1FWnE
    draggedFig = model.findFigure(...)
    // Edges due to the above field write expression
    E5 = <dragTrack, dragTrack, figure, EXP>
    E6 = <selTool, selTool, figure, EXP>
  }
}

class BoardATool<OWNER, M, C> extends ATool<OWNER, M, C> { ... }
class DragTrack<OWNER, M, C> extends ATool<OWNER, M, C> { ... }
class SelTrack<OWNER, M, C> extends ATool<OWNER, M, C> { ... }
class Breakthrough {
  // M maps to MODEL
  // C maps to CTRL
  domain MODEL, CTRL;
  // Subtypes of Tool in reachable domains
  SelTool<CTRL, MODEL, CTRL> selTool = ...;
  BoardATool<CTRL, MODEL, CTRL> baTool = ...;
  DragTrack<CTRL, MODEL, CTRL> dragTrack = ...;
  SelTrack<CTRL, MODEL, CTRL> selTrack = ...;
  // Subtypes of Figure in reachable domains
  BDrawing<MODEL, MODEL, CTRL> bdrawing = ...;
  BFigure<MODEL, MODEL, CTRL> bfigure = ...;
}

1FRnE: x = editor = 4
1FWnE: draggedFig = model.findFigure(...) = 2

```

FIGURE 3.11: Examples illustrating 1FRnE, 1FWnE.

OGraph shows two export edges E5 and E6 from the type that writes the field SelTool to the type that declares the field, SelTool exporting flow OObjects of types BFigure and BDrawing that are subtypes of the field type in reachable domains after the formal domain parameters are bound to the actual domains (Figure 3.11).

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

FIGURE 3.12: Formal definition of TMO.

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

FIGURE 3.13: Formal definition of PO.

### 3.2.2 DiffMetrics in Many-To-One Category

**TMO**<sup>2</sup> collects **OObject**  $O$  that satisfy the following condition in Figure 3.12.

### 3.2.3 DiffMetrics in MismatchedLocation Category

**PO**<sup>3</sup> computes the percentage of pulled objects compared to all **OObjects** in the **OGraph** (Figure 3.13).

### 3.2.4 DiffMetrics in Precision Category

**PTEP**<sup>4</sup> computes the precision ratio and precision factor for every field associated with points-to edges in the **OGraph** (Figure 3.14).

For example, the **OGraph** shows three points-to edges for the field declaration **fChild** of the type **Tool** in **SelectionTool**. The analysis only shows the types **DragTrack**, **SelTrack** and **NullTool** as subtypes of the **Tool** when a typical type hierarchy shows all the seven types that implement the type **Tool** as subtypes.

---

<sup>2</sup>The definition of TMO is included here for completeness. It includes contributions by co-authors [2].

<sup>3</sup>The definition of PO is included here for completeness. It includes contributions by co-authors [2].

<sup>4</sup>The definition of PTEP is included here for completeness. It includes contributions by co-authors [2].

$$\begin{aligned}
D_{dst} &= \text{mapFtoA}(O_X, p) \text{ // map domain in the code to } O\text{Domain} \\
\text{precisionRatio}(C \langle \bar{p} \rangle f) &= \frac{|\text{OOGPossibleSubTypes}(C, D_{dst})|}{|\text{AllPossibleSubClasses}(C)|} \\
\text{PTEP\_F}(C \langle \bar{p} \rangle f) &= 1 - \text{precisionRatio}(C \langle \bar{p} \rangle f)
\end{aligned}$$

FIGURE 3.14: Formal definition of PTEP.

$$\begin{aligned}
D_{src} &= \text{mapFtoA}(O_{src}, p_s) \text{ // map domain in the code to } O\text{Domain} \\
D_{dst} &= \text{mapFtoA}(O_{dst}, p_d) \text{ // map domain in the code to } O\text{Domain} \\
D_{flw} &= \text{mapFtoA}(O_{flw}, p_f) \text{ // map domain in the code to } O\text{Domain} \\
\text{precisionRatio}(C \langle \bar{p} \rangle v) &= \frac{|\text{OOGPossibleSubTypes}(C, D)|}{|\text{AllPossibleSubClasses}(C)|} \\
\text{DFEP\_F}(C \langle \bar{p} \rangle v) &= 1 - \text{precisionRatio}(C \langle \bar{p} \rangle v) \\
\text{where} \\
p &::= p_s \mid p_d \mid p_f \\
D &::= D_{src} \mid D_{dst} \mid D_{flw}
\end{aligned}$$

FIGURE 3.15: Formal definition of DFEP.

**DFEP** computes precision ratio and precision factor for every variable that may be a receiver of a method invocation or field read or field write, actual arguments of a method invocation that is associated with dataflow edges in the **OGraph** (Figure 3.15).

For example, a typical type hierarchy shows all possible subtypes of **Tool** but **OGraph** shows only three subtypes of **Tool**: **DragTrack**, **NullTool**, **SelfTrack** for the return type of **Tool** of the method **tracker()** in **SelfTool** (Figure 3.16).

### 3.3 Quantitative Analysis of DiffMetrics

We quantitatively analyze the DiffMetrics on on eight systems (training set systems) we previously annotated totalling 100 KLOC. We identify the data points of the DiffMetrics that are above a threshold, which we set to be the 75th percentile, as the *outliers*. The analysis identifies the outlier data points in the detailed tabular output files with a special symbol, so we can manually inspect them. The analysis

### Inheritance Type Hierarchy

```

+- Object
  +-Tool
    +-ATool
      +-BoardATool
      +-DragTrack
      +-SelTrack
      +-SelTool
    +-NullTool

```

### Annotated code fragment

```

class SelTool<OWNER, M, C>
  extends ATool<OWNER, M, C> {
  domain TRACK;
  // points-to edge
  Tool<TRACK, M, C> fChild;
  Tool<TRACK, M, C> tool;
  void mouseDown(...){
    tool = new NullTool();
    // dataflow edge
    tool = tracker();
  }
  Tool<TRACK, M, C> tracker() {
    SelTrack<TRACK, M, C> selTrack = ...;
    DragTrack<TRACK, M, C> dragTrack = ...;
  }
}

class NullTool<OWNER, M, C> extends Tool<OWNER, M, C> {
  ...
}
class DragTrack<OWNER, M, C> extends ATool<OWNER, M, C> {
  ...
}
class SelTrack<OWNER, M, C> extends ATool<OWNER, M, C> {
  ...
}

PTEP_F =  $1 - \frac{3}{7} = 0.57$ 
DFEP_F =  $1 - \frac{3}{7} = 0.57$ 

```

FIGURE 3.16: Examples illustrating PTEP, DFEP.

also computes the maximum, average and other descriptive statistics on the DiffMetrics. The analysis also generates short output files that we load into a statistical analysis package, R. We then wrote scripts to compute p-values and other statistics and generate the output tables (Figure 3.17).

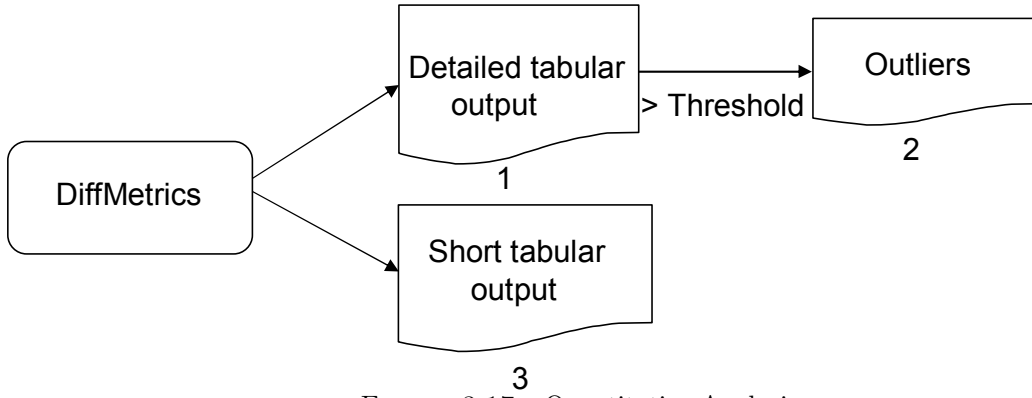


FIGURE 3.17: Quantitative Analysis.

## Credits and Acknowledgements

I would like to thank my advisor and other students, Radu Vanciu and Andrew Giang for their work by the time I joined the project: the previous metrics, including their formalization, implementation, and evaluation on the eight systems, the R scripts, the framework design in the ArchMetrics project, which simplified the task of defining new metrics that are based on dataflow edges. I also would like to thank Radu for supporting the extraction analysis to add missing information that I needed for the dataflow edges.

## Chapter 4: Approach Overview

Adding annotations to a system to extract the **OGraph** from the annotated code can be time consuming. Moreover, the DiffMetrics may indicate that the abstract runtime structure is not be significantly different from the code structure. In such a case, the developers may not benefit from the **OGraph** during program comprehension or code modification tasks to justify the time spent adding annotations. We discuss a principled approach to define a proxy that identifies systems for which the abstract runtime structure is different from the corresponding code structure.

### 4.1 A Principled Approach

As a first step, we compute the DiffMetrics on set of eight systems that were previously annotated. These eight systems constitute the training set. We use the DiffMetrics computed on the training set to build a model to predict characteristics that effect the DiffMetrics based on other code metrics and code patterns that are determined by the proxy. Then, we run the proxy on a system that is not part of the training set to predict if the abstract runtime structure is significantly different from the code structure. The proxy has two phases: Model building and Prediction. As the proxy does not need the developers to add annotations to the system, it is *lightweight*.

**Model Building.** We identify code patterns that may be associated with the outliers of the DiffMetrics. Then we identify other metrics that can be computed from the code (code metrics) which correlate with the DiffMetrics. Then we build a model that indicates what code patterns may lead to statistically significant values of the DiffMetrics, also what code metrics positively correlate with some DiffMetrics. The proxy consists of a visitor that detects the code patterns and simple tools that compute code metrics. The proxy is used on any system.

TABLE 4.1: Research Hypotheses.

<b>H1</b>	Code patterns as predictors
<b>H2</b>	Code metrics as predictors

**Prediction.** We run the visitor to detect the code patterns and compute code metrics on a system for which we want to extract the abstract runtime structure. The number of each identified code patterns and the values of the code metrics predicts if the abstract runtime structure of this system may be significantly different from the code structure. The proxy may indicate that extracting the abstract runtime structure may not be worthwhile. But, running the proxy is less manual effort compared to adding annotations to the code.

## 4.2 Research Hypotheses

We derive two research hypotheses H1 and H2 (Table 4.1) that the proxy tests to predict if the abstract runtime structure of a system is significantly different from the code structure.

### 4.2.1 Testing H1

As a first step (testing H1), we run visitors on the AST and scan for code patterns. The visitors are called the No-Annotation Visitors (Figure 4.1). The No-Annotation visitors reuse code patterns that are previously identified based on our analysis of the DiffMetrics and their outliers. We initially picked MD to examine the outliers of all the DiffMetrics. We chose MD since it is small in size (1.5 KLOC), yet it uses many object-oriented concepts. Moreover, the system has extensive documentation (a textbook), that explains various design patterns and frameworks that are designed into the system. We manually traced the `OObjects` and `OEdges` associated with the outliers to their corresponding code elements in the system. We identified code patterns such as containers, inheritance. The code patterns may also be based on the system

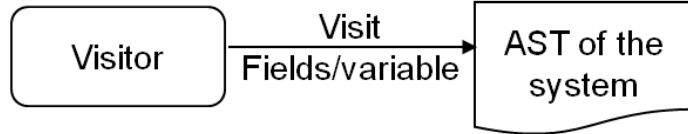


FIGURE 4.1: No-Annotation Visitors.

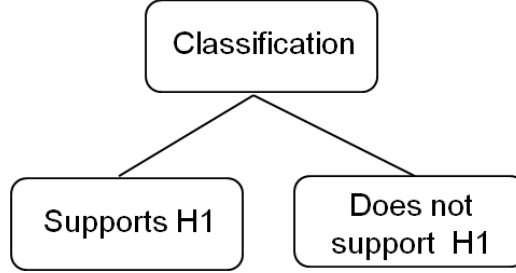


FIGURE 4.2: Test H1 based on the Classification.

specific types of fields or variables, or Java library types of the associated `OObjects`. We implement a model that extracts all the system specific framework types and the Java library types. By manual inspection of the code, we collect data types such as `Rectangle`, `Points` etc. Every system has a file *metrics\_map.xml*. A small portion of the file for MD is shown in Figure 4.3. The code patterns are based on the predefined list of framework types, library types and data types from the *metrics\_map.xml*. The identified patterns from the inspection of MD are grouped into different categories (Table 4.2). To test H1, we designed the No-Annotation visitors. The visitors use the same code patterns to classify code elements in the system that has no annotations added to the code. The number of code elements in each classification supports or does not support H1 (Figure 4.2).

### 4.2.2 Testing H2

As a second step (testing H2), we compute some *code metrics* that measure abstractness and depth of inheritance, among others. Computing such metrics on medium or large-scale systems is possible due to the availability of many open-source and commercial tools. We used an Eclipse plugin *Metrics*<sup>1</sup>. However, this tool does

---

<sup>1</sup><http://metrics.sourceforge.net/>



```

<?xml version="1.0" encoding="utf-8"?>
<model id="0">
  <dataTypes id="33">
    <string id="34">java.awt.Point</string>
    <string id="35">java.sql.Timestamp</string>
    <string id="36">java.util.Date</string>
    <string id="37">java.awt.Dimension</string>
    <string id="38">java.awt.Rectangle</string>
    ...
  </dataTypes>
  <appFrameworkTypes id="39">
    <packageType id="40" packageName="md.frk" typeName="md.frk.Figure"/>
    <packageType id="41" packageName="md.std" typeName="md.std.StandardDrawing"/>
    ...
  </appFrameworkTypes>
</model>

```

FIGURE 4.3: Small portion of metrics\_map from MD.

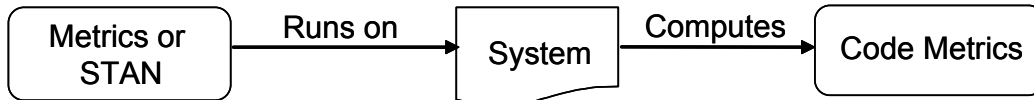


FIGURE 4.4: Compute Code Metrics.

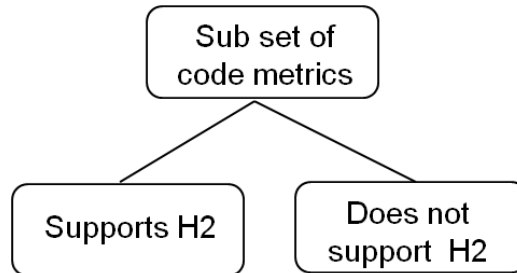


FIGURE 4.5: Test H2 based on the range of the Code Metrics.

not compute some of the code code metrics like coupling between classes and access to foreign data that use links between classes to define the detailed architecture of the system. So, we used another tool, Structure Analysis for Java (*STAN*)<sup>2</sup> that computes another set of code metrics. Both the tools generate files with the computed numbers, average measures for all the packages and the standard deviation for the metrics. From the computed metrics, we pick a subset of them (Figure 4.4) that correlates with the DiffMetrics. The range of the subset either support or not support H2 (Figure 4.5).

---

<sup>2</sup><http://stan4j.com/sample-report.html>

TABLE 4.2: Code Patterns identified from MD.

	<b>Application Specific</b>	<b>Java Library</b>	<b>Predefined List</b>
<b>Code Patterns</b>	Container of General Type	Field/ Variable of Java Library Type	Field/ Variable of Framework Type
	Container of Application Type	Container of Java Library Type	Field/ Variable of Data Type
	Field/ Variable of General Type	Java Exception	
	Custom Exception	Inheritance	
	Inheritance		

When both steps match the ranges and the classifications that we derived from our previous analysis, this may be a good indicator that the abstract runtime structure may be different from the code structure. In that case, one may consider adding annotations to this system and extracting the **OGraph** in order to use that for program comprehension or code modification tasks. In the next two chapters, we discuss the No-Annotation visitors and the correlations between code metrics and the DiffMetrics in detail.

## Chapter 5: Code Patterns as Predictors

We discuss the code patterns, the classification of the outliers of the DiffMetrics and introduce our training set systems. Next, we discuss the first step of the proposed proxy, the No-Annotation visitors and how the visitors may be used to test H1.

**Training set Systems.** Over a period of time, we have collected systems ranging in size from 1 to 18 KLOC. They are large enough to be interesting but yet small enough to be able to analyze them in a limited time. We have the design information for some of the systems that may be used to express design intent. They are from a wide variety of application domains. Some are desktop applications for board games, others process domain-specific data. Yet, others are client-server applications and one is an encrypt-decrypt application. Also, the systems use design patterns, inheritance, composition, interface implementation, type parameterizations that are common in object-oriented design. Some measures from the code structure of the systems are shown (Table 5.1).

TABLE 5.1: List of Training set Systems: *Number of Types*, *Number of GT* are the number of all types, abstract or Interfaces.

Abbr.	Names	KLOC	Number of Types	Number of GT
MD	MiniDraw	1.4	68	21
CDB	CryptoDB	2.3	47	9
AFS	Apache FtpServer	14.4	173	61
DL	DrawLets	8.8	165	54
PX	Pathway-Express	36	300	62
JHD	JHotDraw	18.0	306	65
HC	HillClimber	15.6	171	35
APD	Aphyds	8.2	70	15
<b>Total</b>		104.7		

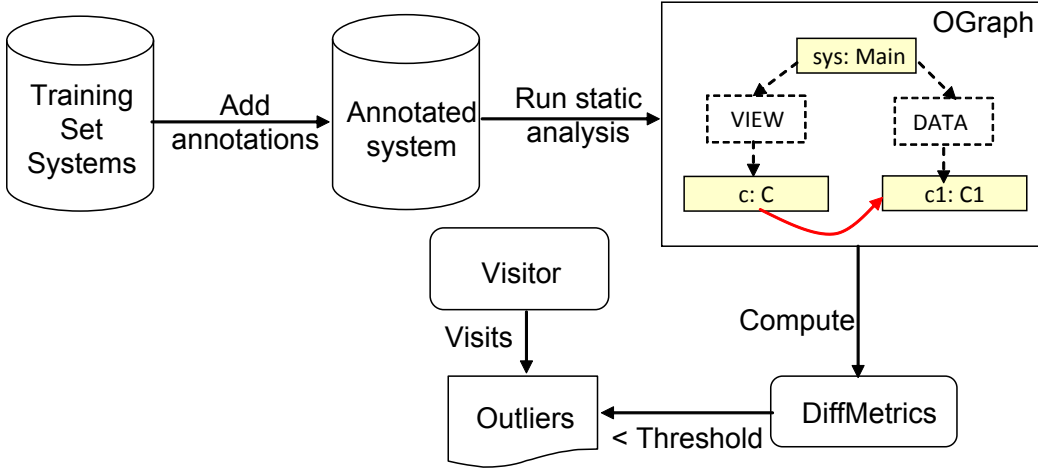


FIGURE 5.1: Visitors visit outliers of the DiffMetrics.

## 5.1 Visitors and Classifications of DiffMetrics

For all the training set systems, we extracted the OGraph from the annotated code. Each of the defined DiffMetrics is associated with a visitor that visits the OObjects and OEdges of outliers of DiffMetrics (Figure 5.1). As an example, the DiffMetrics `1MInE_RecType`, overrides the method `visitOutliers`. This method invokes `visit()` and this method visits OEdges that are outliers collected in the set `HashSet<EdgeInfo>` (Figure 5.2). Each visited outlier is then classified into one of the predefined classifications. Most of the DiffMetrics in MismatchedLocation Category computed on MD did not have any outliers. Manual inspection of these DiffMetrics on the other systems also indicated that there were few or no outliers. We did not investigate such DiffMetrics in this experiment.

### 5.1.1 Classifiers of Outliers

Most of the previously identified code patterns are the classifications. However, for some code patterns like Inheritance, we define slightly varying classifications to classify the outliers associated with the DiffMetrics grouped under One-To-Many Category and Many-To-One Category. The OObjects and OEdges collected from the visitors are classified into one of the classifications. Each of the DiffMetrics have slightly

TABLE 5.2: WABW Outlier.

Outlier	Size	Type	Triplet 1	Triplet 2
X	3	List<A1>	<List<A1>, DATA1, C1>	<List<A1>, DATA2, C3>
			<List<A1>, DATA2, C3>	<List<A1>, DATA2, C4>
			<List<A1>, DATA1, C1>	<List<A1>, DATA2, C4>

different classifications. So, we have one file for each of the DiffMetrics that defines the logic for the classification. As an example, the classification of the visited OEdges associated with the outliers of 1MInE\_RecType are in the type Q\_1MInE\_RecType (Figure 5.2). We discuss the classifications of outliers of the DiffMetrics grouped by their category.

### Classification of Outliers in One-To-Many Category

- I. *Container of general type (CGT)*. An OGraph can express different design intent by mapping the OObjects of the same type in multiple domains to distinguish the context in which the containers of the same type are used in the system. The outliers of WAWB are usually classified into this classification. WAWB measures the number of instances of List<A1> that are in different parent OObjects and the output is presented in Table 5.2 for the code in Figure 5.3. 'X' in the outlier column indicates that the instance of List<A1> associated with corresponding triplet is an outlier. Our visitors visit the outlier and classifies listener of type List<A1> as CGT (Figure 5.3). The hierarchy of OObjects from the extracted OGraph is in Figure 5.4.
- II. *Inheritance*. DiffMetrics of the DFMetrics Subcategory, that relate dataflow edges in the OGraph to a method invocation, or field read, or field write expression, have outliers when the receiver type of such expressions, or the type of the formal parameters of the method declaration or the return type of the method declaration are part of some inheritance hierarchy. We define slightly varying classifications that are associated with the code pattern inheritance for

such DiffMetrics.

(a) Inheritance Type 1 (DFIT1):

The expression and the associated **OEdges** are classified into this classification when the type of the destination **OObjects** of the **OEdges** are subtypes of receiver type or enclosing type of the corresponding expression in reachable domains after the formal domain parameters are bound to the actual domains. Also, the type of the source **OObjects** of the **OEdges** have only one subtype or the type of the **OObjects** are of subtypes that are not in reachable domains after the formal domain parameters are bound to the actual domains.

The expression `a1.m1(str)` has two **OEdges** (E1 and E2) in the **OGraph** from the **OObject** of type **C1**, a subtype of the enclosing type of the method invocation to different destination **OObjects** that are subtypes of the receiver after the formal domain parameter of **A1** is bound to the actual domain **DATA1** in **main** (Figure 5.5). However, there is no edge from **OObject** of type **C1** to **OObject** of one of the subtypes of **A1** **C5**. The analysis that extracts the **OGraph** picks only the **OObjects** that are subtypes of **A1** and in domain **DATA1** as the formal domain parameter **D1** maps to the actual domain **DATA1**. The output of the DiffMetrics is illustrated (Table 5.3). As the destination **OObjects** are subtypes of receiver type, most of the outliers of **1MInE\_RecType** is classified into this classification.

(b) Inheritance Type 2 (DFIT2):

The expression and the associated **OEdges** are classified into this classification when the type of the source **OObjects** of the **OEdges** are subtypes of receiver type or enclosing type of the corresponding expression in reachable domains after the formal domain parameters are bound to the actual

domains. Also, the type of destination **OObjects** of the **OEdges** have only one subtype or the type of the **OObjects** are of subtypes that are not in reachable domains after the formal domain parameters are bound to the actual domains.

The expression **a2.m2(str)** and the associated **OEdges** is classified into this classification. The **OGraph** has two edges (E3 and E4) from **C3** and **C4** that are subtypes of the enclosing type of the expression to the same destination **OObject** of type **C1** (Figure 5.5).

(c) Inheritance Type 3 (DFIT3):

The expression and the associated **OEdges** are classified into this classification when the type of source **OObjects** and the destination **OObjects** of the **OEdges** are subtypes of receiver type or enclosing type of the corresponding expression in reachable domains after the formal domain parameters are bound to the actual domains.

The method invocation **a1.m1(str)** in the enclosing type **A3** and the associated **OEdges** is classified into this category. The **OGraph** has four edges (E5, E6, E7 and E8) from **OObjects** of types **C5** and **C6** that are subtypes of **A3** in reachable domains after the formal domain parameter **D2** is mapped to the actual domain **DATA2** in main to **OObjects** of types **C3** and **C4** that are subtypes of **A1** in reachable domains after the formal domain parameter **D1** is mapped to the actual domain **DATA1** (Figure 5.5).

(d) Inheritance Type 4 (DFIT4):

The expression and the associated **OEdges** are classified into this classification when the type of flow **OObjects** of the **OEdges** are subtypes of actual argument or return type of a method invocation in reachable domains after the formal domain parameters are bound to the actual domains. The source **OObjects** and the destination **OObjects** of the **OEdges** are from the

TABLE 5.3: 1MInE\_RecType Outlier.

Outlier	Expression	Size	Type
X	2	a1.m1(str)	c1 -> c3 [Export str] c1 -> c3 [Export str]

same OObjects of the same types.

(e) Inheritance Type 5 (DFIT5)):

The classification is similar to DFIT4 with a slight change. The flow OObjects associated with an expression are mapped to different domains in the OGraph.

III. *Exception types.* The OObjects of an exception type may be mapped to different domains in the OGraph since the same exceptions may be thrown in different contexts.

### Classification of Outliers in Many-To-One Category

I. *Framework types.* Many concrete types that refer to a predefined list of framework type in the system may have object creation expressions. However, many of these object creation expressions may map to a single OObject in the OGraph. This indicates that the OObject of the framework type is shared amongst all types that refer to it.

II. *Inheritance.* When a type extends other types, then OObjects its super types are merged by the OObject of the concrete type in the OGraph. We define classification that distinguish OObjects that are of type of application, framework, Java Library or Exception. The outliers of TMO are classified into any of the below classifications.

(a) Type <: Type: When a concrete type A extends another concrete type, then the OObjects of subtypes of A are merged by the OObject of type A in



the **OGraph**. Most of the outliers of the training set systems are classified into this classification.

- (b) Type <: Framework Type (Type <: FKT): When a concrete type **A** extends a framework type in a predefined list, then the **OObjects** of subtypes of **A** are merged by the **OObject** of type **A** in the **OGraph**.
- (c) Type <: Java Library Type (Type <: Java Type): When a concrete type **A** extends a Java library type, then the **OObjects** of subtypes of **A** are merged by the **OObject** of type **A** in the **OGraph**.
- (d) Framework Type <: Java Library Type (FKT <: Java Type): When a framework type **F** in a predefined list extends a Java library type, then the **OObjects** of subtypes of **F** are merged by the **OObject** of type **F** in the **OGraph**.

### Classification of Outliers in Precision Category

- I. *Field/Variable of a general type.* The type of the field or variable for which the **OGraph** has more precise information compared with the type hierarchy that uses the AST, is usually a general type. A system may have many subtypes for a general type, however the **OGraph** shows points-to and dataflow edges to only a subset of these subtypes that are in reachable domains after the formal domain parameters are bound to the actual domains. The argument of the method invocation `sel.add(figure)` of the type **Figure** has seven subtypes. But the analysis traces dataflow edges associated with the **OObject** of type **Figure** (Figure 5.6) and shows only two concrete types that are in reachable domains after the formal domain parameters are bound to the actual domains (Table 5.4).

TABLE 5.4: DFEP Outlier.

Outlier	Expression	Type	DFTYPE	No. All_Subtypes	No. Subset_Subtypes	Precision
X	<code>sel.add(figure)</code>	Figure	Argument	7	2	0.71

## 5.2 No-Annotation Visitors

The above experiment indicated that code patterns, especially containers of general types, containers of types, fields of general types and inheritance among others influence the DiffMetrics and implicitly the runtime structure too. As code patterns do not depend on the **OGraph**, both the code patterns and classifications can be identified from the code of a new system with no annotations. We then used the classifications of the new system to predict if the DiffMetrics would indicate runtime structure may be significantly different from the code structure, the first step in our approach. We generalized the visitors and the classifiers of the outliers to visit and classify every field and variable of this system (Figure 5.8). We discuss the visitors and the classifiers in detail (Figure 5.7).

**Visitors of Field/Variables.** The visitors in Figure 5.1 that visit the outliers of the DiffMetrics traverse only the nodes and edges in the **OGraph**. As we do not have an extracted **OGraph** for the new system, we need to visit all the fields and variables that are declared. Hence, we generalize the visitors to traverse the entire AST of the system. The generalized visitors scan fields and variables from various expressions in the AST such as field declarations, method invocations, field reads and field writes.

**Classifiers of Field/Variables.** The visited fields and variables are classified into one of the previously identified code patterns. The classifications based on the type of the field or the variable are reused. However, the classifications that is based on the code pattern inheritance are very specific to **OObjects** and **OEdges** in the **OGraph**. So we define two classifications that work for a new system. The code patterns and classification are presented (Table 4.2). We illustrate the generalized code patterns and the classification (Figure 5.9).

TABLE 5.5: Code Patterns and Classification.

Code Patterns	Classifications
Containers of	Types Framework Types
Inheritance	Inheritance Type 1 Inheritance Type 2
General Types	Fields Other Classification Unclassified

1. Inheritance Type 1 (IT1): When the visited field or variable is of a concrete type that has other concrete subtypes, then such a field or variable is classified into this category.
2. Inheritance Type 2 (IT2): When a field or variable is of a general type and has other subtypes (concrete or general types), then such a field or variable is classified into this category.

### 5.3 Using H1 as Predictor

After the classification of the fields and variables of the new system, we analyze the classification summary to test H1. We compare the percentage of the total classified fields and variables with the total numbers that are not classified. If more than the defined threshold (50%) of the total fields and variables are classified, we inspect the number of fields and variables in each classification.

When a field or variable is classified as CGT or CT, the runtime structure may have many different instances of the same container types playing different roles at runtime. Distinguishing containers of the same type playing different roles may help the developers during program comprehension and code modification tasks that require understanding or modifying container elements that represent a specific role in the system. When a field or variable is classified as FGT, the runtime structure may have multiple instances of different concrete types of this general type that may play

different roles at runtime. Runtime structure of the system may have more precise information on the subtypes of those fields or variables compared to the type hierarchy that uses AST. When a field or variable is classified as IT1 and IT2, runtime structure may indicate that different concrete types of the same inheritance hierarchy may play different roles at runtime. We map the code from MD the participants struggled or explored often from transcripts collected in an experiment conducted [4] to the outliers from the DiffMetrics. The analysis indicated that the outliers of the DiffMetrics that are fields or variables of a general type, trace to the portion of the code that is difficult to comprehend. The details of the analysis are in Appendix A: Transcript Analysis. When a field or variable is classified under Composition, IT1 and IT2, this also indicates that the system uses object-oriented concepts, e.g., inheritance, composition, and abstractness. All the above classifications are indicators that the abstract runtime structure of the system may be significantly different from its code structure. However, if the No-Annotation Visitors classify the majority of fields or variables into Field/Variable of a Library type or Field/Variable of a Data type, then the code structure may be adequate for most program comprehension tasks.

If the No-Annotation Visitors classify the majority of fields and variables as Container of general type or Container of type, Field/Variable of a general type, Inheritance Type 1, Inheritance Type 2, or Composition, we then compute the code metrics of the system.

```

abstract EdgeMetricBase {
    void visitOutliers(Writer writer, Set<EdgeInfo> outliers) {
    }
}

class HowManyEdgesToMethodInvok_RecType extends EdgeMetricBase {
    @Override
    void visitOutliers(Writer writer, Set<EdgeInfo> outliers) {
        // The visited OEdges are classified into DFIT1, DFTI2 etc
        Q_1MinE_RecType qVisit = new Q_1MinE_RecType(writer, outliers, shortName);
        qVisit.visit();
    }
}

abstract class Q_Base{
    abstract void visit();
}

class Q_1MinE_RecType extends Q_Base {

    /* DFIT1: Outliers when destination OObjects are many
    * DFIT2: Outliers when source OObjects are many
    * DFIT3: Outliers when both source OObjects and destination OObjects are many
    * DFIT4: Outliers when flow OObjects are many */

    int numDFIT1 = 0;
    int numDFIT2 = 0;

    @Override
    void visit() {
        Set<Type> sourceOObjects = new HashSet<Type>();
        Set<Type> destinationOObjects = new HashSet<Type>();
        // Traversing through the outliers
        for (EdgeInfo edgeInfo : outliers) {
            // Outliers are instance of DiffMetrics 1MinE_RecType
            if (edgeInfo instanceof HowManyEdges_MIRecType) {
                HowManyEdges_MIRecType MI_edgeInfo = (HowManyEdges_MIRecType) edgeInfo;
                // Get set of OEdges
                Set<IElement> setEdges = MI_edgeInfo.getElems();
                for (IElement eachEdge : setEdges) {
                    // Get source OObject and destination OObject types for each OEdge
                    if (eachEdge instanceof ODFEdge) {
                        Type sourceType = ((ODFEdge) eachEdge).getOsrc().getC();
                        Type destType = ((ODFEdge) eachEdge).getOdst().getC();
                        sourceOObjects.add(sourceType);
                        destinationOObjects.add(destType);
                    }
                }
            }
            // Logic for classification here
            // DFIT1
            if (sourceOObjects.size() == 1 && destinationOObjects.size() > 1) {
                numDFIT1++;
            }
            // DFIT2
            else if (sourceOObjects.size() > 1 && destinationOObjects.size() == 1) {
                numDFIT2++;
            }
        }
    }
}

```

FIGURE 5.2: Pseudo-code of the Visitors.

```

abstract class A1<OWNER> {
}
class C2<OWNER> extends A1<OWNER>{
}
class C1<D1> {
  void C1() {
    // Container of A1
    List<OWNED, <A1><D1>> listener
      = new ArrayList<A1>();

    // Field of C2
    C2<D1> f2 = new C2();
  }
}
class C4<D2> {
  void C4 {
    // Container of A1
    List<OWNED, <A1><D2>> listener
      = new ArrayList<A1>();
  }
}

class C3<D2> {
  void C3() {
    // Container of A1
    List<OWNED, <A1><D2>> listener
      = new ArrayList<A1>();
  }
}

class main {
  domain DATA1, DATA2;
  // D1 maps to DATA1
  // D2 maps to DATA2
  C1<DATA1> c1 = ...;
  C3<DATA2> c3 = ...;
  C4<DATA2> c4 = ...;
}

listener: ArrayList<A1> is classified into CGT
f2: C2 is classified Field/Variable of FKT

```

FIGURE 5.3: Classification of OObjects.

```

+- main:Main
+- DATA1
  +- c1:C1
    + OWNED
    | +- listener:ArrayList
+- DATA2
  +- c3:C3
    + OWNED
    | +- listener:ArrayList
+- c4:C4
  + OWNED
  | +- listener:ArrayList

```

FIGURE 5.4: Hierarchy of OObjects.

```

abstract class A1<D1> {
    // Method declaration
    void m1(String<shared> str) {
    }
    void method2() {
        String str;
        // Method invocation
        A2<D1> a2;
        a2.m2(str)
        // Edges
        E3 = <c3, c1, str, EXP>
        E4 = <c4, c1, str, EXP>
    }
}

abstract class A3<D1> {
    void method3() {
        String str;
        A1<D1> a1;
        // Method invocation
        a1.m1(str)
        // Edges
        E5 = <c5, c3, str, EXP>
        E6 = <c6, c3, str, EXP>
        E7 = <c5, c4, str, EXP>
        E8 = <c6, c4, str, EXP>
    }
}

class main {
    domain DATA1, DATA2;
    void run() {
        //D1 maps DATA1
        //D2 maps to DATA2
        C1<DATA1, DATA1> c1 = new C1();
        C3<DATA1, DATA1> c3 = new C3();
        C4<DATA1, DATA1> c4 = new C4();
        // Subtype of A1 not in reachable domains
        C5<DATA1, DATA2> c5 = new C5();
        // Subtype of A3 in reachable domains
        C6<DATA1, DATA2> c6 = new C6();
        C7<DATA1, DATA2> c7 = new C7();
        // Subtype of A3 not in reachable domains
        C8<DATA1, DATA1> c8 = new C8();
    }
}

abstract class A2<D1> {
    // Method declaration
    void m2(String<shared> str) {
    }
    void method1() {
        String str;
        A1<D1> a1;
        // Method invocation
        a1.m1(str);
        // Edges
        E1 = <c1, c3, str, EXP>
        E2 = <c1, c4, str, EXP>
    }
}

class C1<OWNER, D1> extends A2<D1>{
    // Method declaration
    void m4(A1<D1> a1) {
    }
    void method4() {
        // Method invocation
        c1.m4(a1);
        // Edges
        E9 = <c1, c1, c3, EXP>
        E10 = <c1, c1, c4, EXP>
    }
}

class C3<OWNER, D1> extends A1<D1> {
}

class C4<OWNER, D1> extends A1<D1> {
}

class C5<OWNER, D2> extends A1<D1> {
}

class C6<OWNER, D2> extends A3<D1> {
}

class C7<OWNER, D2> extends A3<D1> {
}

class C8<OWNER, D1> extends A3<D1> {
}

```

a1.m1(str) and E1, E2 are classified into DFIT1  
 a2.m2(str) and E3, E4 is classified into DFIT2  
 a1.m1(str) and E5, E6, E7, E8 is classified into DFIT3

FIGURE 5.5: Classification of OEdges and Expressions DiffMetrics under DFMetrics Subcategory.

```

public class StdDrawing<OWNER, M, C> extends CompFigure<OWNER, M, C> {
    // Adds a figure to the current selection.
    void addToSelection(@Domain("M<M, C>") Figure figure) {
        // Method invocation argument
        sel.add(figure);
    }
}
class Breakthrough {
    domain MODEL, CTRL;
    // M maps to MODEL
    // C maps to CTRL
    // Subtypes of Figure in reachable domains
    BDrawing<MODEL, MODEL, CTRL> bdrawing = ...;
    BFigure<MODEL, MODEL, CTRL> bfigure = ...;
}

```

FIGURE 5.6: Classification of fields and variables of DFEP: *No. All\_Subtypes*, *No. Subset\_Subtypes* are the number of all subtypes and number of subtypes the OGraph identified for the field or variable.

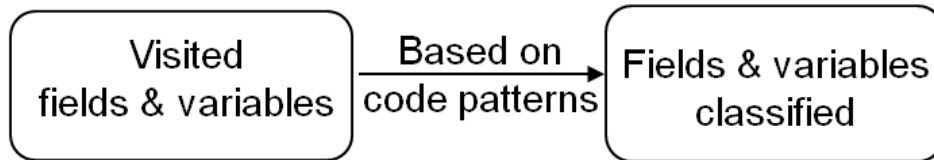


FIGURE 5.7: Classification of Fields and Variables from No-Annotation Visitors.



```

class NoAnnotatMetrics {
    // A private class that visits the AST of the system
    private class Visitor extends ASTVisitor {
        // Visits the VariableDeclarationFragment node in AST
        @Override
        boolean visit(VariableDeclarationFragment node) {
            return super.visit(node);
        }
    }
}
C_AllMetrics extends Q_Base{
    TypeInfo typeInfo = TypeInfo.getInstance();
    int numCGT = 0;
    @Override
    public void visit() {
        QualUtils utils = QualUtils.getInstance();
        for (IVariableBinding eachVarField : variables) {
            fieldTypeBinding = eachVarField.getType();
            Type fieldType = typeInfo.getType(fieldTypeBinding.getQualifiedName());
            if (fieldTypeBinding != null) {
                if (utils.isContainerOfGeneralType(fieldTypeBinding.getQualifiedName())) {
                    numCGT++;
                }
                else {
                    unknown++;
                }
            }
        }
    }
}
}

```

FIGURE 5.8: Pseudo-code of the No-Annotation Visitor.

```

abstract class A1 {
}
class C5 extends A1 {
}
class C1 {
  // Field declaration
  A1 f1;
  //Container of a general type
  List<A1> = new ArrayList<A1>;
}
class C2 extends C1 {
}
class C3 extends C1 {
  // Field declaration
  C2 f2;
}
class C4{
  // Field declaration
  C1 f;
  //Container of a type
  List<C1> = new ArrayList<C1>;
}

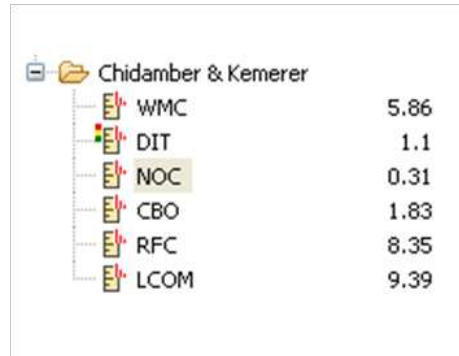
```

C1 f is classified as Inheritance type 1 (IT1)  
 A1 f1 is classified as Inheritance type 2 (IT2)  
 C2 f2 is classified as Composition  
 List<A1> is classified as Container of general type (CGT)  
 List<C1> is classified as Container of type (CT)

FIGURE 5.9: Illustrating Code Patterns and Classification.

## Chapter 6: Simple Metrics as Predictors

We computed code metrics using the tool *Metrics*<sup>1</sup> on the training set systems and outputted them in separate files *src.code.xml* for each the system. The tool *STAN*<sup>2</sup> did not support exporting the computed metrics on the systems in any useful that our analysis may use the data, so we manually outputted the data. The figures 6.2 and 6.1 illustrates a portion of the file along with the metrics computed using both the tools. From the set of metrics in *src.code.xml* and the output of *STAN*. We picked three code metrics that correlates with the DiffMetrics. As the next step, we correlated the average of the metrics with the average of the DiffMetrics across the training set systems (Table 6.2). The correlation and the code metrics computed for a new system is used to test H2.



The image shows a screenshot of the STAN tool's output window. It displays a tree view with a folder icon and the text 'Chidamber & Kemerer'. Below this, there is a list of metrics, each with a small icon and a numerical value. The metrics are WMC (5.86), DIT (1.1), NOC (0.31), CBO (1.83), RFC (8.35), and LCOM (9.39). The NOC row is highlighted with a yellow background.

Metric	Value
WMC	5.86
DIT	1.1
NOC	0.31
CBO	1.83
RFC	8.35
LCOM	9.39

FIGURE 6.1: CBO computed using STAN.

### 6.1 Subset of Code Metrics

Code metrics such as NOC, Lines of Code (LOC), NOV, BLOC and NOV (Figure 6.2) may not indicate if object-oriented concepts e.g., inheritance are used in the system. Other code metrics such as PAR (Figure 6.2) are not associated with an increase or decrease in the number of OObjects and OEdges that affect the Diff-

---

<sup>1</sup><http://metrics.sourceforge.net/>

<sup>2</sup><http://stan4j.com/>

TABLE 6.1: Code Metrics for Training set Systems.

	MD	CDB	AFS	DL	PX	JHD	HC	APD
DIT	1.96	1.58	1.69	3.66	2.40	2.40	3.16	2.39
RMA	0.34	0.00	0.22	0.16	0.07	0.18	0.10	0.01
CBO	1.83	2.73	4.31	3.20	4.05	4.38	4.63	5.85

TABLE 6.2: Average of the Code Metrics for Training set Systems.

<b>Code Metrics</b>	<b>Average</b>
DIT	2.50
RMA	0.13
CBO	3.70

Metrics as method parameters may be primitive types and do not have `OObjects` in the `OGraph`. Also metrics that measure package information are not very helpful to determine properties of the `DiffMetrics`. Other code metrics that compute the code complexity such as the Cyclomatic complexity (VG) are not relevant to be correlated with any of the defined `DiffMetrics` as the analysis of the `OGraph` does not consider control flow information.

We pick the following code metrics that measure abstraction, inheritance and communication between types from the code.

- I. Depth of Inheritance Tree (DIT): This metric measures the number of hops from a type to the topmost level in the class hierarchy. The metric measures inheritance directly.
- II. Abstractness (RMA): This metric measures the number of interfaces or abstract types over the total number of classes in a package (cumulative of all packages in the system). The metric measures abstractness.
- III. Coupling Between Object classes (CBO): This metric measures the number of reference types that occur through method calls, method parameters, return types, thrown exceptions and accessed fields. The metric measures communication between types.

## 6.2 Correlation between DiffMetrics and Code metrics

We implemented R scripts that uses `src.code.xml` and the detailed output of DiffMetrics of all the training set systems to collect the average of code metrics and DiffMetrics of interest. The scripts outputted the values of the code metrics and the DiffMetrics into a file named, `all_metric.csv`. As the tool *STAN* does not output an XML file, we manually edited `all_metric.csv` and added the computed CBO for the training set systems. With metrics (DiffMetrics and code metrics) in one file, we implemented an R script that computes correlations between the two sets of metrics. The script correlates the three code metrics with the DiffMetrics using Pearson's correlation to measure the linear correlation giving a value between +1 and -1. A coefficient of 0 indicates no correlation while a correlation of 0–0.1 trivial, 0.1–0.3 minor, 0.3–0.5 moderate, 0.5–0.7 large, 0.7–0.9 near perfect, and 0.9–1 perfect. The output of the correlation is recorded in `correl.xml` file. From the output, it was evident that not all code metrics correlated with all the defined DiffMetrics. We analyzed the data and discuss the details of correlations between code metrics and DiffMetrics under each category.

### 6.2.1 Correlation with DiffMetrics in One-To-Many Category

- I. DIT with DiffMetrics: Each of the concrete types in different inheritance depth of the same general type may have many `OObjects` mapped to different domains playing different roles at runtime. Thus, as the depth of inheritance hierarchy increases in a package on average, the number of `OObjects` of different concrete types that are mapped to different domains may also increase. The DiffMetrics grouped into DFMetrics Subcategory measure the effect of inheritance. So, the code metric DIT correlates positively with the DiffMetrics in this category.

- II. CBO with DiffMetrics: When communication between two types increase as measured by CBO, the communication between their corresponding OObjects also increase. Thus, this metric correlates positively with the DiffMetrics grouped into the DFMetrics Subcategory.

### 6.2.2 Correlation with DiffMetrics in Many-To-One Category

- I. DIT and RMA with DiffMetrics: When the inheritance depth of a general type and the abstractness of a package increase on average, more distinct types (general or concrete) may be merged by one OObject in the OGraph. So, the code metrics DIT and RMA correlate positively with DiffMetrics in this category. The plot of the correlation between DIT and TMO computed on the training set systems is presented in Figure 6.3. The graph indicates that the two metrics are correlated linearly and if one increases or decreases, the other metrics also follows the same trend.

### 6.2.3 Correlation with DiffMetrics in Precision Category

- I. DIT and RMA with DiffMetrics: When the abstractness of a package on average increase, the OGraph may show more precise subtype information for field or variable type from the points-to or dataflow edges in the OGraph. Also, when the depth of inheritance of a system increase, the OGraph shows points-to and dataflow edges only a subset of subtypes of field or variable type that play the same role in the same context at runtime. Thus the code metrics DIT and RMA correlate positively with DiffMetrics in Precision Category. The plot of the correlation between RMA with PTEP and DFEP on the training set systems is presented in Figure 6.4. The graph indicates that the three metrics correlate

TABLE 6.3: Correlation of Code Metrics with DiffMetrics.

Category	DiffMetric	Code Metric	Correlation
<b>One-To-Many Category</b>	WAWB	DIT	<b>0.80</b>
<b>DFMetrics</b>	1MInE_RecType	DIT	<b>0.99</b>
		CBO	0.65
	1MInE_RetType	DIT	0.50
		CBO	0.59
	1MInE_ArgType	DIT	0.44
		CBO	<b>0.85</b>
	1FRnE	DIT	0.45
		CBO	0.46
	1FWnE	DIT	0.51
		CBO	0.48
<b>Many-To-One Category</b>	TMO	DIT	0.69
		RMA	0.64
<b>Precision Category</b>	PTEP	DIT	<b>0.70</b>
		RMA	<b>0.95</b>
	DFEP	DIT	0.51
		RMA	<b>0.96</b>

linearly and follow the same trend as the other two metrics.

The correlated coefficient of the code metrics with their corresponding DiffMetrics across training set systems is presented in Table 6.3. We highlighted the near perfect and perfect correlated coefficients.

### 6.3 Using H2 as Predictor

If the average of DIT, RMA and CBO for the new system is in the range with the average of the code metrics of the training set systems, this may indicate that the corresponding DiffMetrics for the new system may also follow the same trend (many outliers or few or no outliers) as the systems in the training set. The Table 6.4 shows the range of the code metrics and the trend the DiffMetrics that a new system may follow.

When the DIT is in the range 2.00 – 3.00 for a new system, the runtime structure may be significantly different from the corresponding code structure and DiffMetrics grouped into One-To-Many Category measure the difference. Again, if the DIT and

TABLE 6.4: Average of Code Metrics and corresponding DiffMetrics trends.

Code Metrics	Range of code metrics	DiffMetrics may have outliers
DIT	2.00 – 3.00	One-To-Many Many-To-One Precision
RMA	0.1 – 0.2	One-To-Many Precision
CBO	3.00 – 4.00	DFMetrics

CBO are in the range between 2.00 – 3.00 and 3.00 – 4.00, the DiffMetrics grouped into DFMetrics Subcategory may have many outliers, indicating that the runtime structure is significantly different. Likewise, when the DIT and RMA fall in the range between 2.00 – 3.00 and 0.1 – 0.2, then all other DiffMetrics grouped into Many-To-One Category and Precision Category have many outliers.



```

<?xml version="1.0" encoding="utf-8"?>
<Metric id = "NOC" description="Number of Classes">
  <Values per = "packageFragment" total = "32" avg = "6.4" stddev = "3.666" max = "13">
    <Value name="md.std" package="md.std" value="13"/>
    ...
  </Values>
</Metric>
<Metric id = "DIT" description="Depth of Inheritance Tree">
  <Values per = "type" avg = "1.969" stddev = "1.425" max = "6">
    <Value name="MDApp" source="MDApp.java" package="md.std" value="6"/>
    ...
  </Values>
</Metric>
<Metric id = "RMA" description="Abstractness">
  <Values per = "packageFragment" avg = "0.34" stddev = "0.302" max = "0.846">
    <Value name="md.framework" package="md.framework" value="0.846"/>
    ...
  </Values>
</Metric>
<Metric id = "SIX" description="Specialization Index">
  <Values per = "type" avg = "0.268" stddev = "0.699" max = "3.6">
    <Value name="StdBckgrd" source="StdBckgrd.java" package="md.std" value="3.6"/>
    ...
  </Values>
</Metric>
<Metric id = "MLOC" description="Method Lines of Code">
  <Values per = "method" total = "702" avg = "3.637" stddev = "5.067" max = "41">
    <Value name="IManager" source="IManager.java" package="md.std" value="41"/>
    ...
  </Values>
</Metric>
<Metric id = "NOP" description="Number of Packages">
  <Value value="5"/>
</Metric>
<Metric id = "NOM" description="Number of Methods">
  <Values per = "type" total = "190" avg = "5.938" stddev = "4.841" max = "26">
    <Value name="BFig" source="BFig.java" package="md.boardgame" value="8"/>
    ...
  </Values>
</Metric>
<Metric id = "NOF" description="Number of Attributes">
  <Values per = "type" total = "65" avg = "2.031" stddev = "1.811" max = "6">
    <Value name="BDrawing" source="BDrawing.java" package="md.boardgame" value="5"/>
    ...
  </Values>
</Metric>
<Metric id = "PAR" description="Number of Parameters" max="5" hint="Pass an object">
  <Values per = "method" avg = "1.197" stddev = "1.049" max = "4">
    <Value name="BFigure" source="BFigure.java" package="md.boardgame" value="4"/>
    ...
  </Values>
</Metric>

```

FIGURE 6.2: Sample src.code from MD.

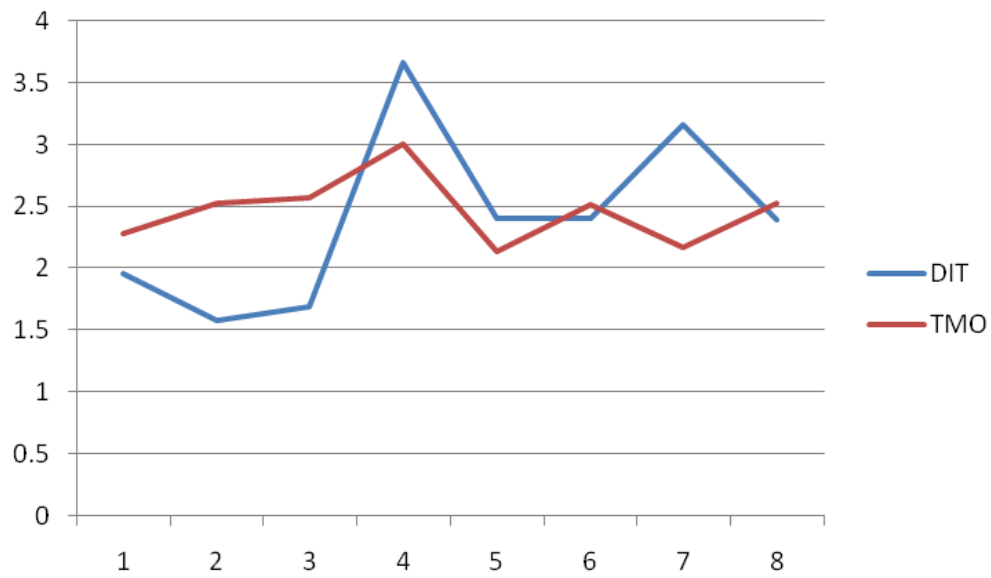


FIGURE 6.3: TMO and DIT.

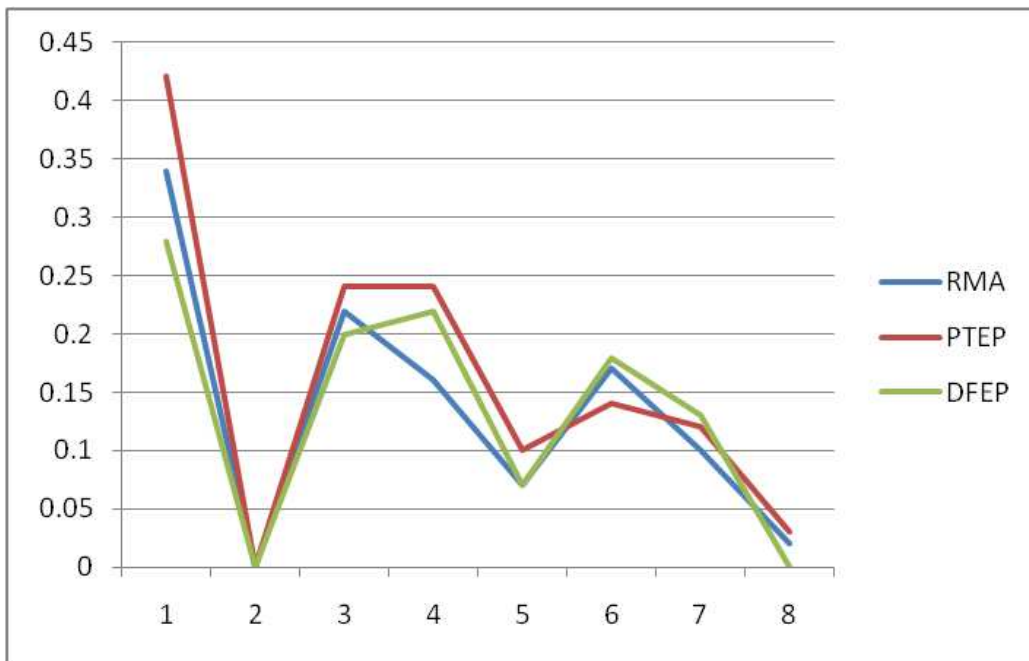


FIGURE 6.4: RMA and PTEP, DFEP.

## Chapter 7: Evaluation Overview

To evaluate the predictions of our approach, we selected systems from different domains than the ones that were previously analyzed. We initially selected four random mobile applications of sizes ranging from 4 to 6 KLOC from a repository of open-source Android applications, F-Droid<sup>1</sup>. We tested the hypotheses on the test systems. The results of applying both the hypotheses H1 and H2 on the test systems is presented in Table 7.1. From the results, the two systems Nectroid and Blokish negate H1 based on the No-Annotation Visitors. The No-Annotation Visitors indicated that the two systems do not extensively use object-oriented concepts e.g., inheritance and do not make much use of the standard library containers. So for those systems, the abstract runtime structure may not be significantly different from the code structure.

TABLE 7.1: Hypotheses tested on four Test Systems.

System Name	Test H1	Test H2
<b>Muspy</b>	<b>True</b>	<b>True</b>
Nectroid	False	True
Blokish	False	True
<b>Ermete SMS</b>	<b>True</b>	<b>True</b>

For the other two test systems, both H1 and H2 predict that the abstract runtime structure may differ significantly from the code structure. The cumulative results of the number of fields and variables declared in each classification for both the systems are shown in Table 7.2. More than 70% of the fields and variables are classified into one of the classifications. The average of the three chosen code metrics for both the systems are shown in Table 7.3. The average of DIT, RMA and CBO matches with the average of code metrics of the training set. Thus, we picked Muspy and Ermete

---

<sup>1</sup><https://f-droid.org/>

TABLE 7.2: Cumulative Classification count on Muspy and Ermete SMS using No-Annotation Visitors.

Classification	Muspy	Ermete SMS
CGT	12	18
CT	24	10
IT1	1	39
IT2	5	2
FGT	49	37
Composition	102	152
Library type	66	90
Others	74	48
Total classified	333	396
Unknown classification	159	167

TABLE 7.3: Average of Code Metrics from Muspy and Ermete SMS.

Metric	Muspy	Ermete SMS	Range of Code Metrics from Model Building
DIT	2.4	2.0	2.00 – 3.00
RMA	0.1	0.1	0.1 – 0.2
CBO	3.5	3.2	3.00 – 4.00

TABLE 7.4: Test Systems Selected for Evaluation.

System Name	KLOC	All types	Description
Muspy	6.2	81	Keeps tracks of musicians and albums
Ermete SMS	4.6	48	Messaging service for T-Mobile users

SMS to evaluate the proposed approach. Some of the measures from the code of the two test systems are presented in Table 7.4.

We analyze the results from No-Annotation Visitors and the computed code metrics for both newly chosen test systems. In order to close the loop and conclude that the predictions from the proxy match with the results from the DiffMetrics, a graduate student (experimenter) annotated the systems and extracted the **OGraphs**. We then computed the DiffMetrics from the **OGraphs**, and analyzed the results from DiffMetrics for each system manually. The manual inspection of the outliers from the DiffMetrics gave us some confidence. So, we computed the p-value (Table 7.5) based on the one-sample Wilcoxon non-parametric test to test if the difference between the abstract runtime structure is statistically significant from the corresponding code

TABLE 7.5: Statistical analysis for the DiffMetrics: p-value ( $p$ ), Cliff's Delta ( $D$ ) and Cliff's Delta size ( $D$ -Size).

DiffMetric	Muspy			Ermete SMS		
	p	D	D-size	p	D	D-size
WAWB	<b>0.00</b>	<i>0.44</i>	0.33	1.00		0.00
TMO	<b>0.00</b>	-0.75	-0.72	<b>0.00</b>	0.30	0.47
PTEP	0.09		0.08	<b>0.00</b>	<i>0.54</i>	0.33
DFEP	<b>0.00</b>	0.10	0.06	<b>0.00</b>	0.23	0.11
1MInE_RecType	0.50			<b>0.00</b>	<i>0.60</i>	1.03
1MInE_ArgType	<b>0.00</b>			<b>0.00</b>	0.22	0.22
1MInE_RetType	<b>0.01</b>	0.25	0.25	<b>0.00</b>	<i>0.75</i>	0.75
1FRnE				<b>0.00</b>	NA	
1FWnE				<b>0.00</b>	NA	

structure. We also estimated the magnitude of the difference using Cliff's Delta  $D$ , a non-parametric effect size for ordinal data. For each of the DiffMetrics, we defined a control value.  $D$  ranges between +1 to -1, if all values from the DiffMetrics are higher than the control value, then  $D$  is positive else it is negative. For one of the systems, the  $D$  values are negative. 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  $D \geq 0.474$ .

## Chapter 8: Evaluation

We present the results of evaluation of the proposed proxy on two test systems here.

### 8.1 Muspy

Muspy is an open source, free Android application that notifies music lovers when there are new releases of albums of the various artists they follow. The system constantly checks the official websites of the artists and in turn eases out the trouble of constantly monitoring the web sites. An user can create a new account linking his email address. Once the first time registration process is complete, the user can use the registered email to log back in and follow or un follow their favorite artists , with just a click of button in the system. The users may also share the released albums on social networking sites e.g., Facebook, Tweeter. The system also sends the users automated email notifications about recent releases of albums. We run the proxy that the proxy consists of the two tests, H1 and H2. We discuss the results from the proxy here.

#### 8.1.1 Testing the Hypotheses

We predict if the abstract runtime structure differs significantly from the code structure based on the results from the proxy.

**Testing H1.** About 40 fields or variables are classified as container of a general type or container of a type. We investigate the code and understand that the developers use standard library containers such as `ArrayList` and `LinkedList`. The elements of containers are of types `Art` or `Releases` of the artists. The classifiers identified only about eight fields or variables as IT1 and IT2. Most of the fields or variables that are associated with inheritance are not system types but are of types from Android

TABLE 8.1: Classification using No-Annotation Visitors.

Type	Instance_Name	Enclosing_Type	Classification
List<model.Release>	releases	utils.RelHolder	CT
List<model.Release>	releases	act.RelAct	CT
List<model.Art>	releases	act.RelAct	CT
List<model.Art>	releases	act.SearchArtAct	CT
List<model.Art>	releases	services.MusicClient	CT
Array<model.Art>	adapter	act.RelAct	CT
LoadBio.Listener	listener	ArtBioAct.LoadBio	FKT
base.AListAct.Listener	listener	base.AListAct	FKT
android.SharedPreferences	sharedPreferences	muspy.MuspyApp	FGT
android.SharedPreferences	sharedPreferences	act.SettingsAct	FGT
android.SharedPreferences	sharedPreferences	base.AListAct	FGT
android.Context	context	template.CustomListAdapter	IT1
List<model.Art>	CREATOR	model.Art	IT1
android.OnItemClickListener	mOnClickListener	base.AListAct	IT2
android.Intent	intent	act.AboutAct	Java Library type
android.Intent	intent	act.RelAct	Java Library type
android.Builder	builder	act.RelAct	Java Library type

library. Investigation of the code indicate that the system only has four general types, e.g., `AAct`, `ABrowserAct`, `AListAct` and `ICustomListAdapterHolderPattern`. Moreover, the abstract type `ABrowserAct` is not used in any system specific tasks. So, the system does use object-oriented concepts like abstractness and inheritance in the implementation but not to a large extent. About 10% of the total classified fields or variables are Field/Variable of a general type. So, fields or variables of 4 general types may be referred to in many concrete subtypes. However, fields or variables that are library types are large in number. More than about 30% of fields or variables are classified as Composition. The category Others in Table 7.2 include fields or variables of final and exception types. Also, the classifiers classified about 70 fields or variables as Java Library Type. Such fields or variables may not reflect the significant difference during the abstract runtime structure as they are usually mapped to a *shared* domain. We present a portion of the output from the No-Annotation Visitors (Table 8.1).

As the classifiers classify many fields or variables into the classifications of container of a general type or container of a type, WAWB may have outliers. The

classifiers classify many fields or variables as field/variable of a general type and so the DiffMetrics from the Precision Category have many outliers. The classifiers identify very few fields and variables that are associated with inheritance. So, all the DiffMetrics grouped into the DFMetrics Subcategory have only few outliers.

**Testing H2.** The values from the computed code metrics are in the same range as the training set systems. So, the code metrics indicate that the corresponding DiffMetrics that correlate with the code metrics may have outliers.

**Predictions from Proxy.** Based on the results from the proxy that tests both the hypotheses on this system, we draw the following predictions.

1. WAWB indicates that the abstract runtime structure is significantly different from the corresponding code structure.
2. DiffMetrics in the Precision Category have many outliers indicating that the abstract runtime structure is significantly different from the code structure.
3. DiffMetrics in the DFMetrics Subcategory do not contribute to the significant difference.
4. DiffMetrics grouped in Many-To-One Category do not have many outliers.

**Closing the Loop.** To support the predictions made by the proxy, we compute the DiffMetrics from the extracted OGraph. To start with, the experimenter annotated the system and extracted the OGraph. A brief about the annotation process is discussed in the subsequent section. The results of the DiffMetrics are also discussed in detail.

### 8.1.2 Annotations

Using the documentation and inspecting the code, the experimenter decide that Muspy followed a three tier architecture. The three tiers are the three top level domains UI, LOGIC and DATA in the OGraph. The types of objects that are strictly



```

class MuspyApp<U, L, D> extends App<U, L, D> {
  Crypto<D<D>> SC = new Crypto();
  String<shared> email = null;
  String<D> pass = null;
  String<shared> userID = null;
  public void setCredentials(String<shared> email, String<D> pass, String<shared> userID) {
    Log<L> log = new Log();
  }
}

class MyArtAct<U, L, D> extends AListAct<U, L, D> {
  List<shared, <Art><shared<D>>> artists;
  ArtHandler<OWNED<U, L, D>> artistsHandler = new ArtHandler(this);
}

class SearchArtAct<U, L, D> extends AListAct<U, L, D> {
  List<shared, <Art><shared<D>>> artists;
  SearchHandler<OWNED<U, L, D>> searchHandler = new SearchHandler(this);
}

class SignInAct<U, L, D> extends AListAct<U, L, D> {
  SignHandler<OWNED<U, L, D>> signHandler = new SignHandler(this);
}

class SignUpAct<U, L, D> extends AListAct<U, L, D> {
  SignHandler<OWNED<U, L, D>> signHandler = new SignHandler(this);
}

```

FIGURE 8.1: Annotation of Muspy.

encapsulated are mapped into *owned* domains and fields and variables of type `String` are mapped into the *shared* domain. During the annotation process, the experimenter refactored some code e.g., add missing constructors. Below, we show a sample of the annotated code from Muspy (Figure 8.1).

### 8.1.3 Results from the DiffMetrics

WAWB has outliers of type `List` of `Art` as indicated by the No-Annotation visitors. The `OObjects` of types `Art` that are containers of a type are mapped into different domains of `OObjects` of different activity types such as `SignInAct`, `SignOutAct` etc. The lists of type `Art` play different roles in each of the different activity types. Since six activities that are subtypes of `base.AListAct` in the system observe the mouse click event, the type `base.AListAct.ClkListener` are mapped into *owned* domains of the types. The `OObjects` of containers of `Array` of `Art` are used in different context and play different roles in the runtime. So they are mapped into different domains of

TABLE 8.2: WABW Outliers from Muspy.

Outlier	Size	Type	Triplets
X	1	List<model.Art>	<List<model.Art>, unique1, act.RelAct>
			<List<model.Art>, unique6, act.RelActSearch>
X	36	base.AListAct.Listener	<base.AListAct.Listener, owned, act.RelActSearch>
			<base.AListAct.Listener, owned, act.RelAct>
			<base.AListAct.Listener, owned, act.SearchArtAct>
			<base.AListAct.Listener, owned, act.SignInAct>
			<base.AListAct.Listener, owned, act.RelAct>
			<base.AListAct.Listener, owned, act.SignInAct>
			<base.AListAct.Listener, owned, MyArtAct>
X	1	Array<model.Art>	<Array<model.Art>, unique1, act.RelAct>
			<Array<model.Art>, unique6, act.RelActSearch>

```

+- Object
+-Act
+-AAct
+-ABrowserAct
+-ALAct
+-SearchArtAct
+-MyArtAct
+-ImtLastfmAct
+-RelAct
+-RelActSearch
+-SignInAct
+-SignUpAct
+-SettingsAct
+-ResetPwdAct

```

FIGURE 8.2: Inheritance hierarchy of AAct.

different parent `OObjects`. Also, the containers of `Releases` are in the enclosing types `ReleasesHolder`, `RelAct` and are used in different contexts. Some of the outliers are discussed here are illustrated in Table 8.2.

As suggested by the proposed proxy, PTEP and DFEP have many outliers. The type of the receiver `AAct` of the method `transition(intent)` has 11 subtypes. A typical type hierarchy shows all the subtypes of the type `AAct` (Figure 8.2). But the `OGraph` only shows 8 of those subtypes that in the reachable domains after formal domain parameters are bound to the actual domains. Some of the outliers are discussed in Table 8.3.

The total number of outliers of each `DiffMetrics` under different categories (Ta-

TABLE 8.3: DEEP Outliers from Muspy.

Outlier	Expression	Type	DFTYPE	All_Subtypes	Subset_Subtypes	Precision
X	transition(intent)	base.AAct	InvkRec	11	8	03

TABLE 8.4: Count of outliers of DiffMetrics.

Category	DiffMetric	Number of Outliers
One-To-Many Category	WAWB	8
DFMetrics	1MInE_RecType	0
	1MInE_RetType	4
	1MInE_ArgType	0
	1FRnE	0
	1FWnE	1
Many-To-One Category	TMO	15
Precision Category	PTEP	12
	DFEP	22

ble 8.4) indicate that the DiffMetrics under the category One-To-Many and Precision and Many-To-One have outliers and that the DiffMetrics under the Subcategory DF-Metrics do not have outliers.

#### 8.1.4 Results from the Wilcoxon test

The results are statistically significant for WAWB and D indicates medium effect. As indicated by the proxy, the metric involves OObjects of type containers. Some OObjects are types of inner classes e.g., MyFactory, NewClickListener that are mapped into different domains play different roles in the system. The results indicate that the DiffMetrics grouped under the Subcategory DFMetrics are not statistically significant. Most of the method invocation expressions are associated only with one or few OEdges in the OGraph. This is the same with field read and field write expressions. The proxy indicates that there are no or few outliers for TMO but the results are statistically significant. Inspection on the data from the DiffMetrics indicate that OObjects of Android libraries types merge OObjects of other library types in the OGraph. As we do not prominently consider fields or variables of library types during the model building phase, the predictions may not reflect for OObjects of library types. D is negative as there are objects that merge only one other type

and the value is less than the control value specified for this metric.

To conclude the evaluation on this test system, the prediction of the proxy matches the the results of the analysis of the outliers of the DiffMetrics and the results from Wilcoxon test for all the DiffMetrics except the DiffMetrics from the Many-To-One Category. Next, we discuss the second test system.

## 8.2 Ermete SMS

Ermete SMS is another open source, free Android application that lets the users to send uninterrupted SMS through the internet. The system lets the users of TIM (Telecom Italia Mobile) to exchange messages between others users and provides options for group chat conversation. The users can create or modify their account via interfaces provided by the system. We run the proxy that tests H1 and H2. We discuss the results from testing the hypotheses here.

### 8.2.1 Testing the Hypotheses

We predict if the abstract runtime structure differs significantly from the code structure based on the results from the proxy.

**Testing H1.** Only about 6% of the total classified fields or variables are classified as container of a general type or container of a type. Inspection of the code indicate that the implementation use Java containers that store elements that have types from the Android library. These types do not lead to the significant difference between the abstract runtime structure and the code structure. Also, further investigation indicated that the ermete developers did not use Java containers extensively. The classifiers indicate that about 40 fields or variables may be associated with Inheritance. The OObjects of such fields or variables may be outliers of the DiffMetrics. It is interesting to note that the system does not implement any system-specific interfaces

TABLE 8.5: Classification using No-Annotation Visitors.

Type	Instance_Name	Enclosing_Type	Classification
List<acc.Acc>	providers	acc.AccManagerAndroid	CGT
List<http.NameValuePair>	reqData	provider.TIM	CGT
List<http.NameValuePair>	reqData	provider.TIM	CGT
List<message.Receiver>	receivers	message.SMS	CT
acc.Acc	ac	android.AccountService	IT1
acc.Acc	ac	android.AccModifyAct	IT1
acc.Acc	oldAcc	android.AccModifyAct	IT1
acc.Acc	newAcc	android.ComposeAct	IT1
acc.AccManager	accManager	android.AccModifyAct	IT1
acc.AccManager	accManager	android.ComposeAct	IT1
acc.AccManager	accManager	android.AccOverviewAct	IT1
acc.AccManager	accManager	android.AccDisplayAct	IT1
msg.ConvMan	convMan	android.AccService	IT1
msg.ConvMan	convMan	android.ComposeAct	IT1
http.NameValuePair	NVP	provider.Tim	FGT
con.ServiceConn	serviceConn	android.ComposeAct	FGT

but has about 10 abstract types. Also, only a few concrete types extend the abstract types. About 40 fields or variables of types fall under field/variable of a general type. However, most of the fields or variables under this classification are not system specific. More than about 30% of fields or variables are classified as Composition. The category Others in Table 7.2 include fields or variables of final and data types. Such fields or variables may not reflect significant difference during the runtime as they are usually mapped into *shared* domains. We present a portion of the output from No-Annotation Visitors in Table 8.5.

As the classifiers identified few fields or variables into the classifications container of a general type or container of a type, WAWB may not have many outliers. The classifiers classify many fields or variables into Inheritance Type 1 or Inheritance Type 2. So the DiffMetrics under the Subcategory DFMetrics have outliers. The classifiers classified many fields or variables as field/variable in general type. However, as many fields or variables are Android library types, the DiffMetrics in Precision Category may have many outliers that show precise information only for these types. The proxy predicts that the abstract runtime structure may be significantly different from

the code structure as indicated by the DiffMetrics under the DFMetrics Subcategory. Thus, we test the second hypothesis.

**Testing H2.** The values from the computed code metrics are in the same range as the training set systems. So, the code metrics indicate that the corresponding DiffMetrics that correlate with the code metrics may have outliers.

**Predictions from Proxy.** Based on the results from the proxy that tests both the hypotheses on this system, we draw the following predictions.

1. DiffMetrics under DFMetrics Subcategory, indicate that the abstract runtime structure is significantly different from the corresponding code structure.
2. DiffMetrics grouped in Many-To-One Category have many outliers.
3. DiffMetrics under Precision Category have many outliers indicating that the abstract runtime structure is significantly different from the code structure.
4. WAWB indicate that same types in the system are not used in different context playing different roles.

**Closing the Loop.** The proposed proxy concluded that the hypotheses are valid and indicate that the abstract runtime structure is significantly different from the code structure. To close the loop, the experimenter, annotate the system and extract the OGraph.

### 8.2.2 Annotations

Using the documentation and inspecting the code, the experimenter decide that Ermete SMS also followed a three tier architecture. The three tiers are the three top level domains UI, LOGIC and DATA in the OGraph. The domain *owned* that encapsulate OObjects are not used so extensively. The types that are of not much interest, like

```

class Main<U, L, D> {
}
class AccCreatAct<U, L, D> extends Act<U, L, D> {
    AccManager<L<U, L, D>> accManager;
}
class AccModifyAct<U, L, D> extends Act<U, L, D> {
    AccManager<L<U, L, D>> accManager;
}
class AccDisplayAct<U, L, D> extends Act<U, L, D> {
    AccManager<L<U, L, D>> accManager;
}
class ComposeAct<U, L, D> extends Act<U, L, D> {
    AccManager<L<U, L, D>> accManager;
}
class SettingsAct<U, L, D> extends Act<U, L, D> {
    AccManager<L<U, L, D>> accManager;
}

```

FIGURE 8.3: Annotation of Ermete SMS.

`String` that does not carry confidential information is mapped into the domain *shared*.

Below we show a sample of the annotated code from Ermete SMS (Figure 8.3).

Using the above annotated code, we extract `OGraph` and computed `DiffMetrics`.

### 8.2.3 Results from the DiffMetrics

The containers of `Acc` declared in the type `AccModifyAct` all play the same role. Thus, the experimenter mapped such `OObjects` into the same domains. Also, the containers of Android library types e.g., `http.NameValuePair` are also mapped into the same domains under the same parent `OObject` of type `provider.Tim`. So WAWB does not have any outliers.

As suggested by the proposed proxy, the `DiffMetrics` from `DFMetrics Subcategory` have many outliers. Various method invocations, e.g., `accManager.delete(acc)` whose receivers and actual arguments are general types of type `AccManager` and `Acc`, are associated with many `OEdges` in the `OGraph`. Methods that are declared in the type `Act` have at least one `OEdge` in the `OGraph` that corresponds to five different activities in the system. Some of the outliers are illustrated below in Table 8.6.

Types in the code that are subtype compatible with `Act` are merged into one

TABLE 8.6: 1MInE\_ArgType Outlier.

Outlier	Expression	Edges: Osrc→Odst [Type]
X	accManager.delete(acc)	AMA:AccModifyAct → AMAnd:AccManager [Export t]
		AMA:AccModifyAct → AMAnd:AccManager [Export ac]
X	accManager.insert(acc)	AMA:AccModifyAct → AMAnd:AccManager [Export t]
		AMA:AccModifyAct → AMAnd:AccManager [Export ac]

TABLE 8.7: Count of outliers of DiffMetrics.

Category	DiffMetric	Number of Outliers
One-To-Many Category	WAWB	0
DFMetrics	1MInE_RecType	5
	1MInE_RetType	5
	1MInE_ArgType	8
	1FRnE	0
	1FWnE	0
Many-To-One Category	TMO	12
Precision Category	PTEP	14
	DFEP	91

OObject of types such as `ComposeAct`, `AccOverViewAct` etc. in the `OGraph`. So, TMO also has outliers. Thus, the conclusions of testing the hypotheses match with the DiffMetrics results.

The total number of outliers of each DiffMetrics under different categories (Table 8.7) indicate that the DiffMetrics under the Subcategory DFMetrics, the categories Precision and Many-To-One have outliers and that the DiffMetrics under One-To-Many category do not have outliers.

#### 8.2.4 Results from the Wilcoxon test

The proxy predicts that WAWB does not have outliers, and the results are not statistically significant for WAWB. The results indicate that the DiffMetrics grouped under the Subcategory DFMetrics are statistically significant. The D value indicates large effect for 1MInE\_RecType and 1MInE\_RetType. From the data, the receivers of method invocations is of type `Act` and has about 12 subtypes. Most of the method



invocations are associated with at least five `OEdges` in the `OGraph` and the destination `OObjects` are subtypes of `Act` in reachable domains after formal domain parameters are bound to the actual domains.

To conclude the evaluation on this test system, the prediction of the proxy matches with the results of the outliers of the `DiffMetrics` and with the results from Wilcoxon test for all the `DiffMetrics`.

## Chapter 9: Related Work

Various research areas that are related to the line of work in this thesis are discussed here. We organize the discussion around research topics that use (1) metrics as predictors in Section 9.1, (2) correlation as predictors, and (3) code patterns as predictors in Section 9.2.

### 9.1 Metrics as Predictors

We discuss research topics that use metrics as predictors for maintainability, detection of defects, fault proneness, program comprehension, and runtime properties.

#### 9.1.1 Predictors of Maintainability and Defects

Taba et al. explore the use of antipatterns for bug prediction in order to improve the accuracy of previously existing bug prediction models [21]. Antipatterns are introduced into the systems by the developers' lack of domain experience or lack of ability to solve a particular problem. Another research also claim that the classes with antipatterns are more prone to bugs than other classes [13]. Taba et al. propose metrics that quantitatively measure antipatterns. Some of the antipatterns are Blob: too large and not cohesive enough; LazyClass: a class that has grown too large with very few fields or methods; MessageChain: a class that uses a long chain of method invocations to realize one of its small functionalities. The antipatterns indicate the data flow and the structure of the system. They define metrics such as Average Number of Antipatterns (ANA), Antipattern Cumulative Pairwise Differences (ACPD) etc to measure the properties associated with the identified antipatterns. There may be a class with large number of methods but many of its methods may not be invoked. The ANA metric counts such a class as a class that demonstrates the antipattern, MessageChain. The DiffMetrics grouped into DFMetrics Subcategory may help us to

determine if such a class that is not coupled with the other classes at runtime does not contribute to bugs.

Dagpinar and Jahnke [11] investigate if the object-oriented metrics from Chidamber and Kemerers [10], Bieman and Kangs [7] can be used as significant predictor for the maintainability of a code base. They evaluate import coupling metrics vs. export coupling metrics, direct coupling metrics vs. indirect coupling. They then conduct an experiment. For each subject system, they collected maintenance activities with intervals of few months over few years. They propose a regression model that correlates the most suitable metrics, based on their previous evaluation, with the frequency of perfective/adaptive maintenance activities. The DiffMetrics grouped under the Subcategory DFMetrics are alternative metrics that measure coupling between objects at runtime. These DiffMetrics indicate the classes that are important in the system, classes whose methods are invoked by other classes. Measuring such properties from the DiffMetrics may bring some useful connection for predicting maintenance and bug prediction models.

### 9.1.2 Predictors of Program Comprehension

Mathias et al. discuss software measurements and metrics that are factors when conducting comprehension studies [17]. They propose various attributes in a system e.g., lines of code and derive measures that are quantitative of the defined attributes. The combination of such measures are metrics categorized into size, object-oriented and structural measures. Such metrics do not predict program comprehension, but are factors that impact program comprehension. For example, for two systems of similar size, a system with fewer data flow paths is easier to understand. A class that is associated with fewer method invocations may be easier to understand than a class with many method invocations. The DiffMetrics grouped under the Subcategory DFMetrics measure dataflow communication in the OGraph, which is an approximate

runtime structure. Also, the identified code patterns effect the DiffMetrics that indicate that the abstract runtime structure is different from the code structure. Our work on the transcript analysis discussed in Appendix A: Transcript Analysis indicate that the outliers of the DiffMetrics trace to the portions of the code in the system that are difficult to comprehend, or are most frequently explored. Thus, the defined DiffMetrics in this work can be used to predict portions of code in that system that may be difficult to understand by the developers.

Yu-ying et al. use runtime information to discover knowledge about software systems [22]. They claim polymorphism, dynamic binding and inheritance rendering cannot be captured using static metrics e.g., Fan-in and Fan-out. They re-define these static metrics that measure properties like inheritance, dynamic binding in method or class to metrics that measure such properties in object level coupling for a scenario 'S', a sequence of user inputs triggering actions in the system that yields an observable results. Such dynamic metrics effectively identifies important classes and methods. Their approach uses dynamic analysis to define metrics and focuses on a particular sequence of event. The DiffMetrics `1MInE_RecType`, `1MInE_ArgType`, `1MInE_RetType` measure object level coupling and interaction using static analysis that covers all possible scenarios in the system.

### 9.1.3 Predictors of Runtime State or Properties

Virtual calls may cause significant performance overhead due to dynamic binding. Zhang et al. proposed techniques that obtain information about the execution frequencies of the targets for unresolved virtual calls at compile time [23]. They explore the frequency distribution or relative frequencies of virtual call targets by defining static program-based metrics derived from features in the system that cause imbalance of usages of different virtual call targets. They build a model to predict certain defined metrics cause imbalance of usages of virtual call targets than other metrics.

The model is two phased: Model Building and Estimation. They analyze the proposed metrics and dynamic profiles for training sets systems to model. Based on their model, they use the metrics computed on other systems to identify the causes of imbalance of usages of the virtual call targets. Our work follows the style in this work closely. We build a model to predict if the abstract runtime structure of a system is significantly different from its code structure. The model identifies code patterns and other metrics that affect the abstract runtime structure by correlating them with the DiffMetrics computed using a statics analysis on the training set systems. The metric, number of callers (No. Callers) that uses the calling relations to measure the popularity of the method may linearly or non-linearly correlate with the DiffMetrics, 1MinE\_RecType, 1MinE\_ArgType and 1MinE\_RetType.

In a system execution, some paths may be executed frequently and some paths not so frequent. This high degree of non-uniformity in a system execution makes characterizing runtime behavior of system an important concern for code optimization and general data flow analysis [5]. Buse and Weimer propose a statistical model of path frequencies based on metrics that can be obtained from the source code of a system. Such metrics predict runtime path frequencies [9]. The infrequent paths involve system instructions associated with error detection, reorganizing data structures, resizing hash tables etc. They claim that the paths that exhibit only small impacts on program state, both in terms of global variables and in terms of context and stack frames are most likely the hot-paths that are executed frequently in runtime. The metrics capture the state changing behavior and are based on the data flow structure of the system. The DiffMetrics grouped under the Subcategory DFMetrics measure dataflow between two objects in runtime due to a method invocation, field read or a field write. Such DiffMetrics can also be used to investigate hot-paths in the system. As the DiffMetrics are derived from the abstract runtime structure, path frequency estimates may be more closely associated with runtime compared with their

TABLE 9.1: Correspondence between DiffMetrics and Metrics from [9].

<b>DiffMetrics</b>	<b>Metrics-Predict Path Freq</b>
1MInE_RetType	return-stmts
1MInE_RecType	invoked method
1MInE_ArgType	parameters
1FWnE	fields written

metrics derived from the source code. The correspondence between the DiffMetrics and their metrics is presented in Table 9.1.

The metrics from both the above research are derived from the call graph. The DiffMetrics are extracted from a hierarchical, sound abstract runtime structure that considers all possible execution path.

## 9.2 Correlations and Code patterns as predictors

The research uses correlations and code patterns as predictors for efforts of testability of systems and bug predictions. Bruntink and Deursen evaluate a set of Chidamber and Kemerer’s metrics [10] with respect to their capabilities to be able to predict the efforts needed for testing [8]. They claim that features like inheritance, polymorphism and other factors e.g., cohesion of methods, coupling affect test cases generation. They define two source-code level metrics Lines of Code for class (dLOCC) and Number of test cases (dNOTC) for their test suite. They correlate standard metrics that measure the factors that affect testability, Lack of Cohesion of Methods (LCOM), Fan out (FOUT) with the derived metrics for their test suite using Spearman’s correlation. They predict the factors that influence test case generations based on the p-value. Their study indicates of how specific object-oriented features affect the test case generation. We compute some Chidamber and Kemerer’s metrics [10] that measure coupling, cohesion. Then we compute correlation between them with the DiffMetrics using Pearson’s correlation. The p-values measure the significance of the DiffMetrics and indicates that the abstract runtime structure is different from the

corresponding code structure. We also identify some code patterns e.g., containers, fields of general types that affect the abstract runtime structure of a system.

Rilling and Klemola [20] propose metrics to predict the location of high frequencies of defects. The developer may not be familiar with system specific identifiers defined within the system. A larger number of variables, classes, methods and other developer-defined labels or entities, leads to greater difficulty system may be difficult to understand as each of the programmer-defined labels or entities must be traced to identify their definition in the system. Code that has been fragmented into many small parts will have a higher concentration of method invocations with parameters. This would need many identifiers and thus, more tracing activity during the comprehension process. Based on the above observation, they propose a metric Identifier Density (ID) that identifies the density of developer-defined entities on inspection of the code. The developers may be interested in focusing measures on a particular execution path. This will provide developers some guidance in identifying the difficulty level of comprehending a particular program execution. So, they propose another dynamic metric Dynamic Identifier Density (DID). They claim that the rise in ID and DID is caused by external coupling, complex expressions and code patterns like arrays, Java collections, complex conditional statements. We identify code patterns like containers, composition, inheritance. The transcript analysis discussed in Appendix A: Transcript Analysis indicate that the outliers of the DiffMetrics trace to the portions of the code in the system that are difficult to comprehend.

## Chapter 10: Discussion and Conclusion

We defined a proxy for a heavyweight technique that requires adding annotations to the code to extract abstract runtime structure. The defined proxy requires less manual intervention to run on any system compared to adding annotations to the systems, thus making it *lightweight*.

We discuss some threats to validity in Section 10.1 and some limitations in our design of the proxy in Section 10.2. We discuss of how other research groups may be able to adapt the principles behind defining the proxy for selecting systems that may be used to evaluate their programming language technique in Section 10.3. Finally, we talk about some future work in Section 10.4 and conclude in Section 10.5.

### 10.1 Threats to Validity

The approach we use to define the proxy has several threats to validity.

**Non-representative.** The systems in our training set may not be representative. The corpus does not include systems from well-established benchmarks such as the DaCapo benchmark and a few of the systems are closed-source or proprietary. We are aware of this limitation, and started analyzing one of the systems in DaCapo, sunflow.

**Small- and medium-sized systems.** The systems we analyzed, where the largest one is around 35 KLOC, are smaller than those in studies of the code structure because our technique requires adding manual annotations. Without automated inference of these annotations, analyzing large systems is currently infeasible. There is promising, active work in the area of automated inference, which will enable conducting larger scale studies.

**Small training set.** The training set is relatively small and includes only 8 systems totaling 100 KLOC. This number of subject systems is lower than is typically seen



in empirical studies of the code structure or studies of runtime heaps using dynamic analysis [16]. Those studies consist of running a fully automated analysis on a large number of systems. In our case, we had to manually add annotations to each subject system before we could incorporate it in our training set.

## 10.2 Limitations

Our design of the proxy had some limitations that could have interfered with our results.

**False positives.** The visitors may produce a large number of false positives. But since the visitors look for structural patterns compared to visitors that look for local information such as method names, the results are likely to be instructive.

**Manual interpretation of the results.** While the visitors and the code metrics are predictive, the approach still requires a human to interpret the output of the visitors, or the values of the metrics to decide what is considered to be within an interesting range.

## 10.3 Global Discussion

We discuss some lessons learned about the proposed approach based on our case study.

**Adaptability of Lightweight Proxy.** Another research group may use the proposed principled data-driven approach to derive a proxy that selects systems to evaluate their programming-language based heavyweight techniques. Firstly, they need to define DiffMetrics for their heavyweight technique. Then identify other simple metrics or code patterns that may be associated with the defined DiffMetrics. Such metrics and code patterns have to be determined by inspection of the code without the manual intervention the heavyweight technique may need. The proxy may be

visitors or other available open source tools that identify the determined metrics and code patterns.

Once DiffMetrics are defined for the heavyweight technique, a model to predict characteristics that impact the heavyweight technique based on the simple metrics or code patterns determined by the proxy must be built. Using the model and the results from the proxy, they predict if running the heavyweight technique on the system may be beneficial. When considering to analyze a system, they may choose a system on which running the heavyweight technique may not be rewarding. However, the time consumed to run the proxy is less compared to running their heavyweight technique that may require radically rewriting the code in different language or adding annotations to the code. They will be able to run the proxy on many systems with ease and identify collection of systems that may be worth studying further.

**Designing the Code Pattern Visitors.** The visitors that visit the OGraph contribute to a better understanding of the relationships between the abstract runtime structure and the code structure by identifying code patterns that may lead to significant differences between the two structures of a system. The code patterns also identify portions of the code that may be difficult to comprehend by the developers. The No-Annotation visitors identify the same code patterns that may lead to significant differences between the two structures of a system with no added annotations. Also, the classifications such as Field/Variable of Framework type need additional inputs such as the list of framework types in a system. We may be able to reduce the count of false positives identified by the classifiers by using a more precise approach for identifying the framework types.

**Identifying the Correlated Metrics.** The simple metrics that are computed using open source tools contribute to a better understanding of the relationships between code metrics that measure properties of the code and the DiffMetrics. We determine metrics that positively and linearly correlate with the DiffMetrics. In general, the

simple metrics determined by the proxy may also be negatively and non-linearly correlate with properties of a heavyweight technique.

## 10.4 Future Work

We will use the proxy to select systems for which it would be worthwhile to extract the abstract runtime structure, which may be helpful to developers during program comprehension or code modifications. We will also use the proxy to select future systems to use in controlled experiments that we may conduct. The style of the data-driven approach to derive proxy can also be followed for other programming language based techniques to predict for what systems they will be most useful.

## 10.5 Conclusion

We propose a lightweight proxy that predicts types of systems on which a heavyweight technique may be beneficial. We define a proxy for a heavyweight techniques that needs adding annotations to the code. The proxy for the heavyweight technique predicts for what systems the abstract runtime structure, a representation extracted using the heavyweight technique, may be significantly different from the code structure.

## Appendix A: Transcript Analysis

### Introduction: Transcript Analysis

In one of the previous studies conducted [4], we had 10 participants implement three coding tasks. We divided the participants into two groups. Participants from the Control group (numbered C1 ... C5) had access to just class diagram (code structure) and ones from the Experimental group (numbered E1 ... E5) had access to the OGraph (abstract runtime structure). The experiment was conducted on MD. We collected data from both groups while they attempt to implement the tasks. The data is referred to as transcripts. Each transcript consists of types in the systems the participants explored, the architectural diagrams and the functionalities in Eclipse the participants used. The DiffMetrics is computed on MD and the outliers are identified. We associated the code from the system the participants struggled or explored often during the tasks with the outliers from the DiffMetrics. The mining of such data indicate that the outliers point us to code in the system that

- I. are often explored
- II. are interfaces or abstract types in the system
- III. indicate portions of the system's code that are difficult to comprehend

**Research Hypothesis.** The outliers of the metrics trace to the code in the system that are difficult to comprehend during program comprehension or code modification tasks.

### Data from Transcript

We analyze the transcripts of all the participants for each task.

## Task1

The participants implementing the validation of piece movements first need to get the current position of the piece that is to be moved. Then, they need to implement a check if the new position is empty or not. The move of the piece is valid only if the new position is empty or a piece may be captured diagonally. The code from the system that the participants explored for this task typically map to the following DiffMetrics:

- I. *PTEP*. Participants implementing above task wonder what constitute pieces on the board. They analyse irrelevant types from the Package Explorer such as `FigureFactory`. The PTEP shows more precise subtype information for the object `figure`: `Figure` and points only to `BoardDrawing` and `BoardFigure` that are the concrete types representing the pieces of the board.

*QUOTES*. Some of the quotes from the participants indicate the struggle.

- I. *“Okay so BoardFigure gets its information from the FigureFactory interface. Ah! quite a few interfaces but BoardDrawing and BoardFigure are actual classes”* [C5]
- II. *“okay so its says that these guys are part of board drawing, but how?”* [E4]
- III. *“yeah so what I am searching for now is a representation of the board itself the collection. So I am not looking at the right place I need to back up”* [E2]

## Task2

The second task is to implement the capture of pieces on the board. A piece may only capture an opponent diagonally and only an opposite colored piece. We analyze the data from the participants and map the portions of system explored to the outliers of PTEP.

- I. *PTEP*. The participants implementing the task need the color of pieces that are on non-empty squares and color of pieces that is to be moved. The participants find it difficult to locate such a method. They look at types that may be associated with figures. The participants analyze types back and forth and finally look at `BoardFigure` and realize that its the right fit for such a method. The `OGraph` shows more precise subtype information for field of type `Figure`. One of the outliers in PTEP indicate that the concrete types for field of type `Figure` are `BoardDrawing` and `BoardFigure`. Investigating only `BoardDrawing` and `BoardFigure`, the participants could guess that such a method would fit in `BoardFigure`.

*QUOTES*. The following quotes from various participants indicate the struggle.

- I. *“I have to go to this class and check whether there is a member to check color. May be `ImageFigure` okay so this is another class, `FigureFactory` may be, not really” [C3]*
- II. *“So in `BoardFigure` which extends `ImageFigure` which extends god knows what. so `ImageFigure` ah wait a minute, where the heck is the color?” [E4]*

**Task3.** The participants implement the undo feature for all moves except piece capture moves on the board. Many participants fail to complete the task and there are no transcript data available for the task.

## Appendix B: The Overall Outlier Classification table

We visited the `OObjects` and `OEdges` of the defined `DiffMetrics` to look for certain code patterns associated with the outliers. Table 1 is the summary of classification of the outliers of the `DiffMetrics` for all the training set systems.

TABLE 1: Number of Outliers in each Classification for Training set Systems.

metrics	category	MD	CDB	AFS	DL	PX	JHD	HC	APD
WAWB	CGT	2	12	3	1277	441	1763	0	0
	CT	1	0	0	0	0	10	12	2
	FKT	2	3	6	30	0	1374	0	0
	Others	0	0	1	416	0	547	65	0
	Unclassified	0	0	15	249	212	1393	202	1
1MInE_RecType	DFIT1	0	0	0	9	93	8	7	0
	DFIT2	0	0	0	6	0	3	4	0
	DFIT3	0	0	1	13	5	67	12	0
	Unclassified	0	0	6	1	0	0	11	0
1MInE_RetType	DFIT1	0	0	0	7	0	14	8	0
	DFIT2	0	0	0	10	0	30	4	0
	DFIT3	0	0	0	15	45	170	19	0
	DFIT4	7	0	0	1	0	13	56	0
	Unclassified	0	0	0	0	0	0	0	0
1MInE_ArgType	DFIT1	0	0	0	8	1	10	5	0
	DFIT2	0	0	0	6	2	25	3	0
	DFIT3	0	0	1	15	10	63	11	0
	DFIT4	0	0	5	7	7	0	31	0
	DFIT5	0	0	0	0	0	0	0	0
	Unclassified	0	0	0	0	0	0	0	0
1FRnE	DFIT1	0	0	0	0	0	0	0	0
	DFIT2	0	0	0	0	0	0	0	0
	DFIT3	5	0	0	29	19	51	59	0
	Unclassified	0	0	0	0	0	0	39	0
1FWnE	DFIT1	0	0	0	0	0	0	0	0
	DFIT2	0	0	0	0	0	0	0	0
	DFIT3	2	0	0	7	15	17	14	0
	Unclassified	0	0	0	0	0	0	3	0
PETP	FGT	8	0	11	12	35	39	46	1
	Others	0	0	0	0	1	0	0	0
	Unclassified	0	0	0	1	12	1	23	1
DFEP	FGT	56	0	97	149	307	549	271	2
	Others	0	0	0	0	1	0	0	0
	Unclassified	0	3	1	0	24	25	254	0



## REFERENCES

- [1] ABI-ANTOUN, M., AND ALDRICH, J. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)* (2009), pp. 321–340.
- [2] ABI-ANTOUN, M., CHANDRASHEKAR, S., VANCIU, R., AND GIANG, A. Are Object Graphs Extracted Using Abstract Interpretation Significantly Different from the Code? In *IEEE Intl. Working Conference on Source Code Analysis and Manipulation (SCAM)* (2014).
- [3] ALDRICH, J., AND CHAMBERS, C. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object-Oriented Programming (ECOOP)* (2004), pp. 1–25.
- [4] AMMAR, N., AND ABI-ANTOUN, M. Empirical Evaluation of Diagrams of the Run-time Structure for Coding Tasks. In *Working Conference on Reverse Engineering (WCRE)* (2012), pp. 367–376.
- [5] AMMONS, G., AND LARUS, J. R. Improving data-flow analysis with path profiles. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)* (1998), pp. 72–84.
- [6] BECKMAN, N. E., KIM, D., AND ALDRICH, J. An Empirical Study of Object Protocols in the Wild. In *European Conference on Object-Oriented Programming (ECOOP)* (2011), pp. 2–26.
- [7] BIEMAN, J. M., AND KANG, B.-K. Cohesion and reuse in an object-oriented system. In *Proceedings of the ACM SIGSOFT Symposium on Software Reusability* (1995), pp. 259–262.

- [8] BRUNTINK, M., AND VAN DEURSEN, A. Predicting Class Testability using Object-Oriented Metrics. In *IEEE Intl. Working Conference on Source Code Analysis and Manipulation (SCAM)* (2004).
- [9] BUSE, R. P., AND WEIMER, W. The Road Not Taken: Estimating Path Execution Frequency Staticly. In *International Conference on Software Engineering (ICSE)* (2009), pp. 144–154.
- [10] CHIDAMBER, S. R., AND KEMERER, C. F. A Metrics Suite for Object Oriented Design. *Transactions on Software Engineering (TSE)* 20, 6 (1994), 476–493.
- [11] DAGPINAR, M., AND JAHNKE, J. H. Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison. In *Working Conference on Reverse Engineering (WCRE)* (2003), pp. 155–164.
- [12] DE ALWIS, B., MURPHY, G., AND ROBILLARD, M. A Comparative Study of Three Program Exploration Tools. In *IEEE International Conference on Program Comprehension (ICPC)* (2007), pp. 103–112.
- [13] KHOMH, F., PENTA, M. D., GUEHENEUC, Y.-G., , AND ANTONIOL, G. An exploratory study of the impact of antipatterns on class change and fault-proneness. In *Empirical Softw. Engg., vol. 17* (2012), pp. 243–275.
- [14] KITCHENHAM, B., LINKMAN, S., AND LAW, D. DESMET: A method for evaluating software engineering methods and tools. In *ACM SIGSOFT Software Engineering Notes* (1997), vol. 21, pp. 11–14.
- [15] MALAYERI, D., AND ALDRICH, J. Is Structural Subtyping Useful? An Empirical Study. In *European Symposium on Programming Languages and Systems* (2009), pp. 95–111.

- [16] MARRON, M., SANCHEZ, C., SU, Z., AND FÄHNDRICH, M. Abstracting Runtime Heaps for Program Understanding. *Transactions on Software Engineering (TSE)* 39, 6 (2013), 774–786.
- [17] MATHIAS, K. S., II, J. H. C., HENDRIX, T. D., AND BAROWSKI, L. A. The Role of Software Measures and Metrics in Studies of Program Comprehension. In *ACM-SE 37 Proceedings of the 37th annual Southeast regional conference (CD-ROM)* (1999).
- [18] OMAR, C., KURILOVA, D., NISTOR, L., CHUNG, B., POTANIN, A., AND ALDRICH, J. Safely Composable Type-Specific Languages. In *European Conference on Object-Oriented Programming (ECOOP)* (2014), pp. 105–130.
- [19] QUANTE, J. Do Dynamic Object Process Graphs Support Program Understanding? - A Controlled Experiment. In *IEEE International Conference on Program Comprehension (ICPC)* (2008), pp. 73–82.
- [20] RILLING, J., AND KLEMOLA, T. Identifying Comprehension Bottlenecks Using Program Slicing and Cognitive Complexity Metrics. In *IWPC '03 Proceedings of the 11th IEEE International Workshop on Program Comprehension* (2003), pp. 115–124.
- [21] TABA, S. E. S., KHOMH, F., ZOU, Y., HASSAN, A. E., AND NAGAPPAN, M. Predicting Bugs Using Antipatterns. In *ICSM '13 Proceedings of the 2013 IEEE International Conference on Software Maintenance* (2013), pp. 270–279.
- [22] YU-YING, W., QING-SHAN, L., PING, C., AND CHUN-DE, R. Dynamic fan-in and fan-out metrics for program comprehension. *Journal of Shanghai University* 11, 5 (2007), 474–479.

- [23] ZHANG, C., XU, H., ZHANG, S., ZHAO, J., AND CHEN, Y. Frequency Estimation of Virtual Call Targets for Object-Oriented Programs. In *European Conference on Object-Oriented Programming (ECOOP)* (2011), pp. 510–532.

**ABSTRACT****QUANTITATIVE AND QUALITATIVE EVALUATION OF METRICS  
ON OBJECT GRAPHS EXTRACTED BY ABSTRACT  
INTERPRETATION**

by

**SUMUKHI CHANDRASHEKAR****August 2015****Advisor:** Dr. Marwan Abi-Antoun**Major:** Computer Science**Degree:** Master of Science

Evaluating programming-language based techniques is crucial to judge their usefulness in practice but requires a careful selection of systems on which to evaluate the technique. Since it is particularly hard to evaluate a heavyweight technique, such as one that requires adding annotations to the code or rewriting the system in a radically different language, it is common to use a lightweight proxy to predict the technique's usefulness for a system. But the reliability of such a proxy is unclear.

We propose a principled data-driven approach to derive a lightweight proxy for a heavyweight technique that requires adding annotations to the code. The approach involves the following: computing metrics (DiffMetrics) that measure differences between a system representation (e.g., the code structure) and the system representation extracted by the heavyweight technique (e.g., abstraction of the runtime structure); identifying the outliers of the DiffMetrics; identifying code patterns and classifying the outliers based on the identified code patterns; implementing visitors that look for the code patterns on systems with no annotations; identifying code metrics that correlate strongly with the DiffMetrics. For a new system with no annotations, a proxy predicts if the heavyweight technique may be useful based on the results from the visitors and the code metrics.

To evaluate the approach, we run the visitors and compute code metrics on four systems that were previously not analyzed. The proxy predicts that the heavyweight technique may be useful two of the systems. Thus, the abstract runtime structure may be significantly different from the code structure for those systems. To validate the proxy's predictions, we run the heavyweight technique on the two systems to confirm the predictions.

Such a principled approach is reusable and can be applied on any programming-language based technique to identify systems for evaluation and for a better understanding the types of systems for which a technique is most useful.

**AUTOBIOGRAPHICAL STATEMENT**

SUMUKHI CHANDRASHEKAR

**EDUCATION**

- Master of Science (Computer Science), May 2015  
Wayne State University, Detroit, MI, USA
- Master of Science (Computer Science), May 2012  
University of Bonn, Bonn, NRW, Germany
- Bachelor of Engineering (Computer Systems Engineering), June 2007  
Vidya Vikas Engineering and Technology College, Mysore

**PUBLICATIONS**

1. ABI-ANTOUN, M., CHANDRASHEKAR, S., VANCIU, R., AND GIANG, A. Are Object Graphs Extracted Using Abstract Interpretation Significantly Different from the Code? In *IEEE Intl. Working Conference on Source Code Analysis and Manipulation (SCAM)* (2014).
2. ABI-ANTOUN, M., GIANG, A., CHANDRASHEKAR, S., AND KHALAJ, E. The Eclipse Runtime Perspective for Object-Oriented Code Exploration and Program Comprehension. In *Proceedings of the Workshop on Eclipse Technology eXchange (ETX)* (2014), pp. 3–8.