Polymorphism in the Spotlight: Studying Its Prevalence in Java and Smalltalk

Nevena Milojković, Andrea Caracciolo, Mircea Filip Lungu, Oscar Nierstrasz
Software Composition Group
University of Bern, Switzerland
{nevena,caracciolo,lungu,oscar}@iam.unibe.ch
David Röthlisberger
School of Informatics and Telecommunications
Universidad Diego Portales, Chile
davidroe@mail.udp.cl
Romain Robbes
PLEIAD Laboratory
DCC, University of Chile, Chile
rrobbes@dcc.uchile.cl

***Abstract*—Subtype polymorphism is a cornerstone of object-oriented programming. By hiding variability in behavior behind a uniform interface, polymorphism decouples clients from providers and thus enables genericity, modularity and extensibility. At the same time, however, it scatters the implementation of the behavior over multiple classes thus potentially hampering program comprehension.**

The extent to which polymorphism is used in real programs and the impact of polymorphism on program comprehension are not very well understood. We report on a preliminary study of the prevalence of polymorphism in several hundred open source software systems written in Smalltalk, one of the oldest object-oriented programming languages, and in Java, one of the most widespread ones.

Although a large portion of the call sites in these systems are polymorphic, a majority have a small number of potential candidates. Smalltalk uses polymorphism to a much greater extent than Java. We discuss how these findings can be used as input for more detailed studies in program comprehension and for better developer support in the IDE.

***Index Terms*—object-oriented programming, polymorphism, programming languages, programming environments**

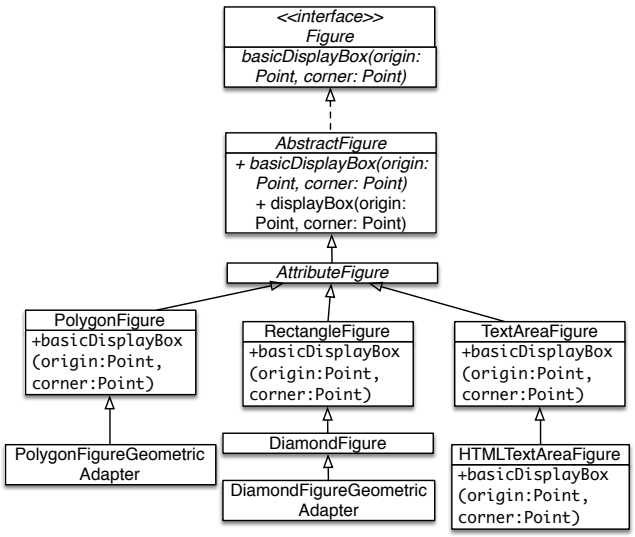

Fig. 1: Sample class hierarchy from JHotDraw, with multiple implementations of the operation basicDisplayBox(Point, Point).

I. Introduction

Polymorphism in programming languages, as opposed to monomorphism, refers to the ability of a variable to be bound to entities of multiple types. In object-oriented languages, subtype polymorphism means that the messages[1] that can be sent to an object are determined by its presumed type (or class), while at run time that object may actually be an instance of some subtype (or subclass).

Subtype polymorphism is fundamental to object-oriented design, allowing developers to write extensible software systems, regardless of whether the target language is statically or dynamically typed.

Since a class can either implement a method itself or inherit it from a superclass, the implementations of a particular message can be scattered throughout the hierarchy. This can impact program comprehension [8], [10].

Consider the classes in Figure 1 from JHotDraw[2], a Java framework for editing structured graphics. The class diagram

[1]In Smalltalk terminology, to invoke a service of an object, one "sends it a message". A message consists of a "selector" (the name of the message) and the arguments. The receiver is then free to decide which "method" to use to respond to that message. This draws an important distinction between "messages" (services offered) and "methods" (implementations of those services)

[2]JHotDraw is a reimplementation by Eric Gamma of HotDraw, originally developed by John Brant in VisualWorks Smalltalk.

shows a subset of the JHotDraw Figure hierarchy[3]. A developer is trying to understand source code of AbstractFigure, one of the key classes of JHotDraw. The following snippet shows an instance of the template method design pattern:

Listing 1: A polymorphic call site

```
public abstract class AbstractFigure
                implements Figure {
  //...
  public void displayBox(Point origin,Point corner){
    willChange();
    basicDisplayBox(origin, corner);
    changed();
  }
  //...
}
```

The receiver at the call site basicDisplayBox(Point, Point) is the implicit variable this, which can be bound to any subtype of AbstractFigure. Since any subtype of AbstractFigure can provide its own implementation of basicDisplayBox(Point, Point), the method that will be actually called is not statically determinable.

The developer could set a breakpoint in the code and observe which of the multiple implementations are invoked at run time, but this usually gives just a narrow selection of all possible invocations. However, unless she sees all possible invocations occurring at that particular place in source code, she will not be able to fully understand the behavior of the basicDisplayBox message. Object-oriented systems highly depend on polymorphism, which impacts software maintenance [18].

We call the number of possible methods that could potentially be called at run time the *cardinality* of a call site. With a slight abuse of terminology, we consider a call site *polymorphic* if its cardinality is greater than 1, a selector polymorphic if it is called at a polymorphic call site, and a method polymorphic if it implements a polymorphic selector (see section II).

The cardinality of the polymorphic call site willChange() equals to just 2, while the cardinality of the call site basicDisplayBox(origin, corner) is 18. A higher cardinality leads to more behavior being scattered through the system, and this will likely impact program understanding [8], [10], [12], [22].

To assess the criticality of polymorphism regarding program comprehension, we investigate in this paper how prevalent its use is in object-oriented software.

Figure 2-a shows the extent of polymorphism in JHotDraw. This visualization, generated by Softwarenaut [14], shows the main package of the system as a treemap, where the area devoted to each class is proportional to its number of lines of code.

The colors indicate presence or absence of polymorphism:

- *red*: classes that define polymorphic methods
- *yellow*: classes that contain polymorphic call sites
- *orange*: classes with both polymorphic methods and polymorphic call sites

Figure 2-a illustrates that polymorphism is almost omnipresent. The same can be observed in HotDraw, its Smalltalk implementation (Figure 2-b). We do not know how polymorphism is being used idiomatically in these projects (for example, whether most usages occur in the context of template methods) nor do we know how much polymorphism impacts comprehension in practice for common development tasks.

[3]The complete hierarchy under AbstractFigure contains 35 classes.

(a) JHotDraw (Java) (b) HotDraw (Smalltalk)

Fig. 2: Polymorphism is endemic in both Java (a) and Smalltalk (b) versions of the main package of the HotDraw framework

Before considering any of these questions, however, we focus in this paper on the more specific questions regarding the prevalence of polymorphism in practice. To our knowledge, there is no large-scale study on the prevalence of polymorphism in open source software. We therefore set out to investigate the actual use of polymorphism in open source development by studying two large corpora of open source software systems, and by posing the following research questions:

RQ1) How prevalent are polymorphic methods in object-oriented systems?
RQ2) How common are polymorphic call sites?
RQ3) What is the distribution of the cardinality of polymorphic call sites?

For the purpose of this study we consider only *static* information, that is, we do not consider how much polymorphism actually occurs at run time. Although this will only give us an upper bound on the actual polymorphism present, we argue that this provides a good estimate of the challenges faced by a programmer reading the source code.

Answering the first question will reveal how much polymorphism is present in object-oriented software. Addressing the second and third questions will disclose how scattered polymorphic code is, and hence help us to assess the potential challenge it poses for program comprehension.

As a result of this study, we obtain a better understanding of the phenomenon of polymorphism and its practical relevance. Results obtained suggest a followup study into idiomatic usage of polymorphism and its impact on program comprehension in practice.

Structure of the Paper. We start by defining our terminology (Section II). Next we introduce our experimental methodology and the analysis infrastructure (Section III). In Section IV we report on the findings regarding the prevalence of polymorphism in practice. In Section V we discuss the implications our results entail and propose a series of questions

to pursue in a followup study. We then describe potential threats to the validity (Section VI). We discuss the related work in the field (Section VII) before concluding in Section VIII.

II. Terminology

In this section we make precise the notion of subtype polymorphism as we measure it on both statically and dynamically-typed languages. To this end, we introduce a simple set-theoretic model in Figure 4 summarized by the UML diagram in Figure 3.

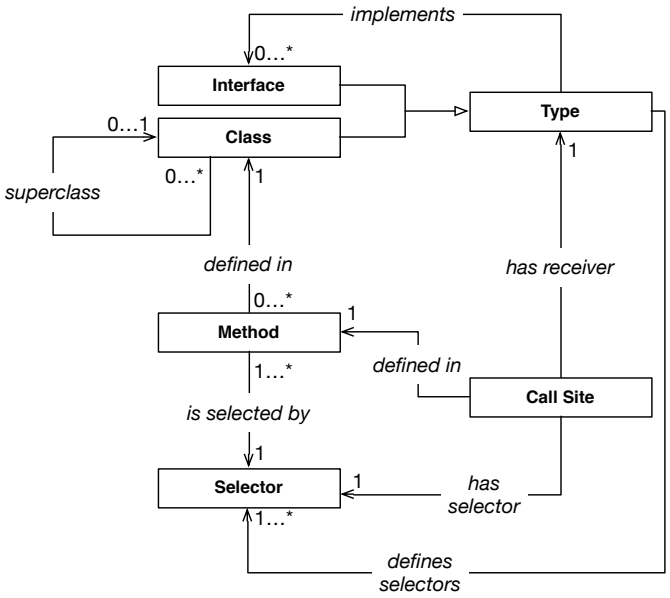

Fig. 3: The core model in UML. The entities Interface and Type are relevant for Java, but not Smalltalk.

A. Core model

Given all source code of a system, C is the set of all classes, I is the set of all interfaces, T the set of all types, M the set of all methods. S is the set of all selectors, *i.e.* method names in Smalltalk (since we do not know the static type of method parameters) and method signatures in Java. *CS* is the set of all method call sites in the system.

$$\begin{aligned} sel &: CS \to S && (1)\\ def_{cs} &: CS \to M && (2)\\ msg &: M \to S && (3)\\ def_{m} &: M \to C && (4)\\ sup &: C \to C && (5)\\ rec &: CS \to T && (6)\\ impl &: T \to \mathcal{P}(I) && (7)\\ sel_t &: T \to \mathcal{P}(S) && (8) \end{aligned}$$

Fig. 4: The core model.

Each method call site cs has a selector $s = sel(cs)$ (1) and is defined in a unique method $m = def_{cs}(cs)$ (2). We say that the method m defines the call site cs and that method m sends the message s. Each method m has a unique selector $s = msg(m)$ (3), and is defined in a unique class $c = def_m(m)$ (4). Class c either has a unique superclass $c' = sup(c)$(5) or doesn't have a superclass.[4] We denote by c^* the"superclass-chain" of the class c (9), and we consider $sup^0(c) = c$. A set of classes $H \subset C$ is a hierarchy if every two classes $c_1, c_2 \in H$ have at least one common class in their "superclass-chains", the intersection of their "superclass-chains" is also contained in H, and for every class $c \in H$ there is no "gap" in its "superclass-chain" within H (10).

$$c^* = sup^*(c) \tag{9}$$

$$(\forall c_1, c_2 \in H)((c_1^* \cap c_2^* \neq \emptyset) \wedge (c_1^* \cap c_2^* \subset H) \wedge (\forall c \in H)(\nexists k, n \in \mathbb{N}_0)(n > k)(sup^n(c) \in H \wedge sup^k(c) \notin H)) \tag{10}$$

In Java, for each call site cs the static type of the receiver is $t = rec(cs)$ (6). The type of the receiver can be either an interface or a class. Each type t has a set of interfaces it implements $i_t = impl(t)$[5] (7), and a set of selectors it defines $s_t = sel_t(t)$ (8). This information is not available for Smalltalk, and is not modeled.

Consider the example in Listing 1. For the call site $cs =$ basicDisplayBox(origin, corner), the selector (a Java signature) is $sel(cs) =$ basicDisplayBox(Point, Point), the receiver type is $rec(cs) =$ AbstractFigure, and the call site is defined in the method $def_{cs}(cs) =$ displayBox(Point, Point).

We can now query the model to compute the metrics necessary to answer our research questions, as summarized in Figure 5.

Definition 1: A call site *cs* is *polymorphic* if there is more than one method that can be invoked at *cs* at run time.

The way we determine this is slightly different in Smalltalk and Java, since we lack static type information in Smalltalk. To find all polymorphic call sites we first introduce the function that maps a type t to its complete subhierarchy (11).

Consider the example in Figure 1. For $t =$ RectangleFigure, subhierarchy is $subh(t) =$ {RectangleFigure, DiamondFigure, DiamondFigureGeomtericAdapter}, while for $t =$ AttributeFigure, $subh(t) =$ {AttributeFigure, PolygonFigure, PolygonFigureGeometricAdapter, RectangleFigure, DiamondFigure, DiamondFigureGeometricAdapter, TextAreaFigure, HTMLTextAreaFigure}. We also introduce the function $impl(t, s)$ which yields the set of methods implementing the selector s and defined in the subhierarchy of the type t (12).

In Smalltalk we do not know the type of the receiver at compile-time, so we must investigate each polymorphic call site based only on its selector. To do this, we use the function $undr(c, s)$ to determine whether the class c understands the

[4] *sup* is a partial function, since we consider only classes defined locally in the corresponding project (*i.e.* ignoring classes from external frameworks or the base system—*e.g.* class *Object*).

[5] To avoid the usage of another function, we say that an interface implements another interface.

$$subh(t) = \begin{cases} (sup^{-1})^*(t) \cup t, \text{if } t \in C \\ \{(sup^{-1})^*(t'), t' \in t^* \cap C\} \cup t^* \\ \quad \text{where } t^* = t \cup (impl^{-1})^*(t), \text{if } t \in I \end{cases} \quad (11)$$

$$impl(t,s) = \{m \in M | msg(m) = s \wedge def_m(m) \in subh(t)\} \quad (12)$$

Smalltalk:

$$undr(c,s) = s \in sel_t(c) \vee undr(sup(c), s) \quad (13)$$

$$intr(s) = \{c \in C | undr(c,s) \wedge \neg undr(sup(c), s)\} \quad (14)$$

$$isp(cs) = (\exists c \in intr(sel(cs))) s.t. \; (|impl(c, sel(cs))| > 1) \quad (15)$$

$$isp(s) = (\exists c \in intr(s)) s.t. (|impl(c,s)| > 1) \quad (16)$$

Java:

$$isp_t(cs) = |impl(t, sel(cs))| > 1 \quad (17)$$

$$isp_t(s) = (\exists t \in T) s.t. (|impl(t,s)| > 1) \quad (18)$$

Fig. 5: Computing polymorphic metrics

selector s, either because it defines the method m such that $msg(m) = s$ or one of the classes in its "superclass-chain" does (13). We also use the function $intr(s)$ to find all classes "introducing" the selector s, *i.e.* defining a method m such that $msg(m) = s$ and not having the superclass which understands it (14). We then say that the call site cs is polymorphic, written $isp(cs)$, if there exists a class c introducing the selector $sel(cs)$ and having at least one more method with the selector s in its subhierarchy[6] (15).

In Java, for each call site cs we know the compile-time type of the receiver, $t = rec(cs)$, so we need to define the polymorphic call site with respect to t. We say that the call site cs is polymorphic if there are at least two methods implementing the selector $sel(cs)$ and being defined in the subhierarchy of the type t (17).

Definition 2: A selector s is *polymorphic* if it can be a selector of a polymorphic call site. By extension, we consider a *polymorphic method* to be a method that implements a polymorphic selector.

Again, since in Smalltalk we don't have the information about the static type of the receiver, this means that the selector s is polymorphic if there is at least one class c defining a method m such that $msg(m) = s$ and having at least one class in its subhierarchy defining another method with the same selector (16). In Java, we need to define the term of polymorphic selector with respect to the possible type of the receiver (18).

Consider the example in Listing 1. The selector $s =$ basicDisplayBox(Point, Point) is not polymorphic with respect to the possible type of the receiver $t =$ HTMLTextAreaFigure, or with respect to DiamondFigure, but $isp_t(s) = true$, when $t =$ TextAreaFigure, or when $t =$ AbstractFigure.

[6]In Smalltalk we do not consider methods implementing the same selector, but defined in classes without a common superclass already defining that selector. Such methods are "duck-typed".

III. Experimental Setup

Our study covers 111 systems written in Java and 1,128 systems written in Smalltalk.

We chose Smalltalk for its reputation as a "pure" object-oriented language (Smalltalk goes as far as implementing conditionals as polymorphic methods in the Boolean class hierarchy), and Java as a representative of a widely used, pragmatic object-oriented language. We statically analyze these systems to gather information about the usage of polymorphism and complexity it imposes to developers during program understanding.

- For the Smalltalk part, we took a snapshot of all the 1,850 software projects stored in the SqueakSource repository in early 2010. At that time SqueakSource contained the majority of all projects implemented in the open-source Smalltalk dialects Squeak and Pharo[7] and hence provided a representative set of Smalltalk projects from both industry and academia.
  We limit our analysis to projects containing more than 50 classes in order to exclude student projects and other small and likely less relevant projects; out of the 1,850 projects, 1,128 projects meet this criterion.
  These projects contain 125,825 classes and 1,637,228 methods in total.
- For Java we selected 111 open-source projects from the Qualitas Corpus — a curated collection of software systems representing widely known open-source Java software systems and libraries [19]. Although we suspect so, we cannot guarantee that the selection is representative for well-engineered and maintained open source Java software.
  The corpus consists of over 130,000 classes and 1,086,000 methods.
  For the Java corpus, we use the Pangea analysis infrastructure [4] which enables us to easily deploy our analysis on an entire Qualitas corpus.

A. Data processing

Each project is parsed in order to extract the relevant metrics. We employed Ecco and Monticello as parsers for the Smalltalk corpus [17], and VerveineJ and Moose for the Java corpus [9]. The data processing consisted of two steps:

1) **Method analysis.** To measure how polymorphism is used we traverse the body of every class in the system, each class c having a list of methods $def_m^{-1}(c)$ it implements. We keep track of the set of methods implemented in the project, as well as of the set of all methods that are either overridden or are overriding, and the set of their selectors. These are polymorphic selectors. We then calculate the metrics related to polymorphism, such as which methods in a system are involved in polymorphic call sites.
2) **Call site analysis.** In the second step, we traverse the body of each method, detecting all call sites within the method body. We then collect all call sites cs for which $isp(cs) = true$.

[7]www.pharo-project.org

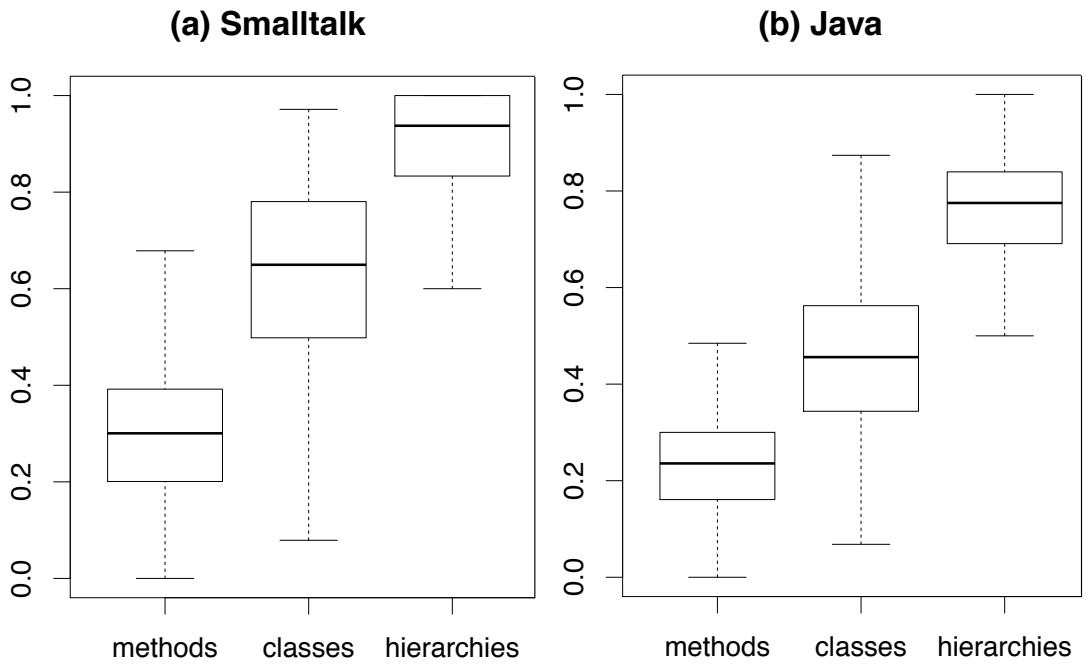

Fig. 6: Distribution of the proportions of methods, classes and hierarchies defining polymorphic selectors, in (a) Smalltalk, (b) Java

B. Data analysis

We are primarily concerned with the distribution of the metrics over the corpus, and, as a secondary concern, whether they are distributed similarly in Smalltalk and in Java. To this aim, we use boxplots to summarize the distribution of the metrics in the studied systems. When possible, we analyze varying degrees of aggregation (*e.g.* methods, classes and hierarchies) in order to evaluate how the results hold at each level of granularity, and to avoid ecological fallacies. Ecological fallacies can occur when one studies the data at the wrong abstraction level [16].

We use statistical tests when we deem them necessary (*e.g.* to compare the distribution of a metric over both Smalltalk and Java projects); most distributions we encounter depart from normality, so we use the Mann-Whitney U-test, which is non-parametric. As a measure for the non-parametric effect size, we use the A_{12} effect size statistics of Vargha and Delaney, instead of Cohen's d [21]; this effect size metric has for example been advocated by Arcuri and Briand in the case of comparing randomized algorithms [1]. A_{12} works on two samples, A and B; it indicates the probability that a random element taken from A is larger than a random element taken from B. An A_{12} of 0.5 is a null effect size, and values nearer to 0 or 1 indicate a larger effect size (with a random element taken from A being smaller than a random element taken from B, respectively larger). The R library we use to compute the effect size also gives us an equivalent Cohen's d, which we report as it is easier to interpret; commonly accepted thresholds for d are 0.2 for a small effect size, 0.4 for a medium effect size, and 0.8 for a large effect size.

IV. Experimental Results

In this section we discuss in turn the research questions that we proposed in Section 1.

For both Smalltalk and Java 99% of the inspected projects define polymorphic methods and polymorphic call sites.

A. Implementing polymorphism

Figure 6 shows the proportion of methods that are potentially involved in a polymorphic call site (*i.e.* methods whose selectors are polymorphic) for both Smalltalk (left) and Java (right). This information is then aggregated to the level of classes, and hierarchies.

Figure 6 shows that, for Smalltalk, in median:

1) **At least one out of four methods (31%)** in the project implements a polymorphic selector.
2) **63% of all classes** in the project implement at least one of those methods
3) **Almost all class hierarchies (97%)** in the project include at least one polymorphic selector

Figure 6 shows that for Java the numbers are lower but they still reveal that polymorphic method implementations are in median:

1) **At least one out of five methods (24%)** in the project implements a polymorphic selector
2) **Almost half (44%) of the classes** in the project implement at least one of those methods
3) **More than three quarters (76%) of all class hierarchies** in the project include at least one polymorphic selector

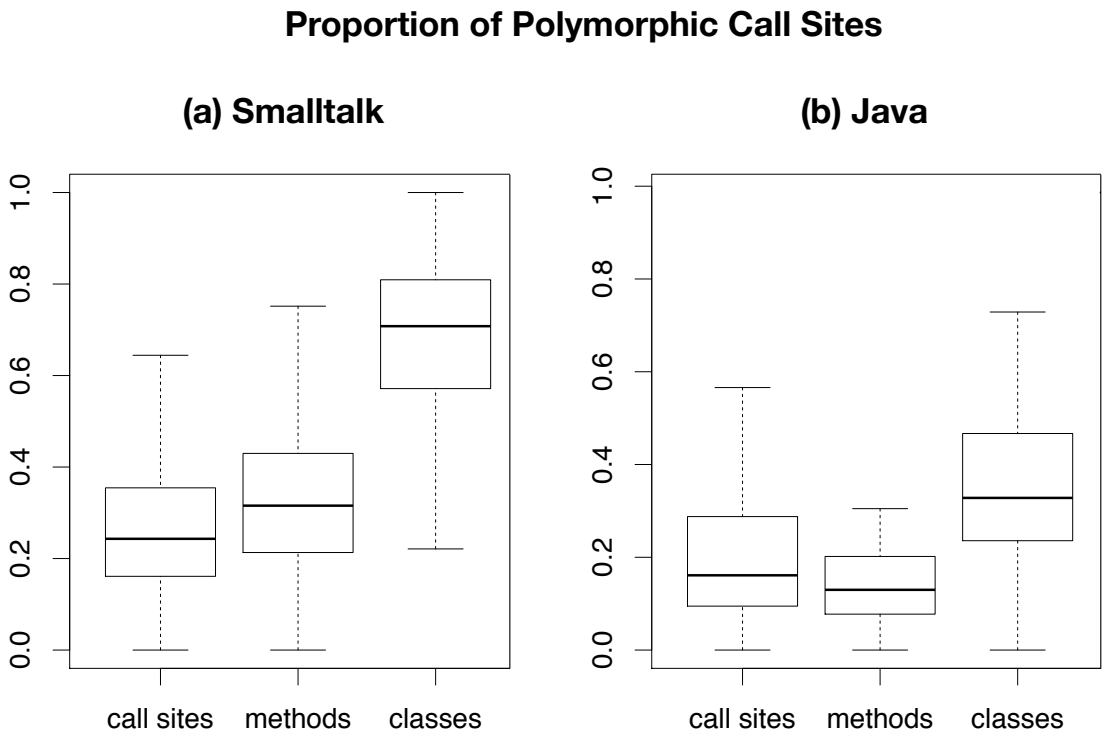

Fig. 7: Distribution of the proportion of polymorphic call sites, as well as methods and classes having them, in (a) Smalltalk, (b) Java

Based on this data, we can now answer our Research Question 1 by stating that:

> In a majority of projects in both Smalltalk and Java more than a quarter of the methods are implementations of polymorphic selectors.

Since there seems to be a difference between Smalltalk and Java we also compare the corresponding aggregation levels between the two languages. We obtain $p < 10^{-5}$ in all cases for the Mann-Whitney test. A_{12} for methods is 0.62 (a randomly chosen Smalltalk project has a 62% probability of having a higher ratio of polymorphic methods than a randomly chosen Java project), translating to a Cohen's d of 0.44 (medium effect size). For classes, A_{12} = 0.76, and d = 0.91 (large effect size); for hierarchies, A_{12} = 0.79, and d = 0.82 (large effect size).

Based on this statistical analysis we conclude that:

> Methods implementing polymorphic selectors are statistically more prevalent in the Smalltalk Corpus than in the Java corpus, especially at the class and hierarchy aggregation level.

B. Using polymorphism

In Figure 7 we present the proportion of all call sites that are polymorphic as well as the proportion of all methods, and classes defining at least one polymorphic call site (on the left for Smalltalk, on the right for Java).

For Smalltalk, Figure 7 shows that in median:

- **A quarter of all the call sites** in the project (24%) are considered to be polymorphic call sites within the algorithm we have implemented
- **A third of the methods** in a project (32%) contain a call site considered to be a polymorphic
- **Three quarters of the classes** in a project (78%) contain a polymorphic call site.

For Java, Figure 7 shows that in median:

- **16% of all call sites** in a project cannot be resolved at compile time using the analysis we have explained in the Terminology section
- **12% of all methods** in a project include a call site which is considered not capable of being resolved at compile time
- **30% of classes** in a project have at least one polymorphic call site

Surprisingly, in Java, the proportions of methods defining a polymorphic call site is lower than the proportion of polymorphic call sites; we hypothesize that polymorphic call sites cluster in methods, although this would have to be verified in future studies.

We can now answer the research question with:

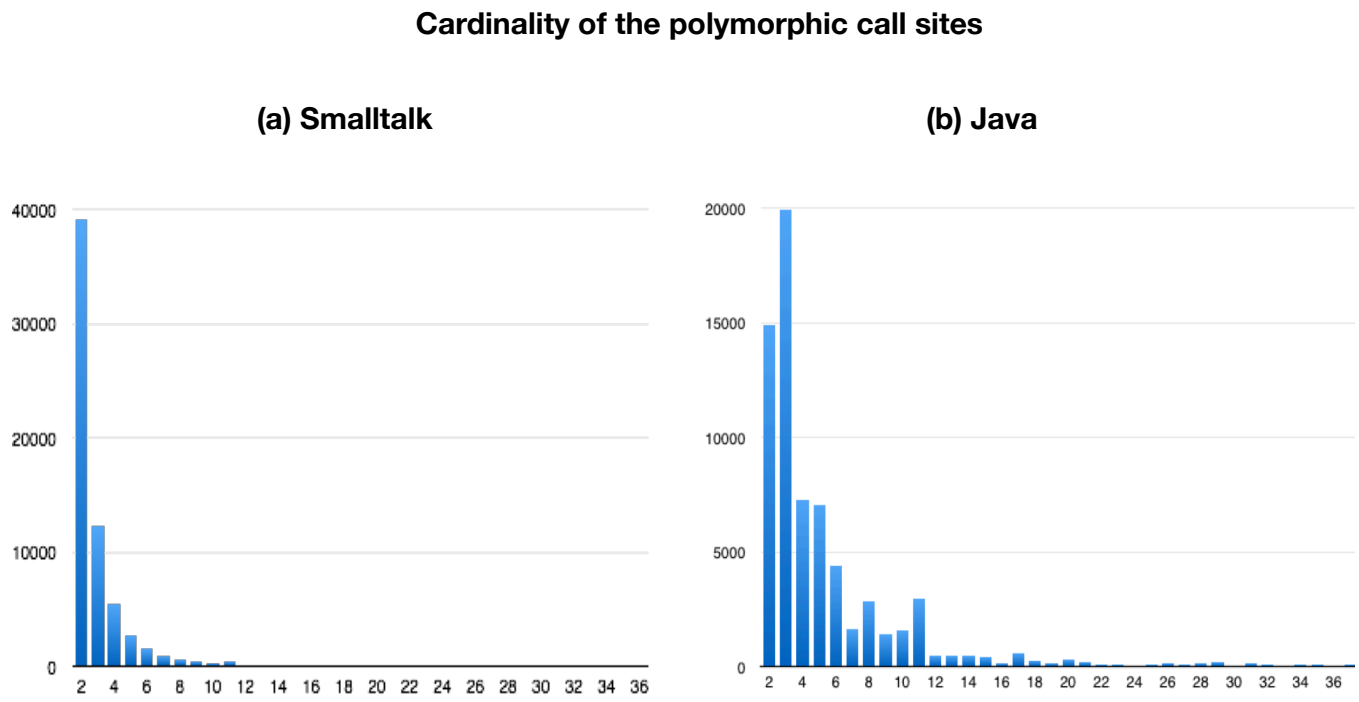

Fig. 8: The cardinality of the polymorphic call sites in the Smalltalk and the Java corpus, respectively

> For a majority of the projects, more than one in ten (for Java) and one in five (for Smalltalk) of the call sites in a system are considered to be polymorphic.

This means that the situation in the example in the introduction is frequently encountered by developers working in both languages.

It is visually intuitive that the percentage of call sites which are polymorphic is much higher in Smalltalk than in Java. To verify this we do a Mann-Whitney test. This holds at the level of call sites, methods, and classes ($p < 10^{-8}$ in all cases for the Mann-Whitney test). A_{12} for call sites is 0.65 in favor of Smalltalk (equivalent Cohen's d of 0.52, medium effect size); for methods, we have A_{12} = 0.83 and d = 1.37 (large effect size); for classes, A_{12} = 0.87, and d = 1.62 (large effect size).

We thus conclude that, statistically speaking:

> The usage of polymorphism is significantly more prevalent in the Smalltalk corpus than in the Java corpus.

This second observation means that tools to support program understanding in the presence of polymorphism are even more important in the Smalltalk context than in the Java context.

C. Understanding Polymorphism

The cardinality of a polymorphic call site is defined as the size of the set defined earlier as $impl(t, sel(cs))$ (12).

The distribution of the cardinality of the polymorphic call sites in Smalltalk is presented in Figure 8. We observe that:

- More than 75% of the polymorphic call sites have cardinality two or three.
- Most of the call sites (90%) that cannot be resolved at compile time using the implemented analysis have strictly less than 6 candidates.

Figure 8 shows the results of running a similar analysis for Java. Based on the analysis of 71K polymorphic call sites in the corpus, we observe that:

- There are fewer call sites with two candidates than with three candidates.
- More than 50% of the call sites have a cardinality of two or three
- More than 75% of the polymorphic call sites have cardinality less than seven
- Most of the call sites (90%) have less than 12 candidates

In some cases, we see call sites with a very large cardinality. Table I presents several of the extreme cases we investigated. In all the cases we have more than 100 potential implementations being called at a call site. We can observe some design patterns that are responsible for this, including Visitor (for jruby) and Command (for ant).

Several of the polymorphic sites with large cardinalities (e.g. the ones in Table I) also happen to occur a significant number of times. This probably is the result of a given call site occurring in multiple places in a given project.

TABLE I: The largest cardinalities in the Java corpus

System	Method Name	Cardinality
weka-3.7.5	RevisionHandler.getRevision()	545
spring-3.0.5	InitializingBean.afterPropertiesSet()	216
ant-1.8.2	Task.execute()	201
weka-3.7.5	CapabilitiesHandler.getCapabilities()	167
jruby-1.5.2	Node.interpret(Ruby,ThreadContext,...)	148

To finally answer the research question, we conclude:

> In both the languages a strong majority (75%) of the polymorphic call sites have a cardinality of up to six and a vast majority (90%) have a cardinality of less than twelve. Corner cases can have hundreds of candidates.

V. Implications and Future Directions

The study presented above only considers the prevalence of polymorphism in open source software. As such, it is only a first step. In this section we consider a series of further research questions that the results of the study open up.

Project Usage Patterns

First, and foremost, the numbers show that polymorphism is a widely used concept in the case of the two studied languages. There are however, several exceptions. Notably, in the case of RQ1 we have seen also strong outliers — systems in which only 10% of the classes define polymorphic selectors. These results raise the question whether the degree of polymorphism present might depend on the type of system (*e.g.* infrastructure vs application code), the application domain, or perhaps even programmer experience.

Polymorphism Usage Idioms

Since the results of the RQ1 show that at least one in five methods is polymorphic in the studied corpora, one may ask to what end is polymorphism used? Are most cases of polymorphism instances of the Template Method design pattern? To what extent can usages of polymorphism be classified as supporting particular idioms or design patterns? Can certain usages be considered as "best practice" and others as "code smells"?

Impact on Program Understanding

Given that the presence of polymorphic call sites is so high, based on the results of RQ2, it would be important to discover to what extent polymorphic call sites pose a problem for program understanding. These results also bring up the question of how polymorphism influences the quality of the source code, *e.g.* whether it is more understandable for developers to use type checking or polymorphism.

With the cardinality of a polymorphic call site measure (RQ3) we tried to estimate the complexity of understanding a given polymorphic call site. However, the cardinality is a preliminary measure. For example, often the methods that could be invoked at a given call site will also contain polymorphic call sites in their turn. A user would have to follow such a *polymorphic call chain* to fully understand the software.

A better proxy for the difficulty of understanding a polymorphic call site would take into account also a tree-like structure of polymorphic call chains. Such a metric would be the equivalent of cyclomatic complexity computed on the call graph induced by the compound polymorphic calls. In preliminary studies we observed some extreme polymorphic call chains that span dozens of methods.

Polymorphic call chains could be combined with other measures to eventually quantify the impact of polymorphism on comprehension. Eventually, empirical studies would have to validate such metrics.

Duck Typing

One particular type of polymorphism is cross-hierarchy polymorphism, also known as duck typing. This kind of language feature is usually encountered in dynamically typed programming languages, but can be simulated also in statically typed languages with the use of interfaces. It would be interesting to know how much of this polymorphism is due to duck typing. Moreover, it would be important to quantify the impact of duck typing on program understanding. We expect duck typing to actually have a more drastic impact on comprehension.

Static and dynamic detection: What is the Ground Truth?

In this paper we used the CHA algorithm (*Class Hierarchy Analysis*) [7] for Java and a modified version for Smalltalk to calculate the candidates for the call sites. We have encountered cases where there are more than 100 method candidates for a polymorphic call site. With more advanced techniques of static analysis, like the RTA algorithm (*Rapid Type Analysis*) [2], it might be possible to get more precise results. While CHA finds the possible method candidates at the call site based only on the declared type of the receiver, RTA takes into account the set of all classes instantiated thus far.

Even with the most advanced static analysis, there will be cases where a call site cannot be resolved statically. The downside of static analyses are the false positives (potential polymorphism might not occur in practice), while dynamic analyses suffer from false negatives (some polymorphic calls may be missed in any given run). This leads us to the question of the best technique to detect polymorphic call sites: what is the ground truth, and which technique yields the best approximation?

How To Improve Program Comprehension Tools?

Given the prevalence of polymorphism in the two corpora we studied, we propose the need for dedicated tool support in the context of program comprehension in the presence of polymorphism. Based on the results of this study, we can propose several needs of IDEs for program comprehension in the presence of polymorphism.

- The outcome of RQ1 is that at least one in five methods implements a polymorphic selector. When displaying a

given method, it would be useful for developers to know which other methods implement the same selector in the class hierarchy of the selected method.
- The results of RQ2 state that at least one in five call sites is polymorphic. Developers should be aware of this, since code reading is one of the most time-consuming activities [3]. A developer trying to understand a given polymorphic call site should be able to easily access the source code of all possible method candidates.
- The conclusion of RQ3 is that 75% of polymorphic call sites have at most six possible method candidates. Tools specialized in displaying up to seven method implementations would be useful in the majority of the cases. For the remaining cases, a scalable tool would have to be designed.

One challenge here is to make such tools always-available, since polymorphism is omnipresent, yet not too intrusive at the same time.

VI. Threats to Validity

Construct Validity. The best way to detect polymorphic methods would have been to use a combination of both static and dynamic analysis. This threatens the validity of our study since we are unable to know whether polymorphic methods are parts of "hot spots" in the source code, or whether they are actually never executed. However, our goal was to look at the problem from the perspective of the IDE where only static information is normally available.

One other threat to the validity of our results is the static analysis algorithms we have used. It could be that more advanced analysis would provide more precise results. However, present IDEs do not typically offer advanced analyses.

One final threat to construct validity concern possible imprecisions in detecting polymorphic methods, due to cross-class polymorphism (*i.e.* duck typing) in Smalltalk.

Internal Validity. We consider only user-defined polymorphism, and not polymorphism defined in libraries being employed by the analyzed projects, nor the usage of these polymorphic library methods. Moreover, we consider polymorphism only within the boundaries of a system, without taking into consideration library classes being extended in the subject systems, thus there might be more user-defined polymorphic methods whose superclass implementations are outside of project boundaries.

External Validity. Since our study features only open-source projects, we cannot generalize our findings to industrial projects. For the Smalltalk corpus, we only considered projects that are found in the SqueakSource repository. Although SqueakSource was at that time the *de facto* standard source code repository for Squeak and Pharo developers, we cannot be sure of how much the results generalize to Smalltalk code outside of SqueakSource, such as Smalltalk code produced by VisualWorks developers. We only take into account Smalltalk projects with more than 50 classes to filter out projects that might be toy or experimental projects. We believe such filtering increases the representativeness of our results, however, it might also impose a threat. Similarly, our corpus of Java systems contains only open-source code, and was built based on the availability of systems. As such, we cannot make strong claims about its representativeness. It does however contain popular applications, such as ArgoUML, components of Apache, FindBugs, etc.

The two sub-corpora exhibit different characteristics: Notably, polymorphism is much more prevalent in Smalltalk—a "pure OO" language—than in Java. Extending the study to other OO languages may yield other insights.

VII. Related Work

Tempero *et al.* conducted an empirical study of method overriding — a concept closely related to polymorphism — in a corpus of 100 open source Java systems [19]. They employed various metrics, such as the number of overriding methods, the number of inherited methods, and the number of classes with replaced implementations; they did not use method-level metrics as we do in our study. Their study showed that most sub-classes override at least one method and many classes only declare overriding methods.

In other work, Tempero *et al.* analyzed the use of inheritance in Java, on a corpus of 97 systems [20]. To this aim, they defined and extracted a suite of structured metrics for quantifying inheritance. Different from previous studies, their work showed a high use of inheritance, variation in the use of inheritance between interfaces and classes, and a different use of inheritance when applied to external libraries. In short, Tempero *et al.*'s study indicated that a high use of inheritance is a characteristic of accepted Java programming practice. While in their work the aim was the study of inheritance in general, in this paper we focus on the investigation of polymorphic methods in the context of inheritance hierarchies.

Several researchers have investigated the relationship between code maintainability and the depth of inheritance. Daly *et al.* [6] found that a system with three levels of inheritance was easier to maintain than a corresponding system with no inheritance; on the other hand, a system with five levels was found to take longer to modify than the one with zero or three levels of inheritance. Cartwright *et al.* and Harrison *et al.* replicated Daly *et al.*'s study, obtaining contradictory results: Cartwright *et al.* found inheritance to have a positive effect on maintenance time [5], whereas the study of Harrison *et al.* revealed that a system with zero inheritance was easier to modify than the equivalent systems with three or five levels of inheritance [13].

Grechanik *et al.* were among the first to perform an empirical study using a large body of source code [11]: They analyze common source code patterns in a large-scale open source code repository, composed of 2,080 Java projects randomly selected from Sourceforge. Their investigation provided a number of interesting and surprising findings, such as that most methods have one or zero arguments, few methods are overridden, most inheritance hierarchies are flat (*i.e.* depth of one), and almost half of the classes are not inherited from any

classes. Our study resembles that of Grechanik *et al.* in terms of project corpus size and methodology followed; the main difference is that we focus on the prevalence of polymorphism, with a number of dedicated metrics, whereas Grechanik *et al.* answer 32 different research questions covering many aspects of OO programming, ranging from number of static classes to number of exceptions per method.

Parnin *et al.* studied the adoption of Java Generics, a form of parametric polymorphism [15]. They found that developers adopted it in the new code they wrote (after the release of Java 1.5, when generics were added), but that adoption was often driven by a single "champion". Also, the old code was not often converted to use generics. Our study differs from theirs as we analyze regular polymorphism only, as parametric polymorphism is more concerned with type safety. In addition, Smalltalk does not support parametric polymorphism.

VIII. Conclusions

We performed an empirical study of polymorphism on two corpora of open source systems written in Smalltalk and Java, respectively. We found that in both languages, polymorphism is frequently used: nearly all projects we analyzed take advantage of polymorphism by implementing polymorphic selectors and by invoking such selectors.

In both languages a strong majority of the polymorphic call sites have a cardinality of up to six and a vast majority have a cardinality of less than twelve. This means that any difficulty in code understanding associated with such polymorphic call sites occurs frequently.

We learned that in Java, half of the polymorphic call sites have a cardinality of two or three, and three quarters have a cardinality of less than seven. We consequently argue that tool support to improve program understanding in the presence of polymorphism could start by tackling these simpler cases.

Smalltalk uses polymorphism to a much greater extent; more than 60% of all classes implement a polymorphic selector in Smalltalk projects, while around 40% do the same in Java projects (for methods: 31% in Smalltalk compared to 24% in Java). Since one fifth of all method call sites in Smalltalk projects are polymorphic, and one third of all methods are implementations of a polymorphic selector, solving the problems associated with understanding polymorphic call sites is of higher priority for Smalltalk than for Java.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015). D. Röthlisberger is partially funded by FONDECYT Project 1140068. We also gratefully acknowledge the financial support of the Swiss Group for Object-Oriented Systems and Environments (CHOOSE) and the European Smalltalk User Group (ESUG).

References

[1] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering, (ICSE 2011)*, pages 1–10, 2011.

[2] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. *SIGPLAN Not.*, 31(10):324–341, Oct. 1996.

[3] B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, Jan. 2001.

[4] A. Caracciolo, A. Chis, B. Spasojević, and M. Lungu. Pangea: A workbench for statically analyzing multi-language software corpora. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 71–76. IEEE, Sept. 2014.

[5] M. Cartwright. An empirical view of inheritance. *Information Software Technology*, 40(14):795–799, 1998.

[6] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1:109–132, 1996.

[7] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 77–101, Aarhus, Denmark, Aug. 1995. Springer-Verlag.

[8] S. Demeyer, S. Ducasse, K. Mens, A. Trifu, and R. Vasa. Report of the ECOOP'03 workshop on object-oriented reengineering. In *Object-Oriented Technology (ECOOP'03 Workshop Reader)*, LNCS, pages 72–85. Springer-Verlag, 2003.

[9] S. Ducasse, T. Gîrba, and O. Nierstrasz. Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, pages 99–102, Sept. 2005. Tool demo.

[10] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[11] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of ESEM 2010*, pages 11:1–11:10. ACM, 2010.

[12] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 112–121, Los Alamitos CA, 2005. IEEE Computer Society Press.

[13] R. Harrison, S. Counsell, and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of System and Software*, 52:173–179, June 2000.

[14] M. Lungu, M. Lanza, and O. Nierstrasz. Evolutionary and collaborative software architecture recovery with Softwarenaut. *Science of Computer Programming*, 79(0):204 – 223, 2014.

[15] C. Parnin, C. Bird, and E. R. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *MSR 2011: Proceedings of the 8th International Working Conference on Mining Software Repositories*, pages 3–12, 2011.

[16] D. Posnett, V. Filkov, and P. Devanbu. Ecological inference in empirical software engineering. In *Proceedings of the 26th ACM/IEEE International Conference on Automated Software Engineering (ASE 2011)*, pages 362–371, 2011.

[17] R. Robbes, M. Lungu, and D. Roethlisberger. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, pages 56:1 – 56:11, 2012.

[18] D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting static source views in IDEs with dynamic metrics. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*, pages 253–262, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[19] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336 –345, Dec. 2010.

[20] E. Tempero, J. Noble, and H. Melton. How do Java programs use inheritance? An empirical study of inheritance in Java software. In *Proceedings of the 22nd European conference on Object-Oriented Programming (ECOOP 2008)*, pages 667–691. Springer-Verlag, 2008.

[21] A. Vargha and H. D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[22] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.