Polymorphism in the Spotlight: How Developers Use it in Practice

Romain Robbes[1], David Röthlisberger[2], and Mircea Lungu[3]

[1] PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Chile
rrobbes@dcc.uchile.cl
[2] School of Informatics and Telecommunications
Universidad Diego Portales – Chile
davidroe@mail.udp.cl
[3] Software Composition Group
University of Bern – Switzerland
lungu@iam.unibe.ch

Abstract. Polymorphism—the ability of a client to send a message that is implemented by objects of different types without needing to know the exact type of the object that will respond to the message—is a cornerstone of object-oriented programming. It is intended as a way to facilitate the modularity and extensibility of software systems by hiding the variability in behavior behind a unified interface and allowing the client and provider to be decoupled in their evolution. The downside of polymorphism is that it scatters the implementation of the behavior over multiple classes, which could result in code that is harder to understand. In this research we study how polymorphism is used in one of the oldest and in one of the most widespread object-oriented programming languages: Smalltalk and Java. For this we study a large corpus of several hundreds of systems to answer questions about the prevalence of polymorphism in practice. We also estimate the burden polymorphism imposes on developers maintaining software systems by determining how scattered are the implementations of polymorphic methods in class hierarchies.

Keywords: object-oriented programming, polymorphism, empirical software engineering, software maintenance

1 Introduction

Polymorphism is one the defining features of object-oriented programming languages. It allows different objects to be replaced with one another as long as they respond to the same interface. The classes of these objects provide different implementations of the same message. As a result of subclassing some classes might inherit an implementation and not redefine it, hence the implementation of a particular message is often scattered over various implementing classes throughout the hierarchy. While polymorphism is supposed to reduce the

costs of maintaining a software system, its influence on scattering the implementation might actually increase the complexity and hence the maintenance costs [4,6,13]. Moreover, it is often possible that developers are using alternate solutions instead of polymorphism to allow for behavior variability, such as switch statements and an imperative programming style.

In this paper, we set out to learn about the actual use of polymorphism in practice by studying a large corpus of open source software systems. We conduct an empirical study to answer the questions of whether polymorphism is prevalently applied in practice, how large, distributed or scattered (and hence complex to understand) are the implementations of a polymorphic method, and how polymorphism evolves over time. Our study covers several hundred systems written in Java and Smalltalk. We choose 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.

The Smalltalk projects are coming from Squeaksource, a software super-repository hosting the majority of Smalltalk code produced by the Squeak and Pharo (`www.pharo-project.org`) open-source communities. The Java projects are extracted from a corpus of 144 OSS projects available online. We statically analyze these systems to gather information about the implementation of polymorphic messages, their usage, and their complexity.

Research questions. We structure our study of polymorphism around the following questions:

1. How prevalent are polymorphic method implementations in object-oriented systems? Polymorphism is a cornerstone of OOP: do programmers use it as extensively as its reputation would lead one to believe?
2. How often are polymorphic methods invoked? Are polymorphic methods used prevalently in software systems, or are they unlikely occurrences?
3. How complex are typical implementations of polymorphic methods in terms of number of lines of code or methods they invoke? Is the average polymorphic method larger or shorter than its non-polymorphic counterpart?
4. How scattered are the implementations of a polymorphic message in a class hierarchy? Do polymorphic messages usually cover the entire class hierarchy, or are they more affecting specific parts of it?

For each question, we contrast Smalltalk with Java to answer an overarching research question: Is the usage, the complexity of polymorphism dependent on the programming language and its associated culture?

As a result of this study, we obtain a better understanding of the phenomenon of polymorphism, its practical relevance, and its complexity.

Structure of the Paper. We start with reporting on related work in the field (Section 2). Next, we explore our experimental methodology and the analysis infrastructure (Section 3). In Section 4 we report on the findings regarding the prevalence of polymorphism in practice, both concerning the implementation of polymorphic messages and their usage. Subsequently, we characterize in Section 5 the implementation of polymorphism in class hierarchies, that is, how

complex polymorphic methods are in terms of lines of code and sent messages and how scattered and distributed they are over the class hierarchies. We report on potential threats to the validity of this study (Section 6) before concluding in Section 7.

2 Related Work

The work most similar to the one presented in this paper was performed by Tempero *et al.* on a corpus of 100 open source Java systems [14]. The authors conducted an empirical study aimed at understanding the use of a concept related to—but different from—polymorphism, *i.e.* method overriding. For their study, they employed different metrics from the ones used in this paper, such as number of overriding methods, number of inherited methods, and number of classes with replaced implementations; the authors did not use method-level metrics as we will do in our study. The empirical study of Tempero *et al.* showed that most sub-classes override at least one method and many classes only declare overriding methods.

Other studies focused on the empirical analysis of inheritance hierarchies. Several researchers investigated the relationship between code maintainability of depth of inheritance. Daly *et al.* [3] found that a system with three levels of inheritance was easier to maintain than the corresponding system with no inheritance; on the other hand, five levels was found to take longer to modify than zero or three levels of inheritance. Cartwright *et al.* and Harrison *et al.* replicated Daly *et al.*'s study, obtaining contradicting results: Cartwright *et al.* found inheritance to have a positive effect on maintenance time [2], 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 [8]. Tempero *et al.* analyzed the use of inheritance in Java, on a corpus of 97 systems [15]. 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.

Grechanik *et al.* were among the first to perform an empirical study using a large body of source code [7]: 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 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 reminds the one of Grechanik *et al.* in terms of project corpus size and methodology followed; however, the main difference between the two studies is that

we focus on the understanding 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 [10]. 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, 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.

3 Experimental Setup

To study polymorphism in practice, we perform an analysis of a large repository of Smalltalk projects and a corpus of Java projects. This section describes the experimental setup, that is, the methodology applied to perform the analysis and the analysis infrastructure, after starting with a few definitions of terms used throughout the paper.

3.1 Definitions

For the definition of terms regarding polymorphism, we reuse the Smalltalk terminology that clearly differentiates between a message and a method.

A message is the name under which behavior of an object is identified. A message is sent to an object for which the method matching the message name is then executed.

A method is the concrete implementation of a message in a concrete class. Instances of subclasses not implementing a method with the same name will execute the superclass method matching the message name. Re-implementing a method with the message name in a subclass is referred to as overriding. The overriding method can, but does not have to, invoke the superclass method with the same name (referred to as as super call).

Message sending is the act of sending a message to an object, ultimately triggering the invocation of a method. Due to overriding, the actually invoked method may be implemented in superclasses of the class of that particular object. A class responds to a message if it, or one of its superclasses, implement a method with the name of the message.

A polymorphic method is a method being overridden by or overriding a method with the same name, *i.e.* a method implementing a polymorphic message which has been implemented by one or more other classes in the same class hierarchy. In our study, we only consider the part of a class hierarchy that has been defined locally in the corresponding project (*i.e.* ignoring classes from external frameworks or the base system—*e.g.* class Object). Similarly, we define a

polymorphic class, hierarchy, or project as one which implements, at least, one polymorphic message.

A call site is a method invoking another method; in our study we are interested in polymorphic call sites, that is, methods invoking a polymorphic method.

3.2 Methodology

Smalltalk corpus For the Smalltalk part, we took a snapshot of all the 1,850 software projects stored in the Squeaksource repository in early 2010. Squeaksource contains the majority of all projects implemented in the open-source Smalltalk dialects Squeak and Pharo and hence provides 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 small projects; out of the 1,850 projects, 1,128 projects meet this criteria. These projects contain 125,825 classes and 1,637,228 methods in total.

Java corpus For Java we selected 144 large open-source projects, *e.g.* from the Apache projects, all of them have at least 50 classes.[4] The corpus of Java projects consists of 120,877 classes and 918,447 methods.

Data processing Each project is parsed in order to extract the relevant metrics. As parsers, we employed Ecco and Monticello for the Smalltalk corpus [12], and Moose for the Java corpus [5]. The data processing consisted of two steps:

1) High-level analysis. For each project, we start by modeling its hierarchies. A hierarchy is composed of classes, each class having subclasses and a list of methods that it implements. We then traverse the hierarchies in order to detect methods with identical signatures that are implemented more than once in the hierarchy; these are the implementors of a polymorphic message at the level of the hierarchy. We keep track of the set of messages defined in all the hierarchies of a project as the set of user-defined polymorphic messages. We also compute metrics related to overriding, such as how many classes understand a given message, either because themselves or a superclass implements it. In the case of Java, interfaces are detected, and classes implementing them will have the messages from the interface they implement counted properly.

2) AST analysis. To measure how polymorphism is used, we parse the body of each method, detecting the messages it sends. We then compare those with the list of user-defined polymorphic messages. We also measure the complexity of a method in terms of lines of code and number of messages sent, as these are simple metrics, yet highly correlated with complexity metrics such as McCabe [9]. We also analyze the structure of the AST to infer whether the method presents some of the "code smells" we look for (i.e., methods simply raising an error, or NOP methods).

[4] A list of the selected Java projects is available online at the following URL: http://scg.unibe.ch/research/FamixCorpus.

Data analysis We are primarily concerned with the distribution of the metrics over the corpus, and whether they behave similarly in Smalltalk and in Java. To this aim, we use boxplots to summarize the distribution of the metrics. When possible, we analyze varying degrees of aggregation (*e.g.* methods, classes, hierarchies, and projects) in order to evaluate how the results hold at each level of granularity, and to avoid ecological fallacies [11].

We use statistical tests when we deem necessary (*e.g.* to compare the behavior of a metric over both Smalltalk and Java projects); most distributions we encounter depart from normality, as such we use the Mann-Whitney U-test, which is non-parametric. As a measure for the non-parametric effect size, we follow the recommendation of Arcuri and Briand [1], and employ the A_{12} effect size statistics of Vargha and Delaney, instead of Cohen's d [16]. 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, values nearer to 0 or 1 indicate a larger effect size (with A being smaller than 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. The results of our study are presented next. We start with the overall prevalence of polymorphism.

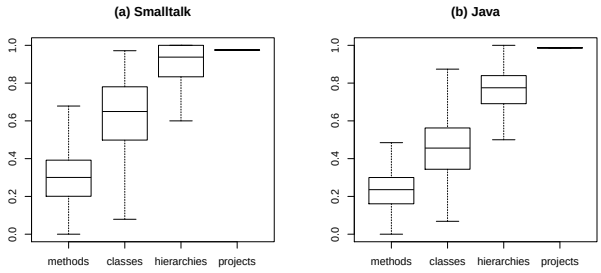

Fig. 1. Distribution of the proportions of methods, classes, hierarchies, and projects defining polymorphic messages, in (a) Smalltalk, (b) Java

4 The prevalence of polymorphism

In this section we study how prevalent polymorphism is in Smalltalk and Java projects and how polymorphic message are distributed over class hierarchies.

First, we look at the implementation of polymorphic messages, that is, how many polymorphic messages exist or how many methods implement these messages (research question 1). Second, we study how these polymorphic messages are actually used, *i.e.* , how many methods, classes and packages send polymorphic messages (research question 2).

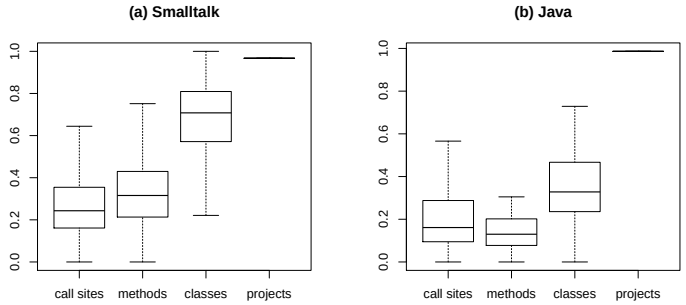

Fig. 2. Distribution of the proportion of call sites, methods, classes, and projects sending user-defined polymorphic messages, in (a) Smalltalk, (b) Java

4.1 Implementing polymorphism

Figure 1 shows the proportion of methods, classes, hierarchies, and projects implementing polymorphic messages (left Smalltalk, right Java).

Looking at the implementation of polymorphism in Smalltalk, we discover that around 31% of all methods are polymorphic (median value as shown in the boxplot in Figure 1), and 63% of all classes implement at least one polymorphic method (both proportions have a high variance). For hierarchies and projects, the proportions are even higher: around 97% of all class hierarchies respond to at least one polymorphic message and 99% of all Smalltalk projects take advantage of polymorphism. In Smalltalk, polymorphism is a very prevalent phenomenon.

For Java polymorphism is less prevalent: Only around 24% of all methods are an implementation of a polymorphic message, and only 44% of all classes implement one or more polymorphic messages. With 76%, the proportion of hierarchies responding to a polymorphic message is also much lower than for Smalltalk projects (97%). As nonetheless 99% of all Java projects apply polymorphism, we conclude that polymorphism is a well-known concept also in Java even though it is clearly not as prevalent as in Smalltalk.

Beyond the boxplots, polymorphism in Smalltalk is statistically much more prevalent than the one in Java at the level of methods, classes, and hierarchies ($p < 10^{-5}$ 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).

4.2 Using polymorphism

In Figure 2 we present the proportion of all message sends that are polymorphic as well as the proportion of all methods, classes and projects that send at least one polymorphic message (on the left for Smalltalk, on the right for Java).

For Smalltalk 24% of all message sends are polymorphic. The proportion of methods sending polymorphic messages is 32%, the proportion of classes us-

ing polymorphic messages is, with 78%, much higher. All projects implementing polymorphic messages also use them, so the proportion of projects using polymorphism is 99%. The proportion of methods, classes sending polymorphic messages is slightly higher than the respective proportions for implementing polymorphic messages, which was to be expected. Like the implementation of polymorphic messages, the usages also vary considerably from project to project.

For Java the proportions for polymorphic message sending is much lower than in Smalltalk. Since Java projects implement fewer polymorphic messages, it is clear that they also send fewer such messages. Concretely, in the Java projects only 16% of all message sends are polymorphic and, similarly, only 12% of all methods send a polymorphic message[5]. The proportion of classes sending at least one polymorphic message is, with 30%, higher, but still much lower than on the Smalltalk side (78%). All projects defining polymorphic messages also use them.

Statistically speaking, the usage of polymorphism is also significantly more prevalent in the Smalltalk corpus than in the Java corpus. 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).

Summary: To summarize the results of this section studying the prevalence of polymorphism in Smalltalk and Java projects we can state that in both languages polymorphic methods are frequently defined and used. Nearly all projects in both languages define and use polymorphism. On the level of classes or methods, it turns out that polymorphism is more prevalent in Smalltalk than in Java (*e.g.* 63% in Smalltalk versus 44% of all classes define polymorphism, or 24% in Smalltalk versus only 16% of all message sends in Java are polymorphic). Nonetheless we conclude that in both languages polymorphism can considered to be a frequently and prevalently applied object-oriented mechanism.

5 The complexity of polymorphism

This section analyzes the complexity of polymorphism, that is, how large polymorphic compared to non-polymorphic methods are (research question 3), and how scattered and distributed polymorphic methods are (research question 4).

5.1 Complexity in terms of size

First, we investigate research question 3 by looking at the size of polymorphic methods. We count the number of lines of code and the number of message sends (to public and private methods) polymorphic methods contain and compare them to non-polymorphic methods. Figure 3 illustrates that non-polymorphic methods invoke slightly more methods than polymorphic methods, in both Smalltalk

[5] Surprisingly, the proportions of methods sending a polymorphic message is lower than the proportion of call sites; we hypothesize that sends cluster in methods.

and Java projects. For Smalltalk, non-polymorphic methods send on average one message more (3 message sends compared to 2 in polymorphic methods). The first quartiles are both at 1 message send, while the third quartile is at 6 message sends for non-polymorphic methods (arithmetic mean: 4.54 message sends) and at 5 for polymorphic methods (arithmetic mean: 3.72 message sends). In Java, the median of number of message sends is the same (one message send) for both polymorphic and non-polymorphic methods; the third quartiles however differ with 5 message sends for non-polymorphic (arithmetic mean: 4.15 message sends) and 3 for polymorphic methods (arithmetic mean: 2.88 message sends); the first quartiles are both at 0 message sends.

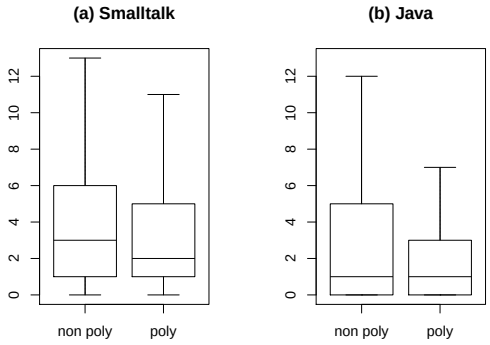

Fig. 3. Distribution of size of methods (by number of message sends), contrasting non-polymorphic with polymorphic methods, in (a) Smalltalk, (b) Java

As using the number of lines of code metric as a proxy for size gives nearly the same results as for number of sends, we omit the details of this analysis due to space reasons.

For both Smalltalk and Java, a Mann-Whitney test turns out to be statistically significant ($p < 10^{-38}$): non-polymorphic methods do tend to be longer than methods implementing a polymorphic message. However, the effect sizes are quite small (A_{12} of 0.44 for Smalltalk, and $d = 0.16$; for Java, $A_{12} = 0.43$, and $d = 0.16$ as well). The values of effect size of d are a little below the common threshold for a small effect size. All in all, if polymorphic methods are on average smaller than other methods, the effect is not very pronounced.

5.2 Complexity in terms of scattering

To study the complexity of polymorphism in terms of polymorphic behavior scattered over class hierarchies (research question 4), we extracted the proportion of classes in a hierarchy implementing a polymorphic message and the proportion of classes responding to the message.[6] These figures give us two data points:

[6] Subclasses of classes implementing the polymorphic message respond to it even when not providing their own implementation.

First, what is the average effort to add a new message (do we usually need to provide an implementation for all the subclasses, or a portion only?), and second, how often can we expect that the entire class hierarchy can be used interchangeably (if all the classes in the hierarch respond to the same message).

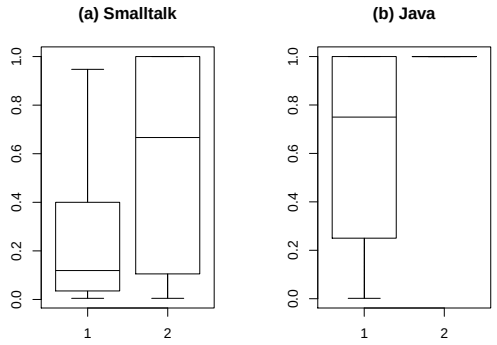

Fig. 4. Distributions of proportion of classes implementing a message in a hierarchy (1) and of classes responding to a given message (2), in (a) Smalltalk, (b) Java

The results for the proportion of classes responding to or implementing a polymorphic message in a hierarchy differ considerably between Smalltalk and Java. A glance at Figure 4 is enough to conclude that statistical tests are not necessary (and omitted for space reason).

For Smalltalk projects, the median proportion is 0.12 (first quartile at 0.04, third quartile at 0.40, arithmetic mean at 0.26), while Java projects have a median proportion of 0.75 (first quartile at 0.25, third quartile at 1.00, arithmetic mean at 0.64). Polymorphic hierarchies in Java have a higher proportion of classes implementing a polymorphic message and are hence much more covered with polymorphic implementation of a given message than in Smalltalk, the proportion medians differ by more than 0.6.

We credit this to the fact that Java hierarchies are in general smaller (27 classes on average compared to 48 in Smalltalk) and therefore Smalltalk developers have a higher tendency to implement generic behavior high in the hierarchy while Java hierarchies encompass more specialized, non-generic behavior in its classes implementing polymorphism. Another factor is the presence of Java Interfaces: since a class implementing an interface must provide an implementation for all its method, it will by definition redefine all the messages in the interface.

In Smalltalk, on average 57% of all classes in a polymorphic hierarchy respond to its polymorphic message (median at 0.66, first quartile at 0.11, third quartile at 1.00). This percentage is with 83% much higher for Java (median, first quartile, and third quartile at 1.00—hence the bar in the boxplot). A possible reason for the high proportion of classes responding to a message in Java is its static type system, preventing compilation of programs in which subclasses do not fulfill their contract, even if they will never be sent the message dynamically. On the

other hand, Smalltalk programmers do not need to provide an implementation for every class in the same situation. This is also a reason for the higher proportion of implementations for Java.

Summary: To summarize the findings in this section concerning the research question of how scattered polymorphism is over the hierarchy, we observe that in Smalltalk polymorphic behavior is often implemented high in the hierarchy, far away from leaf classes while in Java, polymorphic behavior is distributed more over the entire hierarchy. Both ways of implementing polymorphism can be complex to understand, since in Smalltalk behavior implemented in superclasses can affect in a non-obvious way leaf classes in the hierarchy, while in Java, the more scattered nature of polymorphic methods can also be hard to understand.

6 Threats to Validity

Construct Validity. The best way to detect polymorphic methods would have been the usage of both static and dynamic analysis. However, due to the impracticality of performing dynamic analysis on such a large number of projects, we have only used static 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. Other threats 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 in 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 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 is 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-corpuses 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.

7 Conclusions

We performed an empirical study of polymorphism on a large corpus of software systems, featuring two programming languages, Smalltalk and Java. We found that in both languages, polymorphism is frequently used: Nearly all projects we analyzed take advantage of polymorphism by implementing polymorphic messages and by sending such messages. However, it turned out that Smalltalk uses polymorphism to a much greater extent; more than 60% of all classes implement a polymorphic message in Smalltalk projects, while around 40% do the same in Java projects (for methods: 31% in Smalltalk compared to 24% in Java).

Analyzing the complexity of polymorphic methods reveals that non-polymorphic methods are in general larger: They contain on average one message send more than polymorphic methods, for both Smalltalk and Java. The results show that polymorphic methods are often scattered over the entire class hierarchy: While in Smalltalk on average only 26% of all hierarchy classes implement a polymorphic message, 57% respond to it. For Java these percentages are with 64% and 83%, respectively, much higher, which is likely due to Java being statically-typed.

References

1. Andrea Arcuri and Lionel 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. Michelle Cartwright. An empirical view of inheritance. *Information Software Technology*, 40(14):795–799, 1998.
3. John Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1:109–132, 1996.
4. Serge Demeyer, Stéphane Ducasse, Kim Mens, Adrian Trifu, and Rajesh 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.
5. Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Moose: An agile reengineering environment. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, pages 99–102, 2005.
6. Alastair Dunsmore, Marc Roper, and Murray 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.
7. Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo 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.
8. 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.

9. Graylin Jay, Joanne E. Hale, Randy K. Smith, David P. Hale, Nicholas A. Kraft, and Charles Ward. Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. *JSEA*, 2(3):137–143, 2009.
10. Chris Parnin, Christian Bird, and Emerson 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.
11. Daryl Posnett, Vladimir Filkov, and Premkumar 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.
12. Romain Robbes and Mircea Lungu. A study of ripple effects in software ecosystems. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 904–907, 2011.
13. David Röthlisberger, Marcel Härry, Alex Villazón, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz, and Philippe Moret. Exploiting dynamic information in ides improves speed and correctness of software maintenance tasks. *Transactions on Software Engineering*, 2012.
14. Ewan Tempero, Steve Counsell, and James Noble. An empirical study of overriding in open source java. In *Proceedings of the Thirty-Third Australasian Conference on Computer Science (ACSC 2010)*, pages 3–12. Australian Computer Society, Inc., 2010.
15. Ewan Tempero, James Noble, and Hayden 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.
16. 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.