Changes, Defects and Polymorphism: is there any Correlation?

Petru Florin Mihancea
LOOSE Research Group
"Politehnica" University of Timişoara, Romania
Email: petru.mihancea@cs.upt.ro

Cristina Marinescu
West University of Timişoara, and
"Politehnica" University of Timişoara, Romania
Email: cristina.marinescu@cs.upt.ro

***Abstract*—Abstract source code is better than concrete code when it comes to reduced change likelihood. Trying to provide empirical evidence for this assumption, we investigate if there are any correlations between the usage of partially polymorphic calls in classes and their change and defect likelihood. Based on the provided evidence, practitioners are advised about the potential impact of polymorphism at the source code level.**[1] [2]

I. Introduction

Subtype polymorphism and class hierarchies are distinguishing mechanisms of object-oriented technology. On one hand, the main benefit of employing these mechanisms is to reduce the source code changes due to evolving requirements. This is emphasized by the plethora of design principles, heuristics and patterns related to polymorphism (*e.g.*, [1]). On the other hand, the dark side of polymorphism is that it makes an application harder to understand (*e.g.*, *the yo-yo effect*) and can introduce subtle defects into a program due to unpredicted polymorphic interactions between objects (*e.g.*, [1], [2]).

Much effort focusing both the benefit and the drawback of polymorphism has been spent by the research community. Different analysis techniques have been proposed to comprehend, assess and restructure the design of an object-oriented application in order to take advantage of the flexibility given by polymorphism (*e.g.*, [3], [4]). In spite of all these achievements, to the best of our knowledge, there is no work trying to correlate the change and defect proneness of classes with polymorphism usage. Consequently, we have performed an experiment trying to answer the following questions: (RQ1) – Is the intensity of the polymorphic calls from a class correlated with its change likelihood? (RQ2) – Is the intensity of the polymorphic calls from a class correlated with its defect likelihood?

The paper firstly presents the way we measure the polymorphic interactions in a class (Section II). The details of our experiment and the results we obtained are presented in Section III. In the successive section we relate our empirical study with existing works and we summarize the conclusions and address some hints towards the future work in Section V.

[1]The scientific contribution of the two authors is equal.

[2]The work of Cristina Marinescu is partially supported by the European Union under FP7-REGPOT-CT-2011-284595-HOST.

II. Measuring Polymorphic Interactions

We introduce the *Invocation Generality (IG)* metric for measuring the intensity of the polymorphic interactions between classes. The metric is a particularization of a more general metric called *Level of Abstraction (LA)* we introduced in [4].

The aim of the *LA* metric is to measure the degree of abstraction of a statement / expression from a method due to the usage of polymorphism. For each statement, the metric is defined with respect to i) a base class whose instance services are invoked by the method and ii) a reference variable through which the method accesses the base class hierarchy. In essence, the *LA* value for a statement is proportional with the number of concrete classes from the hierarchy whose objects may be referred by the variable before the execution of that statement.

By contrast, the *IG* metric is a particularization of the *LA* measure because:

- The metric is defined only for *invocation statements* that could be dispatched dynamically (*i.e.*, virtual calls).
- For a particular invocation, the value of the metric is computed only with respect to its target reference. The reason is that in this paper, we are interested to measure only the degree of abstraction due to polymorphism of the invocation alone and not of the entire code of the method containing the invocation.
- For a particular invocation, the *IG* metric is computed with respect to the highest base class (*i.e.*, having the maximum height in the inheritance tree) declaring the invoked service. The reason is that an invocation has a maximum degree of generality due to polymorphism when it may target instances of any class having the invoked service in its interface.

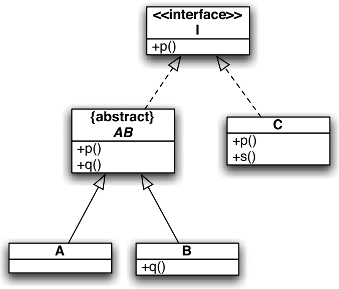
Providing this, the *IG* metric is computed using the following formula. *N* represents the number of concrete classes the target reference may refer to at the invocation site while *M* represents the number of concrete classes from the hierarchy rooted by the highest class declaring the invoked service.

$$IG(inv) = \begin{cases} undefined \leftrightarrow The\ call\ is\ not\ virtual \\ e.g., the\ called\ method\ is\ static \\ 0 \leftrightarrow N = 1 \\ (N-1)/(M-1) \leftrightarrow N > 1 \end{cases}$$

More details on the *LA* metric can be found in [4] and a brief

```
void client1(I t) {
    1: t.p();
}
void client2(AB t)
    2: t.p();
    3: t.q();
}
void client3(C t) {
    4: t.p();
    5: t.s();
}
```

(a) Invocation Examples



(b) The Class Hierarchy

Fig. 1. *IG* Metric Examples

comparison of *IG* with related metrics (*e.g.*, *DIT*) is presented in Section IV.

a) Interpretation and Examples: When defined, the value of the *IG* metric is between 0 and 1. Based on Figure 1 we exemplify and interpret several values of the metric.

A value of 0 means that the invocation, although virtual, does not actually make use of polymorphism. In other words, the target object is always an instance of a single class from the hierarchy. The invocations from line 4 and 5 in Figure 1a exemplify this kind of method calls, since *t* can refer only to instances of class *C*. We emphasize that this value for the *IG* metric also appears when the class containing the invoked method is a "degenerated base class" (*i.e.*, it is not included in an inheritance lattice).

A value of 1 means that the measured call makes full use of polymorphism. For instance, the invocations from lines 1 and 3 use polymorphism at the highest possible level of generality. That is because, the *t* variable in *client1* may refer to instances of any class from the hierarchy. For the *t* variable in *client2*, we can remark that it may refer only to instances of type *A* or *B*. However, the call in line 3 is entirely general because the *q* method is specific only to objects of these concrete types.

An intermediate value for the *IG* metric indicates that the measured invocation does not entirely make use of polymorphism. For instance, the call in line 2 can only target *A* or *B* instances since *t* is of type *AB*. Additionally, *p* represents a general service that can be executed on objects of any class from the hierarchy (including *C*). Consequently, the *IG* metric for this invocation is 0.5.

b) Implementation: The main concern in implementing the *IG* metric is to detect the set of concrete classes a target reference may refer to at run-time at an invocation site. This information is necessary because based on it we determine the *N* component from the *IG* metric formula.

Unfortunately, computing this set (and consequently the *IG* metric) with a high accuracy is a difficult task. As an extreme example, some subclasses may be unknown at compile-time and could be written and loaded dynamically. In this case, we cannot exactly know neither the *M* component of the *IG* formula. For the sake of simplicity and scalability, in this work we have estimated the *IG* metric in the following manner: we consider that the set of concrete classes a target reference may refer to at run-time is the set of all concrete descendants of the reference declared class. If this class is concrete, it is also included in the set. As discussed in Section III-D, this approximation can negatively affect the validity of our experiment. However, more advanced analyses can be employed in the future for a more accurate (but less scalable) computation of the *IG* metric.

III. Experiment

A. Data Collection

In this experiment we have analysed several versions of three open-source systems whose main characteristics are presented in Table I. According to the Overview Pyramid [5], *ArgoUML* and *FOP* have deep and wide hierarchies and, by contrast, we also considered *FindBugs* because, according to the same pyramid, it makes less use of inheritance.

For each class we have extracted the changes and defects between two releases using iProblems [6]. For this purpose the tool makes use of version systems' logs and information provided by bug tracking systems. We have also extracted an aggregation of the *IG* metric at the class level, namely *MIN_IG*. It represents the minimum value of the *IG* metric computed for all the invocations in the measured class. Choosing the minimum value of this metric is in line with previous studies where subjects were classified into one or another category taking into account that they expose *at least* a characteristic's type [7]. The aggregated metric is undefined when the measured class does not contain any virtual invocation. The *IG* metric and its aggregation have been implemented and computed using the CodePro/Patrools [8] Eclipse plugin.

System	Referred Version	Start Date Archive	End Date Archive	LOC	Types
ArgoUML	1	30/11/2003	30/11/2004	83,487	1180
	2	01/12/2004	09/02/2006	107,125	1237
	3	10/02/2006	13/02/2007	155,223	1476
	4	14/02/2007	27/09/2008	144,075	1550
FindBugs	1	31/05/2006	31/05/2007	52,206	635
	2	06/01/2007	06/05/2008	73,484	791
	3	05/07/2008	05/08/2009	84,638	931
	4	08/06/2009	30/11/2010	98,082	1022
FOP	1	26/03/2008	31/07/2008	89,398	933
	2	01/08/2008	02/08/2009	97,397	1089
	3	03/08/2009	25/12/2010	120,255	1457

TABLE I
Some Characteristics of the Analyzed Systems.

B. Conducting the Study

In order to answer the mentioned research questions we have employed the Chi-Square (χ^2) test. This test, as it is presented in [9], evaluates if within the underlying population represented by the sample in a contingency table (rxc), the two involved dimensions are independent of one another.

The structure of the first contingency table created for answering RQ1 consists of two dimensions: MIN Invocation

Generality (*MIN_IG*) and Changes. The *MIN_IG* dimension is the row dimension and we consider six categories that compose this dimension: undef. – a class does not have virtual calls, 0 – a class uses directly at least a concrete type, (0 ; 0.5), 0.5, (0.5 ; 1) and 1 for classes that use only fully polymorphic calls. The two categories which composed the Changes dimension (*i.e.*, dependent variable) are: Have Changes – classes reveal at least a change and Do not have Changes – classes do not reveal changes.

In a similar manner we define another contingency table for answering RQ2, the row dimension being the same as in the previous case, while the column dimension is comprised of defects (*i.e.*, Have Defects and Do not have Defects).

C. Results

For each of the contingency tables we compute the values of the χ^2 test as well as p-values using the R Project for Statistical Computing[3]. If p-value is less than a 0.05 level of significance, we have enough evidence to consider that the two dimensions of the contingency table are not independent. Next, based on the values of the observed and expected frequencies (computed as a result of the test) of the contingency tables, we establish the way (positive or negative) in which the involved dimensions are correlated. A trait from the row dimension is positively correlated with a trait from the column dimension if the observed frequency is greater than the expected frequency. Based on this fact we discuss below each type of the inferred correlation.

(RQ1) Polymorphism and Changes. We summarize the obtained results in Table II, where + denotes a positive correlation between a MIN_IG values belonging to the set {undef., 0, (0;0.5), 0.5, (0.5;1), 1} and changes, - denotes a negative correlation, space denotes no existing correlation (p-value greater than 0.05) and NA denotes a situation when the χ^2 cannot be computed. From Table II we can see that:

- in most of the cases the employed test reveals different types of correlations between the two dimensions.
- all the inferred correlations between classes containing at least one call that always targets a particular concrete type (*i.e.*, *MIN_IG = 0*) and changes are positive. This means that these classes are more likely to be changed.
- usually, classes using partially polymorphic calls are more likely to be changed (*e.g.*, for *MIN_IG* = 0.5, out of 8 analyzed systems, 6 reveal a positive correlation).
- most of the times, classes using fully polymorphic calls are less likely to be changed (*i.e.*, 6 out of 8 correlations are negative for *MIN_IG = 1*).

All these observations conform to the expected benefit of using polymorphism in object-oriented systems: usually, when a class makes full use of polymorphism, the class is less likely to be changed. The single two situations when this is not the case appear in *ArgoUML* versions *3* and *4*. These situations might be caused by some design flaws or defects related to some class hierarchies. Consequently, it would worth taking a closer look at the inheritance latices from *ArgoUML* and at their fully polymorphic – but changed – clients. By contrast, classes that do not make full use of polymorphism, proved to be more change prone. This is also in line with the object-oriented theory and emphasizes again the importance of polymorphism related design principles.

		undef.	0	(0;0.5)	0.5	(0.5;1)	1
ArgoUML	1	NA	NA	NA	NA	NA	NA
	2	-	+	-	+	+	-
	3	-	+	+	-	+	+
	4	-	+	+	+	-	+
FindBugs	1	-	+	-	+	+	-
	2	-	+	+	+	+	-
	3	-	+	-	-	+	-
	4						
FOP	1						
	2	-	+	-	+	-	-
	3	-	+	+	+	-	-

TABLE II
CORRELATIONS BETWEEN CHANGES AND MIN_IG.

(RQ2) Polymorphism and Defects. From Table III that summarizes the results related to possible correlations between polymorphic calls and defects we can see that:

- *FindBugs* shows no correlations between the two involved dimensions, probably due to its less usage of inheritance.
- all the inferred correlations between defects and classes containing at least one call that always targets a particular concrete type (*i.e.*, *MIN_IG = 0*) are positive. This means that the involved classes are more defect prone.
- the greater the intensity of polymorphism usage in a class, the lower the likelihood for the class to exhibit defects. For instance, when the *MIN_IG* value is higher than 0.5, 6 out of 10 correlations are negative.
- the lower the polymorphism usage, the higher likelihood of classes to contain defects. For instance, 12 out of 15 correlations are positive when *MIN_IG* is lower or equal to 0.5.

Based on these observations we can conclude that, usually, classes using high polymorphism constructs have smaller chances of exhibiting defects than the other classes. The positive correlations for full usage of polymorphism appear again in version 3 and 4 of *ArgoUML*. As a result, it seems that the changes in these versions are actually caused by defect fixings. This emphasizes again that it would worth to take a closer look at the *ArgoUML* hierarchies and at their clients. By contrast, when polymorphism is not or only partially used, the defect likelihood increases. Again, it seems that the changes from *ArgoUML* and *FOP* are predominantly caused by defect fixings.

Classes with undefined MIN_IG may not use external services at all and, consequently, this may be the cause for both of the the negative correlations from Tables II and III.

D. Threats to Validity

Within the presented case study the construct validity threats are mainly related to the errors performed during the data

[3]http://www.r-project.org/

		undef.	0	(0;0.5)	0.5	(0.5;1)	1
ArgoUML	1	NA	NA	NA	NA	NA	NA
	2	-	+	-	+	+	-
	3	-	+	+	-	+	+
	4	-	+	+	+	-	+
FindBugs	1						
	2						
	3						
	4						
FOP	1						
	2	-	+	-	+	-	-
	3	-	+	+	+	-	-

TABLE III
CORRELATIONS BETWEEN DEFECTS AND MIN_IG.

extraction. We consider that classes, changes and defects were extracted by IPROBLEMS [6] with a high precision and recall using probably the most widespread approach.

Another construct validity threat appears due to the way we approximate the *IG* metric. Because our implementation is based on pure static information there can be cases in which we incorrectly classify an invocation as making full or partial use of polymorphism. For instance, the execution of an invocation via a target reference declared as being of a super-type type can be guarded by an instanceof condition checking the compliance of the target object to some particular sub-type. These cases are not captured in the current approximation of the *IG* metric. To overcome this issue we can improve the current implementation of the metric using more advanced *points-to and data-flow* analyses. However, a highly possible drawback would be a serious decrease of the metric computation scalability.

We do not suggest generalising our results (external validity) unless further case studies are performed. In this context we consider we have provided enough information about our study to let it be replicated (reliability validity).

IV. RELATED WORK

On one hand, our work is related with the field of class hierarchy and polymorphism measurements. In [10] two software metrics related to class hierarchies are introduced, namely *Depth of Inheritance (DIT)* and *Number of Children*. In [5] are proposed several other metrics for class hierarchies such as *Base Class Overriding Ratio* or *Number of Added Services*. More polymorphism-specific metrics are introduced in [11] and [12]. All these metrics could offer us hints about the polymorphic capabilities of a class hierarchy. However, as is it recognized in the case of [12], none of the aforementioned metrics address the call-sites targeting a class hierarchy: they do not capture the manner in which the clients of the hierarchy really use it (*i.e.*, polymorphically or not). In contrast, the *IG* metric directly addresses this aspect by considering a relative number of concrete classes whose objects could be targeted by an invocation.

On the other hand, our work is related with the plethora of empirical studies trying to validate the capability of various metrics to emphasise defect proneness of classes. The well-known suite of metrics introduced in [10] was validated as a good defect predictor in [13]. The authors consider the CBO (Coupling Between Objects) [10] metric as being the best predictor for defects. However, i) *CBO* is not capable to distinguish between polymorphic and non-polymorphic couplings, and ii) *CBO* cannot capture the intensity of a polymorphic coupling.

To the best of our knowledge, there is little to no empirical evidence regarding the change and defect proneness of classes taking into account metrics that capture the degree of polymorphism usage within classes. As previously mentioned, in [12], several polymorphism related metrics are defined and evaluated from the early risk prediction capabilities point of view. However, due to the metrics definitions, this study does not consider the polymorphic invocation effect [12]. Due to the usage of the *IG* metric, our study takes into consideration this view of the calling source code and thus, the experimental perspectives are complementary.

V. CONCLUSIONS. FUTURE WORK

In this paper we presented an empirical study that provides evidence about a positive correlation between classes using concrete types or partial polymorphism and their change and defect proneness. We do not want to suggest that using concrete types or partially polymorphic calls is the cause of an increased likelihood to exhibit changes and defects. However, we do provide evidence these kinds of invocations are most of the times statistically correlated with changes and defects.

We intend to provide within the PROMISE data set a database that contains the values of the extracted metrics.

REFERENCES

[1] R. C. Martin, *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[2] J. Offutt, R. T. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson, "A Fault Model for Subtype Inheritance and Polymorphism," in *Proc. ISSRE*. IEEE Computer Society, 2001.

[3] S. Ducasse, S. Demeyer, and O. Nierstrasz, "Transform conditionals to polymorphism," in *Proc. EuroPLoP*, 2000.

[4] P. F. Mihancea, "Type Highlighting : A Client Driven Visual Approach for Class Hierarchies Reengineering," in *Proc. SCAM*. IEEE Computer Society, 2008.

[5] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[6] M. Codoban, C. Marinescu, and R. Marinescu, "iProblems - an integrated instrument for reporting design flaws, vulnerabilities and defects." in *Proc. WCRE, Limerick, Ireland*. IEEE Computer Society Press, 2011.

[7] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 2012.

[8] R. Marinescu, G. Ganea, and I. Verebi, "inCode: Continuous Quality Assessment and Improvement," in *Proc. CSMR*. IEEE Computer Society, 2010.

[9] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures, 4th edition*. Chapman&Hall/CRC, 2007.

[10] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Transactions on Software Engineering*, 1994.

[11] F. Brito e Abreu, M. Goulao, and R. Esteves, "Toward the Design Quality Evaluation of Object-Oriented Software Systems," in *Proc. ICSQ*, 1995.

[12] S. Benlarbi and W. L. Melo, "Polymorphism measures for early risk prediction," in *Proc. ICSE*. ACM, 1999.

[13] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, 2005.