Polymorphism Measures for Early Risk Prediction

Saïda Benlarbi
Cistel Technology
210 Colonnade Road, Suite 204
Nepean, Ontario, Canada K2E 7L5
benlarbi@cistel.com

Walcelio L. Melo
Oracle Brazil *and*
Univ. Católica de Brasilia
SQN Qd. 02- Bl A-Salas 604
Brasilia, DF Brazil 70712-900
wmelo@acm.org

Abstract. Polymorphism is a key feature of the object-oriented paradigm. However, polymorphism induces hidden forms of class dependencies, which may impact software quality. In this paper, we define and empirically investigate the quality impact of polymorphism on OO design. We define measures of two main aspects of polymorphic behaviors provided by the C++ language: polymorphism based on compile time linking decisions (overloading functions for example) and polymorphism based on run-time binding decisions (virtual functions for example). Then, we validate our measures by evaluating their impact on class fault-proneness, a software quality attribute. The results show that our measures are significant predictors of fault proneness as well as they constitute a good complement to the existing OO design measures.

Keywords: object-oriented design measurement, Polymorphism, overloading, C++ programming language, software risk prediction.

1. Introduction

Object-Oriented (OO) design is common practice in the software engineering field; C++ is one of the most used OO programming language [14] [16]. OO design is based on *abstraction, encapsulation, modularity* and *inheritance* [7]. These concepts are often translated into different *design* and *coding* mechanisms in programming languages which prescribe decomposition, encapsulation and visibility in a software system [14].

Mainly, OO techniques use inheritance, association, aggregation, polymorphism and message passing to model any static or dynamic computation feature. Polymorphism is claimed to provide benefits such as greater extensibility and reusability of OO systems [7], [10], [13], and [15]. However, some polymorphic forms, when used in combination with inheritance, can penetrate encapsulation boundaries and create hidden dependencies.

The goal of this work is to empirically evaluate the quality impact of polymorphic forms on OO software design. We aim at the definition and validation of a suite of OO design measures, which can be collected early in the product-life cycle and that, can predict software quality. While we have focused on C++ systems, our polymorphism measures can be tailored to other polymorph languages. Section 2 discusses and relates our study to currently developed research work. Section 3 first presents the different polymorphic forms we have identified in OO designed systems, and then details our suite of OO polymorphism measures. Section 4 addresses the empirical evaluation of our OO polymorphism measures. Finally, Section 5 concludes the paper by presenting lessons learned and future prospects.

2. Related Work

Several software OO metrics have been proposed in the last years [1][6][9][11][12]. We will focus here only on design metrics that can be evaluated early in the software life cycle. Chidamber and Kemerer [11] have proposed a set of coupling metrics (CK metrics in the rest of the text) which have been empirically validated by Basili, Briand and Melo [2]. The CK measures address visible class interactions forms regardless of the hidden effect of specific elements involved in an interaction. Briand, Devanbu and Melo [9] have investigated coupling metrics that take into account of C++ specific features (e.g. friendship mechanism. Three main views have been considered along which several coupling measures have been defined: *locus,* the expected coupling impact location (import/export); *type,* the kind of the involved items in an interaction (attribute or method); and *relationship,* which differentiates the inheritance and friendship mechanisms. This set of metrics has been extended by Benlarbi and Melo [3] with a fourth view along which class coupling may occur: *scope* or access level.

In our knowledge, few of the proposed OO design metrics has focused on polymorphism. In particular, Abreu and Carapuca [1] have considered a polymorphism factor (POF) in their set of design metrics MOOD. The POF measure quantifies the percentage of the overridden methods across an inheritance lineage in a OO system (member functions in C++). However, the POF number gauges only one visible polymorphic form: the relative number of overridden methods in a descending inheritance line. The POF measure stems from the assumption that a message sent to a class is eventually bound (statically or dynamically) to one of its method names or one of its descendant method names. Polymorphic behaviors in OO systems include other forms discussed in the next section.

3. Polymorphism Metrics for OO Design

3.1 Forms of Polymorphism in OO design

In the OO paradigm, polymorphism stems from the combination of message passing, inheritance and substitutability that allow implementing the *is-a* relationship. This combination may yield to different techniques used in OO programming languages to achieve code sharing and reuse. There is no consensus on the polymorphism terminology in the programming languages community. Mainly, we can cite five different definitions for polymorphism in the OO paradigm: pure polymorphism, overriding, deferred methods, overloading and generics. Note that generics may stand for polymorphic methods as well as for polymorphic classes (or types). *Template* functions are an example of polymorphic methods in C++.

In the current work, we will consider only three of these definitions: pure polymorphism, overriding and overloading. *Pure polymorphism* is achieved by applying a single method to arguments of different types. In this case the same code may have several interpretations. *Method overriding* is achieved when the behavior described in a parent class is altered in the descendant class. *Overloading* or *ad hoc polymorphism* is achieved when the same name denotes different methods (different code).

Here, we will restrict ourselves to polymorphic forms in C++ that respond to one of the three previous definitions, namely polymorphic member functions. Consider a member function as formally defined by a tuple *<Name, Signature, Return Type>;* each of these three characteristics may change in a new declaration while the others may remain fixed. This eventually generates different forms of polymorphic behaviors that may influence the quality of a OO design developed in C++.

Depending on the encapsulation level considered, different forms of polymorphic member functions may occur in a C++ design: within the class boundary (elementary level) and within a class hierarchy encapsulated in a system (composed level). At the class level, polymorphism is almost synonymous with overloading; at the composed level two forms of polymorphic behavior may occur: pure polymorphism or overriding.

Our classification of polymorphic behaviors in C++ includes polymorphic behaviors that are based on *run-time binding decisions* (e.g., virtual functions) as well as on *compile time linking decisions* (e.g, overloading functions). It identifies visible and hidden polymorphic forms of a method in a C++ design. From this categorization we derived measures that account for different types of polymorphic behaviors, and thus evaluate and quantify the impact of each type of behavior on the quality of a given artifact. Mainly, we consider three polymorphic forms for a C++ member function:

- **Pure Polymorphism:** this capability also called *parametric overloading*, is provided in C++ by the evocation of the *same name* with *different signatures* inside the class scope. Pure polymorphism within a class is implemented by creating several methods with different signatures. Based on static binding, a parametric overloaded member function is recognized by the number, types and order of the arguments in an invocation.

- **Static polymorphism**: In C++, different functions having the same name but with different signatures can be defined in different classes linked or not by inheritance relations. This corresponds to *method overriding*. In C++ two different forms of overridden methods may occur: non-virtual methods and virtual methods. Using compile-time decisions, non-virtual overridden methods are recognized by the different signatures they hold in different declarations. We call this form *static polymorphism* because they are based on static bindings.

- **Dynamic polymorphism**: It is the ability to use the *same name* and the *same signature* in an overridden method. It corresponds to the implementation of *ad hoc polymorphism* in C++. We call it dynamic because run-time decisions are used by the compiler to recognize the right method evoked.

Static and dynamic polymorphism result from the OO design features of *combination and specialization*. The forms of polymorphism defined above constitute one view under which a design may be scrutinized, and which may be combined to others such as the class inheritance relations view or the class friendship relations view. We distinguish the following types of inheritance relations: no inheritance, simple inheritance or multiple inheritance. In the present work we do not consider multiple inheritance or friendship relations. Combining static/dynamic polymorphism forms with simple inheritance relationships, we derive the set of measures shown in Table 1.

Table 1. List of metrics for polymorphism forms

Metrics	Definition
OVO	Overloading in stand-alone classes
SPA	Static polymorphism in ancestors
SPD	Static polymorphism in descendants
DPA	Dynamic polymorphism in ancestors
DPD	Dynamic polymorphism in descendants

In object orientation, generic methods or classes can minimize the introduction of new methods and objects. The OVO measure is intended to gauge the degree of methods genericity in a class by counting the number of function members that implement the same operation. For example, if an add operation is implemented in a class C, via a set of C++ function members each one evoked by a different parameter type list (e.g., integer, float, short), OVO counts the total number of functions that are used in C to implement the add operation. Static and dynamic polymorphism measures are intended to gauge separately the impact of static binding and dynamic binding on a OO design.

Polymorphism may introduce hidden forms of class dependencies that break modularity and thus impact the design quality of an artifact. Combining the static and dynamic polymorphism forms with the inheritance relationships we are able to evaluate the impact of changes made to polymorphic behaviors coming from inherited classes on a given class C. We can thus predict how much class C will be impacted if any modification is made in its ancestors or descendants polymorphic member functions. This is different from the POF measure from the MOOD set of metrics in that it gives more precision on the quality impact of each specific OO feature on a given design. For example, one can evaluate the quality impact of static binding and dynamic binding separately.

3.2 Polymorphic Measures for Object-Oriented Design

In this section, we provide formal definitions for the metrics presented in Table 1.

Figure 1. Example of C++ Polymorphism cases

```
Class A {
        virtual void m (int   i);
        virtual void m (float   f);
        virtual void m (int    i , int   f);
        virtual void p ();
        virtual void p (int   x);    }

Class  B : public A {
          virtual void m (); }

Class C : public B {
        virtual void m  ();
        virtual void p ();     }

Class D {
        virtual void m ();   }
```

For the current study we consider a OO design as a collection of classes.

- **Overloading**: The OVO measure is given by:

$$OVO(C) = \sum_{f_i \in C} overl(f_i, C) \qquad \textbf{Eq. 2}$$

where *overl*(f_i , C) is an operator which returns the number of times the function member name f_i is overloaded in the class **C**. For instance, from Figure1 *overl(m, A)*=3 and *overl(p, A)* =2, therefore *OVO(A)* = 5.

- **Static polymorphism**:

SPoly(C_i ,C) is an operator which returns the number of static polymorphism function members that appear in C_i and *C*. Note that the static polymorphism relation is symmetrical, *i.e*. for any pair of classes C_i and *C*, *SPoly*(C_i ,C) = *SPoly*(C_i , C). For instance in Figure 1, *A* and *B* has one polymorph function member m, so *Spoly(B,A)= Spoly(A,B)=1.*

Static polymorphism in ancestors is given by:

$$SPA(C) = \sum_{C_i \in Ancestors} SPoly(C_i, C)$$

Eq. 1

where *ancestors(C)* is an operator that returns the set of distinct ancestors of class C. For instance, from the pair of lines: 2-10 or 3-10 or 4-10 we have SPA(**B**)=1.

Static polymorphism in descendants is given by:

$$SPD(C) = \sum_{C_i Descendents} SPoly(C_i, C) \qquad \textbf{Eq. 3}$$

where *Descendents(*C*)* is an operator that returns the set of distinct descendents of class C. For instance, from Figure 1 in the pair of lines 7-14, we have SPoly(A,B)=1 and SPoly(A,C)=1, then SPD(A)=2.

- **Dynamic polymorphism**:

DPoly(C_i ,C) is an operator which returns the number of dynamic polymorphism function members that appear in C_i and *C.* Again, we can note that dynamic polymorphism relations are symmetrical, i.e. for any pair of classes C_i and *C,* $DPoly(C_i, C) = DPoly(C_i, C)$.

Dynamic polymorphism in ancestors is given by:

$$DPA(C) = \sum_{C_i \in Ancestors} DPoly(C_i, C) \qquad \textbf{Eq. 4}$$

For instance from Figure 1 in lines 6, 10, 13 and 14, we have *DPoly(B,A)=0* and *DPoly(C,A)=1* and *DPoly(C,B)=1.*Then, *DPA(B)=0* and *DPA(C)=2.*

Dynamic polymorphism in descendents is given by

$$DPD(C) = \sum_{C_i \in Descendents} DPoly(C_i, C) \qquad \textbf{Eq. 4}$$

For instance, from in lines 6, 10, 13 and 14, we have *DPoly(A,B)=0* and *DPoly(A,C)=1* and *DPoly(B,C)=1.* Then, *DPD(A)=1* and *DPD(B)=1.*

The last four measures can be aggregated in order to capture the level of polymorphism in a class pertaining to a class hierarchy. The aggregate polymorphism measures are:

- Static polymorphism in inheritance relations:

$$SP(C) = \sum_{C_i \in (Ancestors \bigcup Descendents)(C)} SPoly(C_i, C) \qquad \textbf{Eq. 5}$$

- Dynamic polymorphism in inheritance relations:

$$DP(C) = \sum_{C_i \in (Ancestors \bigcup Descendents)(C)} DPoly(C_i, C) \qquad \textbf{Eq. 6}$$

Relations than inheritance can occur in combination with polymorphism in C++ systems. Methods with the same name may appear in classes that have no inheritance relations. While such methods may be conceptually and computationally unrelated, they may lead as well to maintenance difficulties: developers reading the code may get confused. Thus, for example, for two unrelated classes C_1 and C_2 having the same function member ***f*** appearing at unrelated places in the source code may cause confusion for the maintainer. Polymorphism in non-inheritance relations is formally defined by:

Eq. 8

$$NIP(C) = \sum_{C_i \in Others(C)} SPoly(C_i, C) + DPoly(C_i, C)$$

where *Others(C)* is an operator that returns the set of distinct classes that are neither ancestors nor descendants of class *C.* For instance, from Figure 1 in lines 10 and 17, we have *NIP(D)=3,* since *DPoly(D,C)=1, DPoly(D,B)=1,* and *SPoly(D,A)=1.* We note here that this is not actual polymorphism--just potential for human confusion.

4 Empirical Validation of Polymorphism Measures

In this section, we present the empirical validation of our polymorphism measures. We use the product metric validation methodology proposed in [8] which has been successfully used to validate other suites of OO design measures, e.g. [2][9].

4.1 Validation Data

Our goal here is to validate the ability of our polymorphism measures to predict fault-prone classes. We have used data collected from an open multi-agent system's development environment, called LALO. This system has been developed and maintained since 1993 at CRIM; it includes 85 C++ classes with approximately 40K source lines of code (SLOC).

We collected: (1) the source code of the C++ system, (2) data about its classes, (3) fault data. The fault data collected report concrete manifestations of errors found by the 50 beta-testers of LALO on the versions 1.1. and 1.1.a, respectively delivered on November 1996 and January 1997. The data for the polymorphism measures were collected from the source code by a tool set comprising a GEN++ [18] analyzer and a C program. The polymorphism measures are calculated by simple static analysis of only the C++ class interfaces.

4.2 Validation Strategy

To validate our OO design measures as quality indicators, we use a binary dependent variable, which capture the fault-proneness of classes: was a fault detected in a class due to an operational failure? As in [2][9], we have used logistic regression to analyze the relationship between our suite of measures and class fault-proneness.

A multivariate logistic regression model is based on the following relationship equation (the univariate logistic regression model is based on just one variable):

$$p(X_1,...,X_n) = \frac{e^{(C_0+C_1X_1+...+C_nX_n)}}{1+e^{(C_0+C_1X_1+...+C_nX_n)}} \qquad \textbf{Eq. 7}$$

where p is the probability that a fault does not occur in a class during the software operation given the polymorphism measures taken as explanatory variables in the model; and the C_i's are the regression coefficients to be estimated. For each measure, we provide the estimated regression coefficient C_i, the relative odds ratio given by:

$$\Delta\Psi = \Psi(X+1)/\Psi(X) \qquad \textbf{Eq. 8}$$

and the statistical significance coefficients (p-values) (see [17] for more details). The relative odds ratio is the ratio between the probability of having a fault when the measure has a value X and the probability of having a fault when the measure value is X + 1. Here the odds ratio indicates how the risk of having a fault in a class changes with the polymorphism measure values. In the current study we use $\Delta\Psi$ to evaluate how the risk of the predicted quality factor would change with each of our polymorphism measures.

The statistical tool we have used performs the regression analysis with the best fit according to the chi-square test (χ^2) (a non-parametric test). Chi-square provides a simple test based on the difference between observed and expected frequencies with few assumptions about the underlying data set. Large values of χ^2 indicate a large deviation from the tested hypothesis thus little credibility. In our study the null hypothesis is that a linear correlation exists between the probability of having no fault in a class and the considered OO measures.

4.2.1 Univariate Analysis

Descriptive Statistics

Table 13 shows the descriptive statistics of the polymorphism measures listed in Table 1 and defined in Section 3.2. Many measures show a limited variance in our data set. For instance, the measures of polymorphism in descendants SPD and DPD present a very week variance. This is due to the fact that LALO classes have low inheritance tree depth. This confirms other studies, which have found that the use of inheritance is low in domain-specific applications and high in class libraries [2][6]. Low use of inheritance in LALO classes can explain the weak distribution of polymorph forms in descendent relations. As a consequence, at least in our data set, these two measures, *per se*, are not likely to be useful predictors in this study. This issue is discussed in the next section.

Univariate Logistic Regression

Table 2 shows the results of univariate logistic regression of the polymorphism measures defined in Section 3.1. The following measures present a significant reverse linear relationship with π the probability of having non fault: DPA, SP and DP. All of these are statistically significant: they have p-values below the 0.05 threshold. Regarding $\Delta\Psi$, which evaluates our measures impact on the relative probability of not having a fault, these *3* measures present high values. For instance, there is a decrease of 59% and 95% of the odds ratio when DPA and SP respectively increases by one unit. This means that the probability of error increases by 59% and 95% when the DPA and SP measures increase by one unit. In our data set, the impact of static and dynamic polymorphism on the probability of having a fault is quite high when these measures increase by one unit.

From the coefficients of the significant measures, i.e. DPA, SP and DP, we can conclude that OO polymorphic design tends to increase the probability of fault-proneness in classes. This was expected since, as we have stated it in Section 3.1, polymorphism introduces hidden forms of coupling. Our results need to be corroborated by further studies, but our preliminary results suggest that OO polymorphic design tend to increase the chances of having faults in classes. This might reverse the claim that polymorphism fosters better maintainability.

Table 2 Univariate Results for the Polymorphism Measures

Measure	Coefficient	p-value	ΔΨ
OVO	-0.0262	0.7525	97%
SPA	-0.0782	0.1073	92%
SPD	-0.0298	0.1210	97%
DPA	-0.5288	**0.0035**	59%
DPD	-0.08216	0.2600	92%
SP	-0.0504	**0.0267**	95%
DP	-0.2330	**0.0164**	79%
NIP	0.00469	0.7418	100%

In our data set, the SPD, DPD, and NIP measures present a week distribution. So we cannot empirically validate the ability of these measures to predict class defectiveness. Their level of variability in other systems needs to be determined. On the other hand, from our data set at least, the OVO measure does not show any evident linear impact on probability of having (non) fault though it is not correlated to any of the other polymorphism measures. This suggests that OVO is capturing a different dimension than the other polymorphism measures. However, further empirical studies need to be done on the kind of relations that might exist between the class genericity measured by the OVO metric and the probability of class defectiveness.

Table 3. Multivariate Model for The Polymorphism Measures

	Coefficient	ΔΨ	p-value
Intercept	1.4413		0.1210
SPA	-0.1143	89%	0.1073
SPD	-0.0497	95%	0.0475
NIP	-0.0220	97%	0.0578

4.2.2 Multivariate Analysis

According to the rank correlation (non-parametric Spearman-Rho) shown in Table 4, a statistically significant linear relation exists between the pair of SPA and DPA (static and dynamic polymorphism in ancestors), and the pair SP and DP.

Table 4. Rank Correlation between Polymorphism measures

	SPA	SPD	NIP	DPA	DPD	SP	DP
OVO	0.06	0.05	0.00	0.04	0.07	0.00	0.00
SPA	1.00	0.03	0.37	**0.71**	0.00	0.51	0.42
SPD		1.00	0.09	0.01	0.67	0.27	0.13
NIP			1.00	0.18	0.06	0.50	0.24
DPA				1.00	0.00	0.43	0.63
DPD					1.00	0.22	0.29
SP						1.00	**0.70**

This may be explained as follows:

- DPA and SPA count the number of overridden member functions between a class and its ancestors. This suggests, at least from our data set, that static polymorphism *(overriding)* and dynamic polymorphism *(ad hoc polymorphism)* tends to have the same quality impact in ancestors' relations. Further studies need, however, to corroborate this evidence.

- Since SP and DP are calculated from the four measures SPA, DPA, SPD and DPD, where SPA and DPA are correlated and SPD and DPD contributions are not statistically significant in our data set, it is expected that SP would be highly correlated with DP.

In Table 3 we present the predictive model using the statistically significant and non correlated polymorphism measures as predictors for the probability of having no fault in a class. As we can see from the table, the higher the polymorphism measures the higher the probability of having a fault in a class.

Relationships between Polymorphism Measures and Class Sizes

Table 5 presents the rank correlation between the six significant polymorphism measures and SLOC (source lines of code), a class size measure, across the analyzed C++ classes. The six measures have a correlation coefficient value approaching 0, which means that the polymorphism measures and SLOC do not capture the same dimensions. Moreover, SLOC can only be obtained at late phases of the product life cycle, whereas our polymorphism measures can be easily obtained from design documents quite early in the software development life cycle.

Table 5. Rank correlation between polymorphism measures and class sizes

	OVO	SPA	SPD	NIP	SP	DP
SLOC	0.28	0.03	0.03	0.00	0.00	0.01

Relationships between Polymorphism Measures and MOOD Measures

Table 6 presents the rank correlation between the six significant polymorphism measures and the POF measure from the MOOD set of measures developed by Abreu and Carapuça [1]. As expected, POF and our SPA and DPA measures are highly correlated since they are capturing the same forms of polymorphism namely the class polymorphic import coupling (*overriding*). However, OVO, SPD, DPD and NIP seem to capture different dimensions than POF given that they are presenting a very poor correlation with it.

Table 6. Rank Correlation between Polymorphism measures and MOOD Measures

	OVO	SPA	SPD	NIP	DPA	DPD	SP	DP
POF	0.06	**0.98**	0.02	0.37	**0.72**	0.00	0.50	0.42

Relationships between Polymorphism Measures and Chidamber and Kemerer Measures

We have also compared the significant polymorphism measures with Chidamber & Kemerer's (C&K) measures [11]. The six C&K measures are: DIT (class Depth Inheritance Tree); RFC (Response for a Class); WMC (Weighted Method per Class); CBO (Coupling Between Objects); NOC (Number of Children); and LCOM (Lack of Cohesion among Methods). Again LCOM presents a very poor distribution (see Table 14) which confirms the results provided in [2][9]. For the analyzed classes depth inheritance tree are quite flat, given that DIT_{median}= 1. The number of children of the analyzed classes (which excludes verbatim reused classes from libraries), is also very low, given that NOC_{median}= 0. This confirms the non-statistical significance in our study of the polymorphism measures in descendent relations (SPD and DPD).

Based on the results provided in , we can verify that only two C&K measures present a significant relationship with two polymorphism measures. The pair of measures NOC-SPD is very highly related. Also, the pair of measures DIT-SPA is highly correlated. These two relationships are, in fact, expected and the explanation for this phenomenon is straightforward:

Table 8. Rank correlation between the significant polymorphism and C&K measures

	WMC	DIT	NOC	CBO	RFC	LCOM
OVO	0.35	0.01	0.06	0.17	0.32	0.00
SPA	0.09	**0.76**	0.04	0.02	0.03	0.00
SPD	0.00	0.02	**0.92**	0.02	0.04	0.01
NIP	0.00	0.44	0.09	0.02	0.00	0.01
DPA	0.18	0.38	0.02	0.06	0.06	0.03
DPD	0.00	0.00	0.55	0.00	0.04	0.04
SP	0.08	0.33	0.21	0.07	0.00	0.02
DP	0.15	0.21	0.08	0.07	0.01	0.06

- NOC measures the number of children in classes. SPD measures the level of coupling due to static polymorphism (import coupling) a class has with its descendents. Greater is the number of children in a class higher will be the chances of having coupling due to static polymorphism.

- DIT calculates how deep a class is in the inheritance tree. SPA measures the level of coupling of a class C with its ancestors due to static polymorphism of C with its ancestors. By the results showed in , it seems that deeper is a class C in the inheritance tree, greater will be the chances to have high level of coupling of C with its ancestors.

As we mentioned in Section 3.1, our polymorphism measures aim at capturing different forms of class dependencies. It is interesting to notice that CBO, which measures algorithm class coupling, present a very poor correlation with the significant polymorphism measures. This confirms that our polymorphism measures are able to capture different dimensions of class coupling that are not highlighted by the CBO measure.

In order to further understand the relationships between our polymorphism measures and the C&K measures we have performed a forward and backward stepwise multivariate logistic regression. The goal is to generate a predictive model for fault-prone classes combining these two sets of OO design measures. As polymorphism measures we have considered only the statistically significant ones as potential covariates candidate along with the set of C&K measures. It is worth mentioning that given the linear correlation between static and dynamic polymorphism measures in our data set, they cannot be fitted in the same multivariate model. Thus, we have performed two separate combinations. One model combines the static polymorphism measures (OVO, SPA, SP and NIP) with C&K. The second model combines the dynamic polymorphism measures (OVO, DPA, DP, NIP) with C&K. Table 9 and Table 10 show the resulting coefficients of the forward regression models with a p-value of 0.0741 and 0.0044 respectively, and with a confidence level of 95%. The results show that DPA, SP, NIP and the C&K measure NOC are the significant linear covariates. The backward analysis did not eliminate any of the covariates in either model. This means that our polymorphism measures along with the C&K NOC measure constitute good predictors for the probability of fault-proneness.

Table 9. Multivariate Model Combining Static Polymorphism and C&K Measures

Measure	Coefficient
Intercept	1.3542
SPA	-0.0964
SP	-0.0500
NIP	-0.0195
NOC	0.2043

Table 10. Multivariate Model Combining Dynamic Polymorphism and C&K Measures

Measure	Coefficient
Intercept	1.2733
DPA	-0.6264
DP	-0.1445
NIP	-0.0178
NOC	0.3128

Relationships between Polymorphism Measures and Coupling Measures

Our measures are intended to gauge class dependencies. It is important to compare them with other set of metrics that captures class coupling such as the C-FOOD set investigated by Briand, Devanbu and Melo in[9]. Table 15 shows the rank correlation between the polymorphism measures and the C-FOOD set of measures. As we can notice, only AMMEC and SPD present a very high linear correlation. Although some statistically significant associations exist, most of them are not sufficiently strong (i.e., r^2's is not near 1) to claim that they capture identical or similar phenomenon. In fact, only the pairs ACMIC-SPA, ACMIC-DPA, and AMMEC-SPD have a r^2 greater than 50%. As we mentioned in the previous paragraphs, this confirms that the deeper the inheritance level the higher the probability of static and dynamic polymorphism between a class C and its ancestors, which makes higher the probability of coupling between C and its ancestors. Since ACMIC gauges the amount of import coupling of C with its ancestors by message exchanges, it is expected that SPA-AMMIC would be correlated. This confirms that SPA and DPA are capturing some of the import coupling forms between a class and its ancestors. As for the correlation between AMMEC (method-method export coupling in ancestors) and class static polymorphism within descendents (SPD) it is again expected. Moreover, SPD and AMMEC are symmetrically counting the same class coupling forms.

It is important to point out that none of OVO, NIP, DPD and DP has a significant linear correlation with any of the C-FOOD measures. This suggests that these polymorphism measures are indeed capturing different class coupling dimensions than those covered by the C-FOOD. Therefore, we can conclude that these polymorphism measures are complementary to the C-FOOD measures. In order to built a class defectiveness prediction model using both sets of metrics: polymorphism and C-FOOD, we have run a multivariate regression analysis on the statistically significant measures from both sets. We have first removed from our data set the noncontributing and correlated measures from the C-FOOD measures. We have obtained the subset composed of OCAIC, OCAEC, OCMIC and ACMIC. Table 11 shows the multivariate model combining the static polymorphism to the significant C-FOOD measures. The resulting coefficients of the forward regression model are computed with a p-value of 0.0421 with a confidence level of 95%. A backward analysis has removed SPA and NIP and introduced OCAEC and ACMIC with a p-value of 0.0383. This is expected in a way, since the removed polymorphism measures are capturing some of the coupling information captured by their corresponding C-FOOD measures (SPA-ACMIC are correlated) .

Table 11. Multivariate Model Combining Static Polymorphism with C-FOOD Measures

Measure	Coefficient
Intercept	1.7192
SPA	-0.0450
SP	-0.0549
NIP	-0.0184
OCAIC	-0.4279
OCMIC	0.0159

Table 12 shows the multivariate model combining the dynamic polymorphism to the significant C-FOOD measures. The resulting coefficients of the forward regression model are computed with a p-value of 0.0136 with a confidence level of 95%. The backward analysis did not remove any of the selected covariates measures.

Table 12. Multivariate Model Combining Dynamic Polymorphism with C-FOOD Measures

Measure	Coefficient
Intercept	1.5851
DPA	-0.4447
DP	-0.1030
NIP	-0.0169
OCAIC	-0.3432
OCMIC	0.0130

We should note that according to our data set, the static polymorphism measures seem to overlap with some of the C-FOOD measures since they are capturing some of the class coupling forms covered by the C-FOOD measures. However, the dynamic polymorphism measures seem to capture new coupling dimensions that are not covered by the C-FOOD measures. This confirms the previous conclusion we draw from the POF measure and the C&K measures. As also expected, the OVO measure has not been selected in any of the linear regression models since it does not have a linear impact on class defectiveness.

The multivariate regression analysis, again, indicates that our polymorphism measures constitute a good complementary set of measures to existing OO coupling metrics. Moreover, our set of metrics can be collected very early in the design phase. We have collected the polymorphism measures by static analysis performed on header files only.

5 Conclusion

In this paper, we have introduced a new set of metrics to quantify polymorphism in OO software. A set of tools to automatically extract these metrics from early designed artifacts has been built. In addition, we have empirically validated our metrics by analyzing their ability to predict fault-prone classes. We have used fault and product data from an industrial OO software system developed at CRIM for the last 5 years. The results show that:

- The suite of polymorphism measures can be used at early phases of the product life cycle as good predictors of its quality. Some of the polymorphism measures may help in ranking and selecting software artifacts according to their level of risk given the amount of coupling due to polymorphism. For example the SP and DP measures gauge the degree of method polymorphism present in an inheritance graph in a OO design.

- The suite of our polymorphism measures complements already existing OO design measures by giving more accurate information on overridden methods in a OO design. Predictive models combining our suite of polymorphism measures with other OO design measures have proved to be accurate

- As a first empirical study of polymorphism in OO design, we can draw some conclusions. First our data set indicates that contrary to the popular claims, polymorphism may increase probability of fault in OO software. This has to be further investigated, but it at least indicates that polymorphism should be used with due precautions by designers. Second, as intuitively expected, polymorphism tends to occur more in OO software design with high number of methods and deep inheritance trees. This suggests that designers tend to reuse structures and objects when they face complex systems to design. Third, static and dynamic polymorphism tend to impact the design quality in the same way; the corresponding measures seem to be highly correlated. Further investigation is needed, particularly since our data set shows a week distribution of certain types of polymorphism.

As future work, we intend to analyze polymorphism at the code level. The suite of measures we have defined and validated only accounts for polymorphism that can be detected statically from design documents (in our case C++ class interfaces). For example, dynamic polymorphism may need implementation information to better reflect polymorph behaviors in class methods. In particular, in the current work we did not addressed method invocation effects on design quality. Another aspect of polymorphism has not been considered: parameterized classes such as templates and generics. We intend to investigate the impact of using templates on OO software quality. We also intend to test the predictive accuracy of our suite of polymorphism measures in software reusability (what is the relation between highly polymorph classes and their reuse?), software maintainability (are highly polymorph classes more difficult to maintain than other classes?).

Acknowledgements. We are grateful to Michael Kangas who has built part of the tools to gather the data. We also would like to thank Hervé Marchal for his collaboration and support to collect the data. We are indebted to Prem Devanbu for his collaborative work and the review of the first versions of the paper. During this work, the authors have been in part supported by Bell Canada. And a NSERC operation grant.

6 References

[1] Abreu, F. B.; and Carapuça, R. “Object-Oriented Software Engineering: Measuring and Controlling the Development Process”. 4th International Conference on Software Quality, McLean, Virginia, USA, October 1994.

[2] V. Basili, L. Briand, and W. Melo. “A Validation of Object-Oriented Design Metrics as Quality Indicators”. IEEE Transactions on Software Engineering, October 1996.

[3] Benlarbi, S.; and Melo, W. L. “Risk Assessment in Object-Oriented Systems”. Technical report CRIM-97/04-80, Centre de Recherche Informatique de Montréal, Montréal, Que., CANADA, 1997.

[4] Benlarbi, S.; and Melo, W. L. “Measuring Polymorphism and Common Interconnections in OO Software Systems”. Technical report CRIM-97/11-84, Centre de recherche informatique de Montréal, Montréal, Que., Canada, 1997.

[5] G. Baumgartner and V. F. Russo. Implementing Signatures for C++. Technical Report CSD-TR-95-025, Computer Science Dept., Purdue University, 1995.

[6] J. M. Bieman and B.-K. Kang. “Cohesion and Reuse in an Object-Oriented System”. In Proc. Of the ACM SIGSOFT Symposium on Software Reusability, Seattle WA. pp. 259—262, 1995.

[7] G. Booch. Object-Oriented Analysis and Design with Application. Benjamin/Cummings Publishing Company, Inc., Santa Clara, California, 1994.

[8] Briand, L; Elemam, K. and Morasca, S. “Theoretical and Empirical Validation of Software Product Measures”. ISERN technical Report 95-03 1995..

[9] Briand, L; Devanbu, P.; and Melo W. L. “An Investigation into Coupling Measures for C++”. 19th International Conference on Software Engineering, Boston, USA, May 1997.

[10] Budd, T. “An Introduction to Object-Oriented Programming”. Addison-Wesley Longman, Inc, Second edition, 1997

[11] Chidamber, R.S.; and Kemerer, C. F. “A Metrics Suite for Object-Oriented Design”. IEEE Transactions on Software Engineering, 20(6), June 1994.

[12] Chidamber, R.S.; Darcy, D.P.; and Kemerer, C. F. “Managerial Use of Metrics for Object-Oriented Software: An exploratory Analysis”. KGSB Working paper no 750, University of Pittsburgh, 1996.

[13] Meyer, B. “Object-Oriented Software Construction”. Prentice Hall PTR, Second Edition 1997.

[14] Page-Jones, M. “Comparing Techniques by Means of Encapsulation and Connascence”. Communications of the ACM, September 1992.

[15] Stroustrup, B. “The C++ Programming Language”. Addison-Wesley Publishing Company, 1991.

[16] Weiss, M. A. “Algorithms, Data Structures, and Problem Solving with C++”. Addison-Wesley Publishing Company, 1996.

[17] D. Hosmer and S. Lemesho w. “Applied Logistic Regression.” Wiley-Interscience. 1989.

[18] P. Devanbu, “A language and front-end independent source code analyzer”. In Proc. of the 12th Int’l Conf. on Softw are Engineering, Melbourne, Australia, 1992. Everitt, “Cluster Analysis.”, Edward Arnold, 1993.

Table 13. Descriptive Statistics for the Polymorphism Measures

Measure	Max.	Min.	Mean	Median	Std. Dev.
OVO	15.00	0.00	3.47	3	2.71
SPA	18.00	0.00	3.54	1	4.63
SPD	111.00	0.00	3.73	0	13.87
DPA	5.00	0.00	0.73	0	1.29
DPD	28.00	0.00	0.77	0	3.35
SP	111.00	0.00	7.28	3.5	13.90
DP	28.00	0.00	1.50	0	3.49
NIP	50.00	0.00	9.38	0	16.04

Table 14 Descriptive Statistics of C&K Measures

Measure	Max.	Min.	Mean	Median	Std. Dev.
WMC	126.00	1.00	16.27	12.00	17.45
DIT	3.00	0.00	0.81	1.00	0.85
NOC	8.00	0.00	0.56	0.00	1.33
CBO	40.00	1.00	13.20	9.00	9.20
RFC	229.00	2.00	35.20	25.00	35.18
LCOM	2214.00	0.00	55.56	0.00	329.30

Table 15. Rank Correlation between the Polymorphism and the C-FOOD Measures

	OVO	SPA	SPD	NIP	DPA	DPD	SP	DP
OCAIC	0.26	0.02	0.00	0.04	0.02	0.01	0.05	0.04
ACAIC	0.01	0.04	0.00	0.00	0.05	0.00	0.03	0.04
OCAEC	0.02	0.02	0.00	0.07	0.00	0.01	0.02	0.00
DCAEC	0.02	0.00	0.06	0.00	0.02	0.09	0.03	0.04
OCMIC	0.39	0.00	0.03	0.03	0.00	0.04	0.02	0.01
ACMIC	0.06	**0.66**	0.07	0.17	**0.63**	0.04	0.30	0.33
OCMEC	0.07	0.04	0.00	0.02	0.08	0.02	0.06	0.12
DCMEC	0.13	0.02	0.41	0.03	0.00	0.37	0.19	0.13
OMMIC	0.11	0.01	0.01	0.06	0.00	0.03	0.02	0.01
AMMIC	0.07	0.57	0.03	0.36	0.29	0.00	0.24	0.14
OMMEC	0.10	0.03	0.02	0.02	0.00	0.01	0.00	0.00
AMMEC	0.02	0.03	**0.89**	0.08	0.01	0.51	0.20	0.08