On the Actual Use of Inheritance and Interface in Java Projects: Evolution and Implications

Carlos E. C. Dantas, Marcelo de A. Maia
Faculty of Computing
Federal University of Uberlândia
Uberlândia – MG – Brazil
carloseduardodantas@iftm.edu.br,marcelo.maia@ufu.br

ABSTRACT

Background: Inheritance is one of the main features in the object-oriented paradigm (OOP). Nonetheless, previous work recommend carefully using it, suggesting alternatives such as the adoption of composition with implementation of interfaces. Despite of being a well-studied theme, there is still little knowledge if such recommendations have been widely adopted by developers in general. *Aims:* This work aims at evaluating how the inheritance and composition with interfaces have been used in Java, comparing new projects with older ones (transversal), and also the different releases of the same projects (longitudinal). *Method:* A total of 1,656 open-source projects built between 1997 and 2013, hosted in the repositories GitHub and SourceForge, were analyzed. The likelihood of more recent projects using inheritance and interfaces differently from older ones was analyzed considering indicators, such as, the prevalence of corrective changes, `instanceof` operations, and code smells. Regression analysis, chi-squared test of proportions and descriptive statistics were used to analyze the data. In addition, a thematic analysis based method was used to verify how often and why inheritance and interface are added or removed from classes. *Results:* We observed that developers still use inheritance primarily for code reuse, motivated by the need to avoid duplicity of source code. In newer projects, classes in inheritance had fewer corrective changes and subclasses had fewer use of the `instanceof` operator. However, as they evolve, classes in inheritance tend to become complex as changes occur. Classes implementing interfaces have shown little relation to the interfaces, and there is indication that interfaces are still underutilized. *Conclusion:* These results show there is still some lack of knowledge about the use of recommended object-oriented practices, suggesting the need of training developers on how to design better classes.

CCS CONCEPTS

•Software and its engineering → Design classes;

KEYWORDS

Inheritance,Interfaces,Cohesion,GitHub,SourceForge,Code Smells

CASCON'17, Toronto, Canada

DOI: 10.1145/1235

ACM Reference format:
Carlos E. C. Dantas, Marcelo de A. Maia. 2017. On the Actual Use of Inheritance and Interface in Java Projects: Evolution and Implications. In *Proceedings of 27th Annual International Conference on Computer Science and Software Engineering, Toronto, Canada, November (CASCON'17),* 10 pages.
DOI: 10.1145/1235

1 INTRODUCTION

Since the emergence of the object-oriented paradigm (*OOP*), the inheritance feature has been highlighted, mainly because of the benefits of source code reuse, and the possibility of designing flexible projects using polymorphism [16]. In addition, several design patterns have been proposed using inheritance [10].

However, several studies have recommended caution regarding its use, mainly because it is likely to break class encapsulation [20] [15]. Previous works suggest the use of composition with interfaces replacing inheritance [2] [10]. Encapsulation breakage directly affects aspects such as maintainability and comprehensibility of projects [1] [7]. Some practitioners suggest the inheritance should be never used [13]. Other work suggests that at least complex inheritance hierarchies with high depth should not be constructed [7]. Since then, several new Java *APIs* and *frameworks* have emerged using different levels of interfaces abstractions. Some examples are the *JPA/JDBC APIs*, and the *java.util.Collections framework*.

Just as the Java *API* itself has undergone modifications with respect to the adoption of inheritance and interfaces, there may have been an influence on developers in general towards this practice. The Java platform has existed for over 20 years, and before that, several works have been published, alerting about possible class encapsulation breaks with the use of inheritance [14] [20] [22]. At the same time, other works have shown that inheritance has been consistently employed internally in the Java environment [23], and also in applications [24]. In addition, there is too much variation in the use of techniques, such as, encapsulation [11]. Thus, it makes sense to investigate whether inheritance has been employed differently in more recent projects. Moreover, the investigation should be extended to classes implementing interfaces, because some works suggest its use in replacement of inheritance [2] [13].

In this paper, we investigate 1,656 open-source projects developed in a time frame of 16 years, to compare new projects with old ones (transversal study), and verify if inheritance and interface implementation are being used differently with respect to the prevalence of their use, prevalence of encapsulation breaks, structural metrics [4], prevalence of code bad smells [9] [21], prevalence of corrective changes. We also conducted a study on how often and

why a programmer adds or removes inheritance and interface during the life cycle of a project (longitudinal study), in order to better understand their motivations to use the inheritance or interfaces implementation resources.

Paper organization. Section II presents the research questions, and the study design proposed to collect data, modeling and data analysis approach. The results are reported and discussed in Section III. Section IV discusses the threats that could affect the validity of our study. Section V discusses the related literature. Finally, Section VI summarizes our observations in lessons learned, and outlines directions for future work.

2 STUDY DESIGN

The goal of this research is to investigate the use of inheritance and interfaces in Java projects, analyzing the change history of Java projects and some specific releases. This analysis is restricted to classes extending internal superclasses or implementing internal interfaces. In this way, we discarded inheritances and interface implementations like `extends javax.swing.JFrame` or `implements java.lang.Runnable`. More specifically, the study aims at addressing the following six research questions:

RQ #1: Does the time the project was built influence the prevalence of use of inheritance or implementation of interfaces? We aim at answering if the practice of *avoiding inheritance and favoring composition with interfaces* has been, at least partially, recently followed by developers. If so, the trend would be to detect a growth in the use of interfaces in more recent projects, and at the same time, a decrease in the use of inheritance.

RQ #2: Does the time the project was built influence on the number of encapsulation breaks by the `instanceof` *operator?* The goal is to verify whether newer projects have achieved fewer encapsulation breaks caused by using the `instanceof` operator, in inheritance and interface implementation classes. The result would point out if developers have been observing the practice of avoiding encapsulation break.

RQ #3: Does the time the project was built influence on the number of corrective changes? We aim at answering if the inheritance and interface implementation classes are more stable in recent projects, having less corrective changes.

RQ #4: Inheritance or interface implementation classes have adequate levels of cohesion and coupling? The aim is to investigate if classes in inheritance or with interface implementation are more likely to have adequate indicators of cohesion and coupling, compared to non-inheritance or non-interface implementation. The result of this analysis would help to better characterize which *OOP* indicators are related to classes in inheritance or implementing interface.

RQ #5: Which code smells occur predominantly for classes in inheritance or classes implementing interface? The aim is to verify if classes with inheritance or interface implementation are more likely to be infected with specific code smells, and also, if those infections (in cases they exist) continue until the last project release. The result of this analysis would help to understand which design flaws are more likely to occur in classes with inheritance or interface.

RQ #6: How often does adding or removing inheritance and implementation of interfaces occur? For what reasons are these operations performed? The aim is to understand how often inheritance and interface features are inserted and removed from classes. The motivations for those operations could explain how inheritance and interface are actually employed in projects, from developers' point of view.

A. *Context Selection*

Figure 1 summarizes the steps used to filter the projects. The first step (A) obtains information from as many projects as possible, written in Java. For this, the *BOA* infrastructure was used, which has 282,693 projects from the GitHub repository and 26,688 from the SourceForge repository [8]. Both repositories were used because GitHub contains a larger number of recent projects, while SourceForge contains more older projects. Thus, the proposed research obtains a set of projects with a greater age range.

In the second step (B), projects without tags were discarded. Also, duplicate projects, git forks and tags in parallel branches were discarded. Thus, 274,479 (97.09%) projects were discarded from GitHub, and 14,150 (63.25%) from SourceForge.

In third step (C), tag names were analyzed because many tags contain names of initial release development, and our intention was to capture major releases. Thus, 7,371 distinct names were analyzed, and tag names like *start, initial, test, before, beta, alpha, demo, old, init, none, dev, example, first import, experimental, hello world, First commit, RC [0..9]* and *CR [0..9]* were discarded. This filter discarded tags from 6,498 projects, and 2,625 projects were discarded because they only had tags with these keywords.

In the fourth step (D), projects migrated from a different version control system (VCS) were discarded. The discarded projects were those whose initial commits (around 20) included more than 50% of their files. This scenario suggests more than half of development project life was implemented in another VCS. One example is OPENJDK7 [1], which 15,476 distinct classes were found in all commits, and 10,103 of these were added in their first 20 commits.

Next, the remaining projects were analysed considering four measures: number of classes in initial release, number of commits, lifetime (last commit date minus first commit date) in months, and the debut interval in months (time between creation date and initial release date). We choose the projects whose four measures are all greater their respective first quartiles Q1, as shown in Figure 2. For the number of classes, Q1 is 32 in both repositories. For commits, Q1 is 122 for GitHub, and 150 for SourceForge. For lifetime, Q1 is 9 months for GitHub and 16 for SourceForge. For debut interval, Q1 is 0 for both repositories. However, for debut interval, we expect to discard those that are greater than the third quartile Q3, which is 15 months for GitHub and 8 for SourceForge. These filters were applied in order to select projects with the best balance between number of classes, commits, lifetime and initial version with a suitable interval after the creation of the project. After applying these filters, 880 GitHub projects and 776 SourceForge projects were selected, totaling 1,656 projects. Figure 3 shows projects per year. There are at least 50 projects in each year between 2001 and 2013. Table 1 shows the number of projects, commits, classes and creation interval.

[1] https://github.com/ikeji/openjdk7-jdk

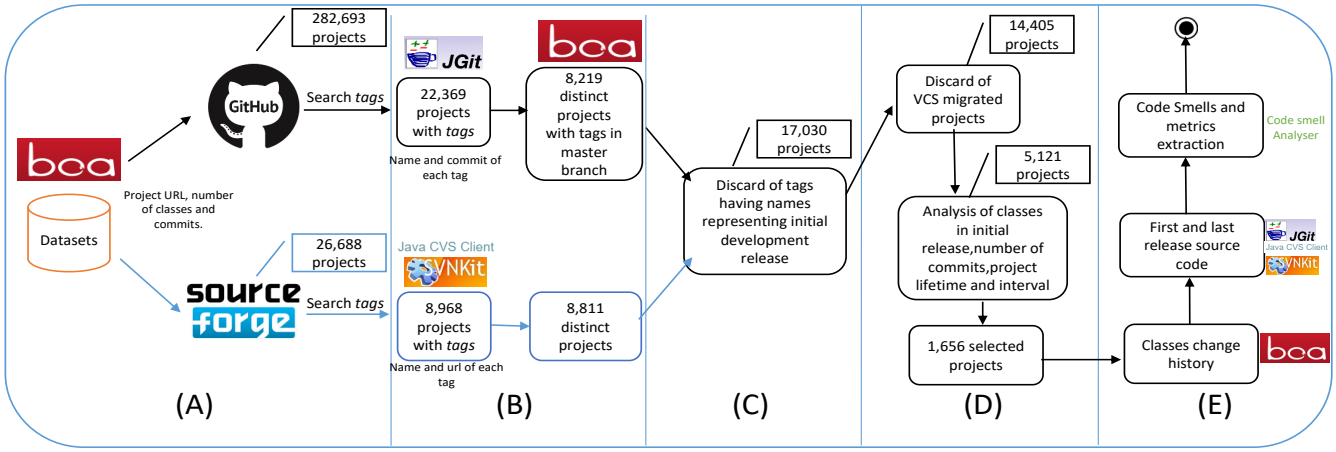

Figure 1: Steps for extract and filtering projects.

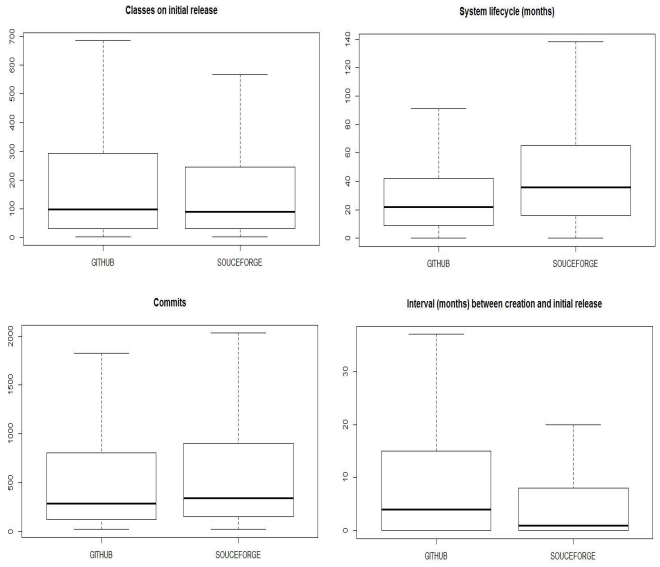

Figure 2: Boxplot to filter projects.

In fifth and last step (E), *BOA* was used to get commits and classes history, plugins VCS to download first and last release source code for all projects, and use them to input in Code Smell Analyser tool [25], to get structural metrics and code smells of each class.

Table 1: Characteristics of analyzed projects

Repository	Projects	Commits	Classes & Interfaces	Creation
GitHub	880	558,403	516,327	1997-2013
SourceForge	776	743,453	780,872	1997-2011
Total	1,656	1,301,856	1,297,199	-

B. *Data Analysis*

This subsection describes the data analysis process conducted to answer the research questions.

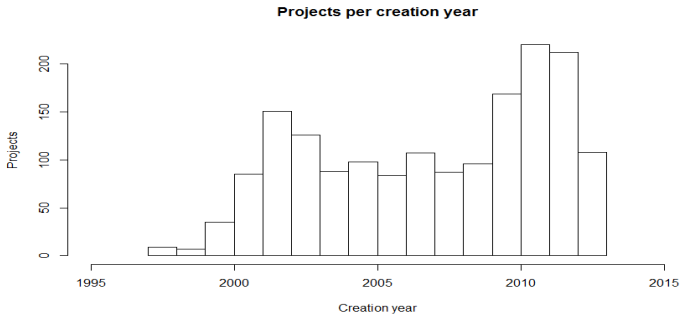

Figure 3: Number of projects per creation year.

1) The influence of age on the use of inheritance and interfaces: to answer *RQ #1*, the initial release of each Java project was evaluated, to measure the use of inheritance and interface. The initial release is expected to represent how developers thought about the architecture and building their abstractions. Then, we evaluate whether the initial release age (in months) influences the number of classes in inheritance or implementing interfaces. To perform this evaluation, we use a negative binomial regression (NBR) model, because the response variable (for each project, the number of classes in inheritance or implementing an interface) is a count. NBR can also handle over-dispersion i.e., when response variance is greater than the mean [5]. In this way, the proposed NBR model is:

$$\ln_{(INHER)} = INTERCEPT + \beta_1(CLASSES) + \beta_2(MONTHS\_RELEASE) \quad (1)$$

$$\ln_{(INTER)} = INTERCEPT + \beta_1(CLASSES) + \beta_2(MONTHS\_RELEASE) \quad (2)$$

The response variables are the lg of *INHER* (number of classes in inheritance) and *INTER* (number of classes implementing interface). The model is controlled by the number of classes – *CLASSES*, and we analyze the predictor *MONTHS_RELEASE* – the number of months between the initial project release date and the most recent project initial release date (July/2015).

Firstly, we verified if the predictive variable *MONTHS_RELEASE* is statistically significant for the response. *ANOVA* is performed,

with two NBR models, one with the *MONTHS_RELEASE* predictor variable and other without it. Next, we conducted the statistical likelihood ratio test to verify if the NBR model is more appropriate to be used with the data produced in this research question. The test will compare the NBR with the Poisson model, which has no dispersion parameter to deal with over-dispersed samples.

2) The influence of age on encapsulation breaks by `instanceof`: to answer *RQ #2*, for each project, distinct classes extending superclasses or implementing interfaces occurring as parameter of the `instanceof` operator are counted, considering all commits, i.e., if a class occurred as parameter of an `instanceof` in any of the revisions, it is counted once. Thus, we aim at identifying if the project age (in months) has any influence on the number of classes extending superclasses or implementing interfaces referenced by the `instanceof` operator. We propose the NBR model:

$$\ln_{(INHER)} = INTERCEPT + \beta_1(CLASSES) + \beta_2(EXT) + \beta_3(MONTHS) \quad (3)$$

$$\ln_{(INTER)} = INTERCEPT + \beta_1(CLASSES) + \beta_2(IMP) + \beta_3(MONTHS) \quad (4)$$

The response variables (*INHER* and *INTER*) represents the number of classes extending superclasses or implementing interface occurring as parameter of the `instanceof` operator. The predictor variable *EXT* is the number of classes extending superclasses, and *IMP* is the number of classes implementing interfaces. The last predictor variable is *MONTHS*, which is the number of months between the project creation date and the most recent project creation date (October/2013).

3) The influence of age on corrective changes: to answer *RQ #3*, fixing commits are identified using a *BOA* function called isFixingRevision(). Next, for each project, we count the number of fixing commits that have modified classes with and without inheritance or interface implementation. We analyse these groups because its necessary to know if only classes in inheritance or implementing interfaces are influenced or if all classes are influenced together. For this, as in *RQ #1* and *RQ #2*, the NBR model is proposed:

$$\ln_{(INHER)} = INTERCEPT + \beta_1(CLASSES) + \beta_2(FIX) + \beta_3(MONTHS) \quad (5)$$

$$\ln_{(NT\_IH)} = INTERCEPT + \beta_1(CLASSES) + \beta_2(FIX) + \beta_3(MONTHS) \quad (6)$$

$$\ln_{(INTER)} = INTERCEPT + \beta_1(CLASSES) + \beta_2(FIX) + \beta_3(MONTHS) \quad (7)$$

$$\ln_{(NT\_IT)} = INTERCEPT + \beta_1(CLASSES) + \beta_2(FIX) + \beta_3(MONTHS) \quad (8)$$

The response variables (*INHER* and *INTER* represents, for each project, the number of changed classes in inheritance or implementing interface in a fixing commit. The response variables (*NT_IH* and *NT_IT*) represents the number of changed classes not in inheritance or not implementing interface in a fixing commit. The predictor variable *FIX* is the total of fixing revisions for each Java project.

4) The influence of cohesion and coupling indicators: to answer *RQ #4*, we get the initial and final release of each Java project, and apply the Code Smell Analyzer tool to extract the following structural metrics values:

- Coupling between Objects (CBO) - the number of classes which a class is coupled;
- Effective Lines of code (ELOC) - The number of lines of code in a class excluding comments and white spaces;
- Lack of Cohesion (LCOM) - Correlation between methods and attributes in a class;
- Number of methods (NOM) - The number of methods in a class;
- Response for Class (RFC) - The number of distinct methods and constructors invoked by a class;
- Weighted Methods per Class (WMC) - the complexity of a class.

Thus, we aim at verifying if there is a significant difference in these metric values between classes in inheritance or implementing interfaces and classes neither in inheritance nor implementing interface. We apply the *Mann Whitney U* test. This comparison occurs twice, using two distinct releases of projects (initial and final).

5) The influence of code smells: to answer *RQ #5*, we get the initial and final release of each Java project, and apply the Code Smell Analyzer tool to its source code. We extract the classes infected by the following code smells:

- Class Data Should be Private - Classes exposing its attributes, allowing their behavior to be manipulated externally, breaking its encapsulation;
- Complex Class - Classes difficult to test and maintain, with several conditional structures and different execution paths;
- Functional Decomposition - Classes declaring many attributes and implement few methods, poorly use of OOP resources;
- God Class - Classes controlling many other objects. These classes usually have low cohesion, accumulate many responsibilities with many lines of code;
- Lazy Class - Classes having few features, not justifying their existence;
- Long Method - Methods unnecessarily long. Therefore, they could be divided, generating other methods;
- Spaghetti Code - Classes do not represent any type of behavior for the project. They usually own only a few long methods without parameters.

To identify if some code smell is occurring more frequently for classes in inheritance or implementing interfaces, a contingency table was constructed for each code smell, applying X^2 test, to verify if the hypothesis is confirmed.

6) Frequency and motivation for addition and removal of inheritance and interface: to answer *RQ #6*, we analyzed the classes change history. Each class was classified in one of four groups:

- *Always* - when classes in inheritance or implementing interfaces were built that way, and as such remained until their exclusion or last project commit.
- *Lost* - when classes in inheritance or implementing interfaces were built that way, but the inheritance or implementation was removed in some project commit.
- *Later* - when classes not in inheritance or not implementing interfaces were changed to participate in an inheritance (or to implement an interface) after some project commit.
- *Never* - when classes not in inheritance or not implementing interfaces remained as such until their exclusion or last project commit.

The first and fourth groups had the same structure from its creation to the end. However, classes in the second and third groups changed, adding or removing inheritance and interface implementation. To get better understanding, we perform a qualitative analysis

(similar to the thematic analysis [6]), on a sampling of these classes to discover the motivation for the changes. These steps were performed with 40 changes in 40 distinct projects, to prevent the same team performing more than one change. For each change, two UML class diagrams were generated [19]. The first class diagram consists the previous state of the changed class with its dependencies. And the second diagram contains the structural modification after the commit. Each change were codified, generating a theme. And then, a merge of these themes were performed, defining name and final themes.

3 RESULTS

This section reports on the achieved results, aiming at answering our six research questions.

RQ #1: Does the time the project was built influence the frequency of use of inheritance or implementation of interfaces?

Table 2 presents the *NBR* model for the influence of the initial release age in number of classes in inheritance and implementing interfaces. The *Estimate* column contains the regression coefficients for all predictor variables. For both cases, the variable *MONTHS_RELEASE* is positive. Thus, the trend is as older is the project release, the greater the number of classes with inheritance and interface, but a small association is encountered. The coefficients are interpreted as follows: for classes in inheritance, the variable *CLASSES* has a coefficient of 0.0032. This means for each class added to the project, the ln of the response variable *INHERITANCE* is summed as 0.0032, i.e., it will be summed in 1.003 classes (= $e^{0.0032}$). The same is true for the predictor variable *MONTHS_RELEASE*, for each month added in the project age, ln of the number of classes in inheritance increases by 0.002, i.e., the number of classes in inheritance increases in 1.002 (= $e^{0.002}$).

Table 2: *NBR* model for classes in inheritance and implementing interfaces

	Estimate	Std. Error	z value	Pr(> |z|)
Inheritance				
(Intercept)	3.0155966	0.0512405	58,852	< 2e-16 ***
CLASSES	0.0032857	0.0000574	57,241	< 2e-16 ***
MONTHS_RELEASE	0.0020224	0.0005005	4,041	5.32e-05 ***
Interface				
(Intercept)	2.704e+00	5,602e-02	48,266	< 2e-16 ***
CLASSES	2,880e-03	5.553e-05	51,857	< 2e-16 ***
MONTHS_RELEASE	2.240e-03	5.472e-04	4,094	4.25e-05 ***

After obtaining the coefficients, we verified whether the variable *MONTHS_RELEASE* is statistically relevant to the model. When executing the *ANOVA* test, we obtained p-value $Pr > X^2 = 6.462469e^{-}05$ for classes in inheritance and $Pr > X^2 = 6.326497e^{-}05$ for classes implementing interface. So, we accept the assumption that the coefficients of the *MONTHS_RELEASE* variable are greater than 0. Finally, the likelihood ratio test was performed to verify the *NBR* model assumptions are met. For the class in inheritance model, the likelihood log returned *195,195.1 (df = 4)*. For classes implementing interface, the value obtained was *154,311.3 (df = 4)*. The $Pr > X^2$ returned 0 for both cases. These suggest the *NBR* model is more appropriate than the *Poisson* model.

Summary for RQ #1: It was hypothesized that classes in inheritance are less commonly used in newer projects. However, there was also a reduction in the use of interface, which was not expected, since the literature recommends the use of interface composition.

RQ #2: Does the time the project was built influence on the number of encapsulation breaks by `instanceof` *operator*?

Table 3 presents the *NBR* model for the influence of project age in the use of `instanceof` operator in classes extending superclasses or implementing interfaces. The *Estimate* column shows *MONTHS* has a positive coefficient for both cases, however the association is small. Thus, older projects tend to be more discreetly associated with the `instanceof` operator to refer to classes extending superclasses and implementing interface. We discuss this result in the last section.

Table 3: *NBR* model for the influence of inheritance and interface in the use of `instanceof` operator

	Estimate	Std. Error	z value	Pr(> |z|)
Inheritance				
(Intercept)	9.977e-01	7.390e-02	13,501	< 2e-16 ***
EXT	1.808e-03	1.973e-04	9,162	< 2e-16 ***
CLASSES	3.662e-04	8.197e-05	4,468	7.9e-06 ***
MONTHS	7.305e-03	8.493e-04	8,602	< 2e-16 ***
Interface				
(Intercept)	0.3996833	0.0722599	5,531	3.18e-08 ***
IMP	0.0026862	0.0002193	12,249	< 2e-16 ***
CLASSES	0.0004840	0.0000625	7,744	9.64e-15 ***
MONTHS	0.0088140	0.0008202	10,747	< 2e-16 ***

As in *RQ # 1*, statistical tests were run to verify if *MONTHS* is statistically relevant to the model with Prob > $X^2 = 1.110223e^{-}16$ for classes with inheritance and Prob > $X^2 = 0$ for classes with interface, i.e., the assumption is accepted the coefficients of the *MONTHS* variable are greater than 0. Finally, the likelihood ratio test was performed to verify if the assumptions of the *NBR* model are met. For the class in inheritance model, the likelihood log returned *54,502.1 (df = 5)*. For classes implementing interface, the value obtained was *45,146,42 (df = 5)*. The *Prob* > X^2 returned 0 for both cases, indicating the *NBR* model is more appropriate than the *Poisson* model.

Summary for RQ #2: A small decrease in the use of `instanceof` was observed in classes extending superclasses and mainly implementing interface, which obtained a less small decrease compared to inheritance.

RQ #3: Does the time the project was built influence on the number of fixing changes?

Table 4 presents the *NBR* model for the number of corrective changes for classes in inheritance and implementing interfaces. The *Estimate* column shows *MONTHS* has a positive value for both cases. To verify if older projects tend to have more corrective changes only for classes in inheritance and implementing interface, the same NBR model was applied, but for classes not in inheritance or not implementing interface. The result shows *MONTHS* has a small influence and negative value for both cases. This means older projects tend to have a little more corrective changes for classes in inheritance and implementing interface, and newer projects tend to have more corrective changes for classes not in inheritance or not implementing interface.

When executing the *ANOVA*, we obtained the *Prob* > X^2 = 0.008424232 for classes in inheritance and *Prob* > $X^2 = 7.300391e^{-}05$ for classes implementing interface. And *Prob* > $X^2 = 0.04612451$ for classes without inheritance and *Prob* > $X^2 = 0.04100823$ for

Table 4: *NBR* model for the influence of inheritance and interface in corrective changes

	Estimate	Std. Error	z value	Pr(> |z|)
Inheritance				
(Intercept)	2.717e+00	4.896e-02	55,489	< 2e-16 ***
FIX	4.474e-03	9.751e-05	45,877	< 2e-16 ***
CLASSES	1.141e-04	2.267e-05	5,033	4.83e-07 ***
MONTHS	1.474e-03	5.763e-04	2,557	0.0106 *
Without Inheritance				
(Intercept)	3.168e+00	4.095e-02	77,360	<2e-16 ***
FIX	4.019e-03	7.947e-05	50,573	<2e-16 ***
CLASSES	-4.542e-07	1.956e-05	-0.023	0.9815
MONTHS	-9.740e-04	4.876e-04	-1,997	0.0458 *
Interface				
(Intercept)	2.316e+00	5.357e-02	43,235	< 2e-16 ***
FIX	4.032e-03	1.063e-04	37,933	< 2e-16 ***
CLASSES	1.795e-04	2.471e-05	7,265	3.72e-13 ***
MONTHS	2.445e-03	6.296e-04	3,884	0.000103 ***
Without Interface				
(Intercept)	3.352e+00	3.979e-02	84,231	<2e-16 ***
FIX	4.324e-03	7.738e-05	55,873	<2e-16 ***
CLASSES	-2.109e-06	1.904e-05	-0.111	0.912
MONTHS	-9.588e-04	4.738e-04	-2,023	0.043 *

classes without interface implementation. This indicates the assumption is accepted the coefficients of the *MONTHS* variable are greater than 0. Finally, the likelihood ratio test was performed to verify the assumptions of the *NBR* model are met. For the classes in inheritance model, the likelihood log returned *116,049.3 (df = 5)*. For classes with interface, the value obtained was *85,980.21 (df = 5)*. These very large chi-square values indicate the *NBR* model is more appropriate than the *Poisson* model. For classes without inheritance and interface, similar values were obtained.

Summary for RQ #3: In general terms, the amount of changes have slightly decreased for both classes in inheritance and classes implementing interface. This indicates they are being easier to maintain, taking into account less bugs are appearing in these classes.

RQ #4: Classes with inheritance or interface have adequate levels of cohesion and coupling?

The *Mann-Whitney U* test was applied to verify if any significant difference $\alpha = 0,05$ were found between classes in inheritance and not in inheritance and classes implementing interface and not implementing interface. Only *WMC* metrics for classes in and not in inheritance did not returned significant difference, i.e., $\alpha = 0.1741$.

Twelve violin graphics were generated as shown in Figure 4 to understand the size of effect. Each graphic represents one metric, in initial or final version, and the four violins are: classes in inheritance, classes not in inheritance, classes implementing interface, and classes not implementing interface.

For classes in inheritance, the violin graphic shows high influence on *LCOM* metric (E),(F), i.e., classes in inheritance tends to have lower cohesion than the rest, but tends to be smaller (*ELOC* (C),(D)) and thus having lower coupling (*RFC* (I),(J)).

For classes implementing interface, violin graphics shows higher cohesion for those classes compared to the rest (*LCOM* (E),(F)). Lower *RFC* (I),(J) has been observed for these classes specially in the final releases.

Summary for RQ #4: There is indication that the classes in inheritance tend to have more of a functionality because of the lower cohesion, but in fewer lines of code. For classes implementing interfaces, the metrics have better values in final releases, indicating changes in these classes do not tend to affect their behavior.

RQ #5: Which code smells occur predominantly for classes in inheritance or classes implementing interface?

Table 5 shows the results found for classes with inheritance and interface in the initial and final release of each project. For $X^2 < 0.05$, there is evidence the code smell occurs more in one scenario than in the other (in inheritance or not in inheritance, interface or not in interface). For else, code smell occurs in the same ratio in both scenarios. Thus, for classes in inheritance, lazy class and complex class has no effect in first release, but in last there is an evidence these code smells occurs more in classes in inheritance. This result indicates classes in inheritance tend to absorb more functionality during their life cycle, and some of these classes does not have too much functions. The results in classes implementing interfaces show the same proportion in initial and last release for all code smells.

Summary for RQ #5: For classes in inheritance there is an evidence the code smells lazy class and complex class occur during the changes of the classes. But for classes implementing interfaces, no changes were found in the predominance of code smells found. This indicates there may be a weak relationship between the interface and its classes implementations.

RQ #6: *How often does adding or removing inheritance and implementation of interfaces over classes occur? And for what reasons these operations are performed?*

Table 6 presents the number of classes in which there was addition or removal of inheritance and interface implementation, separated into four groups. Each group has two columns, where the column *Last* represents the number and percentage of classes persisted up to the last project commit, and the column *Delete* representing the classes that were deleted in some commit from the project. The result shows 11.33% of classes were excluded from the project in some commit. The *Always* group shows 37.11% of classes were created in inheritance and thus remained until the final project release. For interfaces, there are 17.91% of classes. And about 12% of the total classes in *Always* group were excluded from the projects, which is consistent with the overall average. This indicates classes built with inheritance or interface implementation do not have a greater tendency to exclude. The *Never* group shows 60.24% of the existing classes in the final projects release never had inheritance, and 79.68% never implemented interface.

The *Lost* group shows 0.5% of the classes had inheritance and lost until the final release, and 1.09% for implementation. And less than 10% classes have lost inheritance or implementation are deleted. This indicates most of these classes still have functionality for the projects. Another interesting factor is, by adding only the classes that acquired inheritance, only 1.45% of these lost the inheritance without immediate deletion (in the same commit). The *After* group shows 2.07% for inheritance and 1.31% for interface implementation, which is higher than the percentage of classes have lost inheritance or implementation of interfaces. When a behavior is removed, dependencies can be affected, making it possible to increase the number of changed classes or interfaces in a single commit. However, when adding a behavior, new dependencies are created along their commits. Therefore, the tendency is to add inheritance or

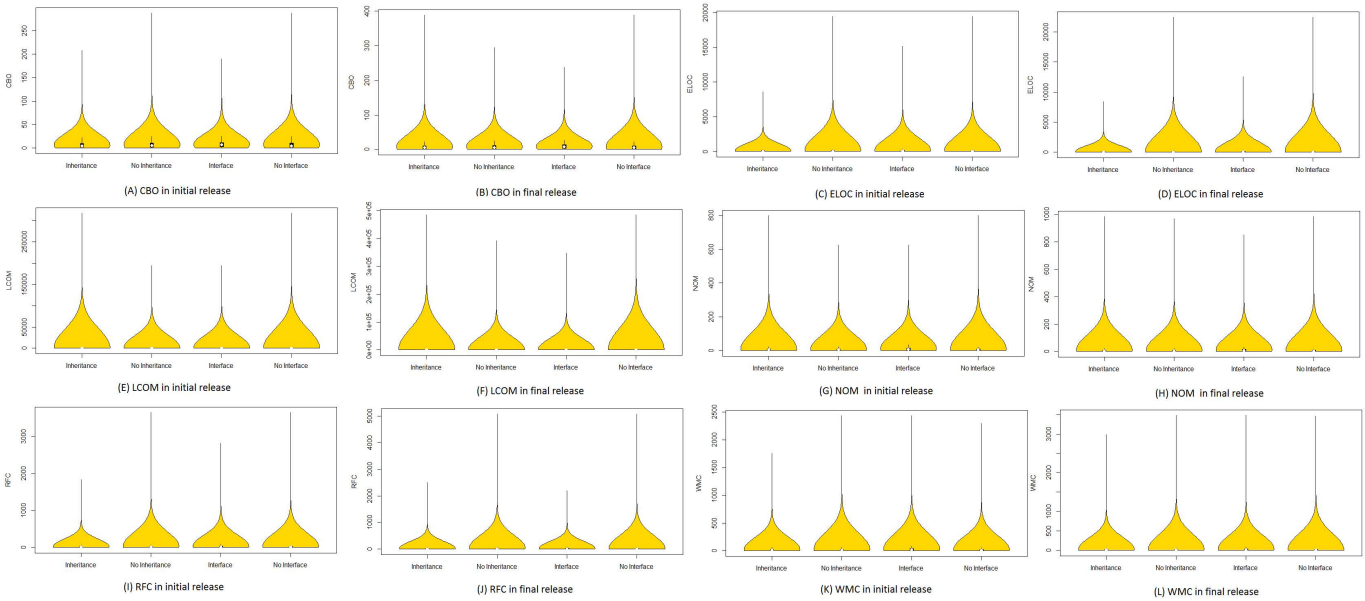

Figure 4: Metrics collected in initial and final release

Table 5: Contingency table for initial and final releases, with each code smell for classes in inheritance and implementing interfaces

	Initial Release			Final Release		
	With C.D.S.Private	Without C.D.S.Private	X^2 p-value	With C.D.S.Private	Without C.D.S.Private	X^2 p-value
In Inheritance	1,033	105,222	< 2.2e-16	1,556	163,391	< 2.2e-16
Not in inheritance	2,569	141,055		3,700	208,119	
In Interface	403	45,227	< 2.2e-16	628	69,267	< 2.2e-16
Not in interface	3,199	201,250		4,628	302,243	
	With Complex Class	Without Complex Class	X^2 p-value	With Complex Class	Without Complex Class	X^2 p-value
In Inheritance	760	105,495	0.5489	1,243	163,704	1.035e-06
Not in inheritance	997	142,627		1,316	210,503	
In Interface	763	44,867	< 2.2e-16	990	68,905	< 2.2e-16
Not in interface	994	203,255		1,569	305,302	
	With Functional D.	Without Functional D.	X^2 p-value	With Functional D.	Without Functional D.	X^2 p-value
In Inheritance	300	105,955	0.5247	460	164,487	0.5521
Not in inheritance	385	143,239		568	211,251	
In Interface	85	45,545	8.842e-05	137	69,758	1.911e-05
Not in interface	600	203,649		891	305,980	
	With God Class	Without God Class	X^2 p-value	With God Class	Without God Class	X^2 p-value
In Inheritance	2,874	103,381	< 2.2e-16	4,789	160,158	6.782e-06
Not in inheritance	4,783	138,841		6,689	205,130	
In Interface	2,538	43,092	< 2.2e-16	3,567	66,328	< 2.2e-16
Not in interface	5,119	199,130		7,911	298,960	
	With Lazy Class	Without Lazy Class	X^2 p-value	With Lazy Class	Without Lazy Class	X^2 p-value
In Inheritance	17,681	88,574	0.9795	27,516	137,431	0.05963
Not in inheritance	23,906	119,718		34,847	176,972	
In Interface	4,782	40,858	< 2.2e-16	6,384	63,511	< 2.2e-16
Not in interface	36,815	167,434		55,979	250,892	
	With Long Method	Without Long Method	X^2 p-value	With Long Method	Without Long Method	X^2 p-value
In Inheritance	944	105,311	2.887e-14	1,680	163,267	1.349e-14
Not in inheritance	1,732	141,892		2,735	209,084	
In Interface	608	45,022	2.254e-09	1,096	68,799	< 2.2e-16
Not in interface	2,068	202,181		3,319	305,552	
	With Spaghetti Code	Without Spaghetti Code	X^2 p-value	With Spaghetti Code	Without Spaghetti Code	X^2 p-value
In Inheritance	2,142	104,113	7.503e-15	3,536	161,411	8.662e-05
Not in inheritance	3,572	140,052		4,947	206,872	
In Interface	1,943	43,687	< 2.2e-16	2,636	67,259	< 2.2e-16
Not in interface	3,771	200,478		5,847	301,024	

implementation of interfaces to be an activity less complex than its removal.

Thematic analysis was applied in 40 classes to better understand the reasons why inheritance or interface implementation is added or removed some commits after creation class. Thus, each found theme is presented.

A) Uncertain Abstractions

Table 6: Number of inheritance and interface implementation additions and removals of analyzed projects

Resource	Always		Lost		After		Never		Total	
	Last	Delete	Last	Delete	Last	Delete	Last	Delete	Last	Delete
Inheritance	384,933 (37.11%)	46,950 (35.39%)	5,875 (0.5%)	480 (0.3%)	21,510 (2.07%)	1,827 (1.37%)	624,860 (60.24%)	83,371 (62.86%)	1,037,178 (88.66%)	132,628 (11.33%)
Interface	185,763 (17.91%)	23,226 (17.51%)	11,372 (1.09%)	814 (0.61%)	13,610 (1.31%)	1,043 (0.78%)	826,433 (79.68%)	107,545 (81.08%)	1,037,178 (88.66%)	132,628 (11.33%)

This theme was detected in 15 classes (37.5%), and presents situations where the superclass or interface were initially built with few functionalities being offered to the subclasses. However, after some commits, these superclasses or interfaces implementations did not obtain a justified increment to maintain their existence, and were excluded or remained without subclasses.

As an example, the JSourcePad [2] project has the subclass kkckkc.syntaxpane.model.Scope that inherits from superclass kkckkc.syntaxpane.model.Interval. However, the subclass only needed two superclass attributes, and the other behaviors were not necessary. Thus, in a given commit, the subclass was decoupled by creating the already existing attributes in superclass. The commit message mentions *reduction of memory usage*, showing in this case that it was better to avoid creating more objects unnecessarily at runtime, to the detriment of the reuse of code provided by the inheritance.

For interfaces, this scenario occurred for example in gobandroid project.[3] The interface org.ligi.gobandroid.logic.GoDefinitions .java had constants were used in classes GoGame, GoMove and GoBoard, all from the same package. However, it was necessary to add the getHandicapArray() defined method in the class GoGame. Because the classes did not rely on interface functionality, the method was moved to the interface, which was converted to class. However, the new class GoDefinitions acquired the code smell Lazy Class.

B) Standard Behavior for Interfaces

This theme was detected in 9 classes (22.5%). Occurs when classes implementing an interface require common behavior to them. Thus, to avoid duplication of source code, in all scenarios a superclass with this functionality was built. The new superclass implements the interface, and the classes previously implemented the interface, begin to extend the new superclass.

As an example, a change made occcurs in plugin gradle for netbeans [4] project. Before the change, MemProjectProperties and ProjectPropertiesProxy classes, both from the package org.netbeans.gradle.project.properties implementing the interface ProjectProperties. However, a common behavior was added to both classes, so the subclasses no longer directly implement the interface. These classes started to extend the new superclass AbstractProjectProperties. And the superclass started to extend the interface ProjectProperties. In this way, subclasses continue to implement methods defined by the interface, but their common behavior is placed in a superclass.

C) New Functions with Adoption of Good Practices

In 7 observed cases (17.5%), the change of a requirement was accompanied by an increase in the classes design quality, adopting good recommended practices. In all cases, the commit was not performed for the purpose of implementing best practices, but for implementing new project requirements, observing good *OOP* practices.

An example, a change from inheritance to composition was detected in jDTO-Binder project. [5] Before commit, com.juancavallotti.jdto.spring.SpringDTOBinder was subclass from DTOBinderBean, which implemented the DTOBinder interface. After the commit, SpringDTOBinder had an instance of DTOBinderBean, and to avoid encapsulation break, implemented the interface DTOBinder.

D) Other topics

There have been 3 cases of inheritance loss from internal superclasses, to extend external libraries. In another 2 cases, a refactoring *move package* operation was detected, which is not expected to be detected in this work. And finally, 4 cases where there was a very large number of classes changed, with several operations being performed. In these cases, no specific cause has been identified.

Summary for RQ #6: In few classes, inheritance has been added or removed after their initial commit. The analysis in a sample of 40 classes shows that inheritance is largely removed because the superclass was designed to address possible new features that never appeared in the project, and thus generally are created as Lazy Class code smell, and thus remain until the inheritance is removed. Interface hierarchies are mostly removed to avoid duplication of source code between classes implementing them. Inheritance addition also prevailed to avoid duplicity, reusing source code. Finally, interface addition occurs mainly due to polymorphism.

4 THREATS TO VALIDITY

A threat to external validity is that the analysis of this research can not be generalized to other object-oriented languages, because all the projects were written in Java. In order to generalize this study, it is necessary to evaluate projects of other object oriented languages. Moreover, only open-source projects were considered. Nonetheless, the large scale dataset provides a representative sample of open source Java projects.

A threat to construct validity concerns tags. There is no guarantee the selected tag actually represents a project release, or even it is in fact the initial or final release. Some projects have their tags names referring to a project release. For example, the initial tag in nsuml [6] is *release0_0_1*, which induces a project release. However, the first tag in *gzigzag* [7] has a name *snapshot-2000-07-11T12_26_00Z*, which does not clearly state whether it is a release or a project landmark. To mitigate this threat, we carried out a process of hygiene of tags, discarding names do not match project releases.

[2] http://tinyurl.com/jjxy5bx
[3] http://tinyurl.com/znck7ep
[4] http://tinyurl.com/hhk6dgu
[5] http://tinyurl.com/hfnlyph
[6] http://sourceforge.net/projects/nsuml
[7] http://sourceforge.net/projects/gzigzag

The detection of code smells is also a threat to validity. This is not a completely accurate process. To mitigate this threat, like other works, we relied on the results of *DECOR* implementations, which is considered a state-of-the-art tool.

Another threat to validity is that refactoring operations such as *move package* or *rename class* were disregarded. The reason for this operation is because in VCS, these operations consists in delete and added operations.

Finally, there is no accurate information about the reliability of the *isFixingRevision()* function present in the BOA infrastructure. Although it is mentioned in API as *a message is considered indicating a bug fix if it matches a set of regular expressions*, there is no information about what are these regular expressions. Nonetheless, Boa dataset is being continuously investigated by other research.

5 RELATED WORK

Several methods have been used to evaluate the use of inheritance. Many of these are prior to Java's own creation. Thus, only the works that have more relation with the purpose of this work will be highlighted.

A study proposed to evaluate how much the inheritance and interface resources are used in Java projects [23]. For this, it considered the analysis of inheritance and interface separately, as well as the inheritance occurred with external libraries and superclasses defined internally in the project. For this, a set of metrics was created, analyzing 93 projects totaling 100,000 user-defined types. In the longitudinal analysis performed, the projects acquire many new types of inheritance over time. In addition, it was found 3 out of 4 types use some form of inheritance or implementation of interfaces in at least half of the evaluated applications. Such use refers to both superclasses and interfaces (internal or external). Although this research has a quantitative purpose in evaluating the use of inheritance, it does not make measurements about the negative effects of its use, unlike this work, which measures negative effects inheritance use can give the developer.

Another study proposes to evaluate why a programmer chooses to use inheritance, and to what extent such a choice is really necessary [24]. This would identify situations where inheritance was used unnecessarily. In addition to the evaluation of class definitions, the methods were also evaluated. It was found 34% of the subclasses use superclass resources, however the subclasses are constantly polymorphically used by their superclass (two thirds of cases). It has also been found there are not many opportunities to perform refactoring operations for composition. The main difference of this study for the proposed work is that this study evaluates if the resources proposed by the inheritance were used, while the proposed work evaluates the side effects the inheritance use can generate in the source code.

Regarding the negative effects evaluation of its use, several studies have been based on the measurement of specific structural metrics inheritance, as *DIT* or *NOC*. For example, a study constructed a set of activities to be executed in classes with inheritance at different levels of *DIT*, and replicated in classes without inheritance [7]. He concluded the inheritance activity got a greater amount of changed lines of code, and the larger the value of *DIT*, the less project maintainability. Such a study was replicated by other research which proved just the opposite, i.e., classes in inheritance are easier to maintain [3]. Other study has confirmed inheritance affects the project maintainability [12]. However, he pointed out large projects are often difficult to understand with or without the use of inheritance.

About code smells, several approaches have been proposed to detect and recommend its removal. Approaches are based on the code smells cataloged from books known in the literature [9]. Some of these approaches are based on metrics [17], and others perform analysis on the history of source code versions [18]. Within metric-based approaches, one research was conducted to construct a tool called *DECOR*, using a domain-specific language *DSL* to specify the code smells [17]. For example, a *long method* can be parameterized for the number of lines the user believes to be convenient, to classify the class with the code smell mentioned. Code smell analyzer tool employed in this work was constructed, using this approach of extracting structural metrics, based on the conditions originally proposed by *DECOR* [25].

6 CONCLUDING REMARKS

In this work, we studied the actual use of inheritance and interface in Java projects. Lessons learned show a set of observations and recommendations.

Lesson # 1 - Recent projects tend to have better designed the inheritance resource, but the growth of these classes occurs faster - some improvements in the use of the inheritance were found. More recent projects have been shown fewer corrective changes, the decrease in inheritance use may indicate its use in more appropriate situations. Also, the `instanceof` operator has been less used to obtain subclass-specific functionalities. When compared with classes without inheritance, this characteristic has been designed without the trend of enlarging lack of cohesion and coupling metrics, and thus not becoming code smells. However, the last projects' releases showed classes in inheritance accumulating features more easily than classes without inheritance, and consequently some code smells appearing, as God Class and Complex Class.

Lesson # 2 - Developers still tend to design inheritance primarily for code reuse - although this work has shown improvements in the use of inheritance, the primary motivation for its use has remained the same since when this feature was conceived, which is the reuse code. Thematic analysis showed this feature is the most used by developers to avoid duplication in the source code, in detriment of the use of composition. The last projects' releases has a tendency to present the code smell Lazy Class, as well as Complex Class. Lazy Classes can be added when superclasses are defined only to avoid duplication of source code, as well as Complex Classes can be added with several functionalities for subclasses that are likely to be Lazy Classes, with few methods complementing the superclass behavior. Thus, the motivation on the part of developers to primarily obtain reuse of code, can lead to inheritance hierarchies becoming extremely complex over time. Only 1.45% classes removed inheritance indicating difficulty to change respective design decisions.

Lesson # 3 - There are some indications that interfaces are still underutilized. The thematic analysis results shown that the interfaces

were constructed with a purpose to reduce the coupling between classes, encapsulating the functionalities defined in its contract for other classes. This type of functionality may not be seen as necessary by developers when designing projects with simpler architectures. This is reinforced because in small and medium-sized projects there was a decrease in the use of interfaces. Another point is the fact that interface is a much less used resource than inheritance, with about 20% use over the project classes, against 40% of inheritance. We should also notice that several classes still extend from external superclasses, making it impossible to extend also from internal superclasses. With interface, a class can implement as many interfaces as desired, so there is no such restriction.

Lesson # 4 - Interfaces tend to have little relation with their implementation classes - Unlike inheritance, interfaces and their implementation classes do not tend to have strong relationship. Evidence for this assertion comes from the thematic analysis, where classes implementing interfaces showed greater amount of methods than those implemented to respect interfaces contract. *Refactoring* operations occur mostly often in classes implementing interfaces compared to inheritance. This indicates this relationship is easier to break. In classes in inheritance, there is a high level of coupling and shared functionality between superclasses and subclasses. In classes implementing interfaces, this coupling is lower and can be any project class that in some moment needs to be used polymorphically in the project. Thus, an interface is created and respective methods are implemented.

For future work, more detailed analysis could be conducted to characterize situations where inheritance and interface implementations have been actually considered adequate and use that finding for design decision recommendation.

ACKNOWLEDGMENTS

We would like to thank the Brazilian agencies FAPEMIG, CAPES and CNPq for partially funding this research.

REFERENCES

[1] Marwen Abbes, Foutse Khomh, Yann-Gal Guéhéneuc, and Giuliano Antoniol. 2011. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *Proc. of the* 15^{th} *Int. Conf. on Software Maintenance, Reengineering and Reverse Engineering (CSMR'2011)*. IEEE Computer Society, Washington, DC, USA, 181–190.

[2] J. Bloch. 2008. *Effective Java (The Java Series)* (2 ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

[3] Michelle Cartwright. 1998. An Empirical View of Inheritance. *Information and Software Technology* 40, 14 (Dec. 1998), 795–799.

[4] S. R. Chidamber and C. F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Trans. on Software Engineering* 20, 6 (June 1994), 476–493.

[5] J. Cohen and P. Cohen. 1975. *Applied multiple regression/correlation analysis for the behavioral sciences.* Erlbaum, Hillsdale, NJ.

[6] Daniela S. Cruzes and Tore Dyba. 2011. Recommended Steps for Thematic Synthesis in Software Engineering. In *Proc. of the* 5^{th} *Int. Symp. on Empirical Software Engineering and Measurement (ESEM'2011)*. IEEE Computer Society, Washington, DC, USA, 275–284.

[7] John Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. 1996. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering* 1, 2 (1996), 109–132.

[8] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories. In *Proc. of the* 35^{th} *Int. Conf. on Software Engineering (ICSE'2013)*. IEEE Press, Piscataway, NJ, USA, 422–431.

[9] M. Fowler, K. Beck, T. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[11] Tony Gorschek, Ewan Tempero, and Lefteris Angelis. 2010. A Large-scale Empirical Study of Practitioners' Use of Object-oriented Concepts. In *Proc. of the* 32^{nd} *Int. Conf. on Software Engineering (ICSE'2010)*. ACM, New York, NY, USA, 115–124.

[12] R. Harrison, S. Counsell, and R. Nithi. 2000. Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-oriented Systems. *Journal of Systems and Software* 52, 2-3 (June 2000), 173–179.

[13] Allen Holub. 2003. Why extends is evil: Improve your code by replacing concrete base classes with interfaces. *JavaWorld.com* (August 2003).

[14] Barbara Liskov. 1987. Keynote Address - Data Abstraction and Hierarchy. In *Proc. of the* 2^{nd} *Conf. on Object-oriented Programming Systems, Languages and Applications (OOPSLAfi1987)*. ACM, New York, NY, USA, 17–34.

[15] Robert Cecil Martin. 2003. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall PTR, Upper Saddle River, NJ, USA.

[16] B. Meyer. 1989. Reusability: The Case for Object-Oriented Design. In *Software Reusability*, T. J. Biggerstaff and A. J. Perlis (Eds.). acm press, New York, 1–33.

[17] Naouel Moha, Yann-Gael Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Trans. on Software Engineering* 36, 1 (2010), 20–36.

[18] Fabio Palomba, Gabriele Bavota, and Massimiliano Di Penta. 2013. Detecting bad smells in source code using change history information. In *Proc. of the* 28^{th} *Int. Conf. on Automated Software Engineering (ASE'2013)*. IEEE Computer Society, Palo Alto, CA, USA, 268–278.

[19] James Rumbaugh, Ivar Jacobson, and Grady Booch. 2004. *Unified Modeling Language Reference Manual* (2 ed.). Pearson Higher Education, Essex, UK, UK.

[20] Alan Snyder. 1986. Encapsulation and Inheritance in Object-oriented Programming Languages. In *Proc. of the* 1^{st} *Conf. on Object-oriented Programming Systems, Languages and Applications (OOPSLA'1986)*. ACM, New York, NY, USA, 38–45.

[21] G. Suryanarayana, G. Samarthyam, and T. Sharma. 2015. *Refactoring for Software Design Smells. Managing Technical Debt.* Elsevier, San Francisco, CA, USA.

[22] Antero Taivalsaari. 1996. On the Notion of Inheritance. *ACM Computing Surveys (CSUR)* 28, 3 (Sept. 1996), 438–479.

[23] Ewan Tempero, James Noble, and Hayden Melton. 2008. How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software. In *Proc. of the* 22^{nd} *European Conf. on Object-Oriented Programming (ECOOP'2008)*. Springer-Verlag, Berlin, Heidelberg, 667–691.

[24] Ewan Tempero, Hong Yul Yang, and James Noble. 2013. What Programmers Do with Inheritance in Java. In *Proc. of the* 27^{th} *European Conf. on Object-Oriented Programming (ECOOP'2013)*. Springer-Verlag, Berlin, Heidelberg, 577–601.

[25] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and Why Your Code Starts to Smell Bad. In *Proc. of the* 37^{th} *Int. Conf. on Software Engineering (ICSE'2015)*. IEEE Press, Piscataway, NJ, USA, 403–414.