Discrimination of Class Inheritance Hierarchies – A Vector Approach

B. Ramachandra Reddy and Aparajita Ojha

PDPM Indian Institute of Information Technology, Design and Manufacturing Jabalpur, Dumna Airport Road, P.O. Khamaria, Jabalpur, 482005, India
{brreddy,aojha}@iiitdmj.ac.in

Abstract. Numerous inheritance metrics have been proposed and studied in the literature with a view to understand the effect of inheritance on software performance and maintainability. These metrics are meant to depict the inheritance structures of classes and related issues. However, in spite of a large number of inheritance metrics introduced by researchers, there is no standard set of metrics that could discriminate the class hierarchies to decipher or predict the change-proneness, defect-proneness of classes or issues that could effectively address maintainability, testability and reusability of class hierarchies. In fact, very different hierarchical structures lead to the same values of some standard inheritance metrics, resulting in lack of discrimination anomaly (LDA). In an effort to address this problem, three specific metrics have been studied from the point of view of providing an insight into inheritance patterns present in the software systems and their effect on maintainability. Empirical analysis shows that different class hierarchies can be distinguished using the trio – average depth of inheritance, specialization ratio and reuse ratio.

Keywords: Inheritance Metrics, Class Hierarchies, Software Maintainability.

1 Introduction

Metrics play an important role in measuring the software quality, cost, complexity, maintenance efforts and many other important attributes affecting the overall software performance. If these issues are addressed and analyzed properly at design level, they lead to more efficient and cost effective software development and maintenance. Numerous metrics have been proposed over a period of two decades to assess different software attributes such as size, inheritance, cohesion, coupling, testing efforts, and maintainability. Metric suites that have been widely studied and applied successfully for object oriented software include CK metrics suite [1] and MOOD metrics suite [2]. However, there is no standard set of rules that helps a software developer or a project manager in selecting specific metrics that would be more useful for a particular development. Further, these metrics often provide overlapping information. But, the use of multiple metrics becomes inevitable for large and complex software systems. This, in its turn, generates large data sets, making it

Á. Rocha et al. (eds.), *New Perspectives in Information Systems and Technologies, Volume 2*, Advances in Intelligent Systems and Computing 276,
DOI: 10.1007/978-3-319-05948-8_12,

extremely unmanageable to analyze and interpret. Experience suggests that better inferences can be drawn by using only a small set of metrics.

Inheritance patterns present in an object oriented software system greatly affect the overall performance and maintainability of the system. Numerous metrics have been proposed by researchers and developers that measure different aspects of inheritance present in a software. Depth of inheritance (DIT), number of children (NOC), average inheritance depth (AID), specialization ratio (SR) and reuse ratio (R) metrics are some of the most commonly used metrics in quantifying the effect of inheritance. Use of inheritance metrics has claimed to reduce the maintenance efforts and enhance the reliability [3]. In an interesting study, Daly et al [4] have concluded that software systems having 3-levels of inheritance are easier to maintain than 0-level inheritance, but deeper levels of inheritance adversely affect the maintainability.

People have performed various empirical studies on design measures for object oriented software development (see for example, ([7], [8], [9], [10], [11], [12], [14], [15], [19], [22], [23], [25] and [26]). Briand et al [12] have studied relationships between coupling, cohesion, inheritance and the probability of fault detection in software. Through an empirical analysis, they conclude that accurate models can be built to predict fault-prone classes. They also show that in general, the frequency of method invocations and the depth of inheritance hierarchies are the prime factors leading to fault-proneness in the system, apart from class size. Cartwright and Shepperd [11] investigated the effect of inheritance on a specific industrial object oriented system and found that classes in an inheritance structures were approximately three times more defect-prone compared to those classes that did not participate in inheritance structure. Further, they were able to build useful prediction systems for size and number of defects based upon simple counts such as number of events or number of states per class. Similar studies were also made by Harrison et. al and Wi Li [16, 4] with a view to understand the inheritance effects on modifiability and maintainability of object oriented systems. Dallal [14] has applied the concept of class flattening to inherited classes for improving the internal quality of attributes such as size, cohesion and coupling. Elish [15] has empirically evaluated that DIT and NOC are good metrics for finding the fault tolerance, maintainability and reusability in aspect oriented systems.

Aggarwal et. al [10] have investigated twenty two different object oriented metrics proposed by various researchers and have made an empirical study of some of the selected metrics that provide sufficient information for interpretation. Metrics providing overlapping information are excluded from the set. Mishra [7] has recently introduced two inheritance complexity metrics, namely Class Complexity due to Inheritance (CCI) and Average Complexity of a program due to Inheritance (ACI). These metrics are claimed to represent the complexity due to inheritance in a more efficient way. More recently, Makkar et. al [19] have proposed an inheritance metric based on reusability of UML based software design. They have presented an empirical analysis of the proposed metric against existing reusability based inheritance metrics.

With the growing complexities of inheritance relationships and polymorphism in large object-oriented software systems, it is becoming increasingly important to

concentrate on measures that capture the dynamic behavior of the system. Han [20] et. al have proposed a Behavioral Dependency Measure (BDM) for improving the accuracy over existing metrics when predicting change-prone classes. With the help of a case study, they demonstrate that BDM is a complementary indicator for change-proneness when the system contains complex inheritance relationships and associated polymorphism. Results of the case study show that the BDM can be used for model-based change-proneness prediction.

In spite of a large number of inheritance metrics introduced by researchers, there is no standard set of metrics that could discriminate the class hierarchies to decipher or predict the change-proneness, defect-proneness of classes or issues that could effectively address maintainability of the system and reusability of class hierarchies. Dallal [13] has recently made an interesting study on cohesion metrics and has demonstrated that most of the metrics reflect the same cohesion values for classes having same number of methods and attributes but distinct connectivity patterns of cohesive interactions (CPCI). This results in incorrect interpretation of the degrees of cohesion of various classes. This is termed as lack of discrimination anomaly (LDA) problem. To resolve the problem, Dallal et. al [13] have proposed a discrimination metric and a simulation-based methodology to measure the discriminative power of cohesion metrics. The proposed metric measures the probability that a given cohesion metric will generate distinct cohesion values for classes having same number of methods and attributes but different CPCIs. Motivated by the work, we have made an empirical analysis of various available class inheritance hierarchy metrics. A tool "ClassIN" is developed to study the inheritance patterns inbuilt in Java projects. Like the cohesion metrics, inheritance metrics also suffer from LDA problem. This contribution focuses on identifying distinctive features in class hierarchies of a software system having same inheritance metric values. In this paper, we propose a vector valued metric 'Discrimination of Inheritance Pattern Vector' (DIPV), to resolve the discrimination anomaly to some extent. This measure proves to be quite useful in improving the understandability of class hierarchies present in a software system. The metric captures the distinction that a particular hierarchy possesses from others. The three measures that we have chosen for defining DIPV are average depth of inheritance, specialization ratio and reuse ratio. This trio gives a good insight of the class hierarchies and related issues such as testing efforts and maintainability.

The paper is organized as follows. Section 2 presents an overview of some of the well-known inheritance metrics. The problem of lack of discrimination anomaly for inheritance metrics is also presented in this section. In Section 3, a new approach for discriminating the inheritance pattern in a software system is presented using the trio named DIPV (a vector valued metric). An Empirical analysis is also presented in this section using ClassIN tool. The tool helps in identifying inheritance patterns present in a software system with special emphasis on the software attributes such as depth and breadth of class hierarchies. This facilitates in visualizing the general inheritance pattern present in the system.

2 Inheritance Metrics and Lack of Discrimination Anomaly

Inheritance metrics measure the depth, width and relative inheritance values reflecting the inheritance patterns in an object oriented system. Inheritance metrics are broadly classified into two types - class level inheritance metrics and class hierarchy metrics. The Class level inheritance metrics represents the inheritance values of individual class, whereas the class hierarchy metrics represents hierarchal structure of the related classes. Table 1 lists metrics that are commonly used for determining class level inheritance and class hierarchy level inheritance.

Table 1. Inheritance metrics

Inheritance Metrics	**Description**
a. Class Metrics	
Depth of Inheritance (DIT)	Maximum length from the node to root.
Number of Children(NOC)	Number of immediate descendents of the class.
Number of Ancestor classes (NAC)	Total number of super classes of the class.
Number of Descendent classes (NDC)	Total number of subclasses of the class.
Total Children Count(TCC)	Number of Subclasses of the class.
Total progeny Count(TPC)	Number of classes that inherit directly or indirectly from a class.
Total Parents count (TPAC)	Number of super-classes from which the given class inherits directly.
Number of Methods Inherited (NMI)	Number of methods inherited by the class.
Number of Attributes Inherited (NAI)	Number of attributes inherited by the class.
Class-to-leaf depth (CLD)	The maximum length of the path from a class to a leaf.
b. Class Hierarchy Metrics	
Maximum DIT (MaxDIT)	Maximum of the DIT values obtained for each class of the class hierarchy.
Average Inheritance Depth (AID)	Sum of depths of classes/Total number of classes.
Number of children for a component (NOCC)	Number of children of all the classes in the component.
Total length of inheritance chain (TLI)	Total number of edges in an inheritance hierarchy graph (number of classes inherited)
Specialization Ratio (S)	Ratio of number of subclasses to the number of super classes.
Reuse Ratio (U)	Ratio of number of super classes to the total number of classes.
Method Inheritance Factor (MIF)	Ratio of number of inherited methods in a class to the number of visible methods in a class.
Attribute Inheritance Factor (AIF)	Ratio of number of inherited attributes in a class to the number of visible attributes in a class.

DIT and NOC are most commonly used class level metrics. While these metrics do give some idea on the inheritance complexity of a system, they do not discriminate between different hierarchical patterns. AIF, MIF metrics do provide some insight into the internal structure of a hierarchy. Specialization ratio and reuse ratio are

commonly used for discriminating the class hierarchies. However, in many instances, these metric values turn out to be the same for relatively very different inheritance patterns. Although class hierarchies can be understood to some extent using specialization ratio, reuse ratio and average inheritance depth, none of the measures alone is capable of providing a good understanding of the inherent hierarchies. In what follows, we shall present some specific cases that exhibit the LDA in class hierarchy inheritance metrics.

Consider the metric MaxDIT defined by

$$MaxDIT = \max_{C_i \in K} \{ DIT\ (C_i) \}$$

where C_i is a class in a class hierarchy K. Note that MaxDIT [21] for all the three hierarchies in Figure 1 turns out to be 2, but the three hierarchies have different structures. MaxDIT does not indicate any inheritance pattern and the relative overriding complexities. Moving onto Average Depth of Inheritance (AID), it is defined by

$$AID = \frac{\sum_i DIT\ (C_i)}{TC}$$

where TC denotes total number of classes in the hierarchy. AID is a ratio of sum of DIT values to the total number of classes. It may be noted that AID value for Figure 1(a) is 1.5 and is 1.33 for hierarchies shown in Figure 1(b) and 1(c). Hence this also does not give any clear picture of the hierarchical structure present in a module. Turning to the metric NOCC [21], all the class hierarchies shown in Figures 1 have the same value 5.

Thus, the possibility of discriminating class hierarchies using NOCC is very low. Similarly, the TLI metric (total length of inheritance chain) value for all the hierarchies of Figure 1 is 5. The Specialization Ratio S (Table 1) gives some idea about the width of class hierarchy, higher the specialization ratio, wider would be the class hierarchy. The hierarchy having higher value of S is considered to be better than that having value close to 1. The specialization ratio for hierarchies of Figure 1(a) and 1(b) is the same and is equal to 2.5, while for Figure 1(c) it turns out to be 1.67.

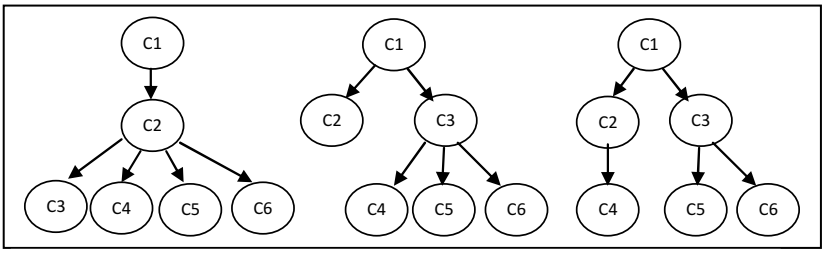

Fig. 1. Class hierarchies

Another related ratio that helps in discriminating different hierarchies is the Reuse Ratio U (Table 1). The reuse ratio is always less than 1. The higher reuse ratio reflects that the system is having deep hierarchy and high reuse value. The reuse ratio for hierarchies of Figure 1(a) and 1(b) is 0.33, 0.33, while the same for 1(c) turns out to be 0.5.

An empirical study was performed to analyze the performance of different class hierarchy metrics using six different projects from sourceforge.net [24]. These projects are Bloat (12) JActor (9), Oxygene (24), Pandora Project Management (19), SGLJ (23) and OpenNLP (41). Numbers in the brackets indicate the number of class hierarchies present in each project. Thus a total number of 128 class hierarchies were taken to analyze the discriminating power of inheritance metrics. Six metrics, namely MaxDIT, NOCC, TLI, AID, S, and U were chosen to examine the discriminating power.

Table 2. Statistics of class hierarchy metrics for 128 classes of six different projects

Metrics	**Min**	**Max**	**Mean**	**Median**	**Standard Deviation**	**Percentage of distinct metric values (%)**
Max DIT	1	5	1.48	1	0.87	3.12
NOCC	1	80	6.9	3	12.63	19.53
TLI	1	80	6.92	3	12.63	19.53
AID	0.5	2.61	1.12	1	0.4	29.68
S	1	69	4.05	2	8.03	24.21
U	0.01	1	0.48	0.36	0.31	20.31

3 Class Hierarchy Metric: A Vector Approach

Let us define the Discrimination of Inheritance Pattern Vector (DIPV) as a triple using the combination of AID, specialization ratio and reuse ratio as follows.

DIPV = (AID, S, U)

AID provides a good understanding of the inheritance levels present in a given class hierarchy. But, this alone is insufficient in giving details of inheritance pattern. Combining it with specialization ratio gives the idea of the breadth of the hierarchy, whereas the reuse ratio also gives the information about the depth. Thus DIPV gives a fair amount of indication on the maintenance efforts needed. Comparison of two DIPV is performed as follows. If $u = (x_1, y_1, z_1)$ and $v = (x_2, y_2, z_2)$ are two vectors then $u > v$ if and only if one of the following conditions hold (i) $x_1 > x_2$ (ii) $x_1 = x_2$ and $y_1 < y_2$ (iii) $x_1 = x_2$, $y_1 = y_2$ and $z_1 > z_2$. Otherwise $u <= v$.

To ascertain better maintainability, we propose the following algorithm.

Algorithm

Input: DIPV1 = (x_1, y_1, z_1), DIPV2= (x_2, y_2, z_2)

Output: Lower DIPV (lower maintainability)

Begin:

1. If $x_1 > x_2$ then
2. return DIPV2.
3. else if $x_1 = x_2$ and $y_1 < y_2$ then
4. return DIPV2..
5. else if $x_1 = x_2$ and $y_1 = y_2$ and $z_1 > z_2$
6. return DIPV2.
7. else
8. return DIPV1.
9. end if.
10. end.

Table 3. LDA cases in some class hierarchies

Class Hierarchy	MaxDIT	NOCC	AID	S	U	AM	AU
1	2	3	1	1.5	0.5	2.5	2
2	2	3	1.25	1.5	0.5	2.875	2.25
3	2	4	1.2	1.33	0.6	2.8	2.2
4	2	4	1.2	2	0.4	2.8	2.2
5	2	5	1.5	2.5	0.333	2.5	3.25
6	2	5	1.333	2.5	0.333	2.33	3
7	2	10	1.54	3.33	0.27	3.31	2.54
8	2	10	1.63	5	0.18	3.45	2.63
9	3	6	2.14	2	0.42	4.21	3.14
10	3	6	1.85	1.5	0.57	3.78	2.85
11	3	8	1.88	1.33	0.66	3.83	2.88
12	3	8	1.55	2	0.44	3.33	2.55
13	3	11	1.5	2.75	0.33	3.25	2.5
14	3	11	1.75	2.2	0.41	3.625	2.75
15	4	16	2.117	2.28	0.41	4.17	3.11
16	4	16	1.82	3.2	0.29	3.73	2.82

In order to develop some understanding on the class hierarchy metrics and their revelation about the inheritance structures, a tool "ClassIN" has been developed in the present study. The tool aims at finding out various inheritance metrics for Java projects. We have considered only abstract classes and concrete classes of the projects in the tool. Using the tool, AID is compared first, lower value gives lower maintainability and hence, we do not go further to compare the second and third components. If both AID values are the same then the second and third attributes of DIPVs are compared to ascertain the breadth and depth of inheritance patterns. Higher specialization and lower reuse ratio indicate a better design from the maintenance point of view.

Table 3 shows some of the LDA cases of class hierarchies taken the six projects mentioned in Section 2. In the first and second case, values of MaxDIT and NOCC are 2 and 3 respectively for both the class hierarchies. Applying the algorithm one gets the first DIPV. The resultant hierarchy gives lower maintenance cost, low testability and better reusability. This can also be asserted by looking at the values of AM and AU, which are precisely the maintainability metrics. In some cases all the values turn out to be the same. It indicates that both the hierarches have same maintainability cost.

3.1 Validation of DIPV

After applying DIPV, the algorithm returns either DIPV1 or DIPV2. For the validation of DIPV results, statistical discriminant analysis test was performed. The resultant discriminate function is

$$D_i = -7.146 + 1.788 * AID + 0.583 * S + 5.536 * U.$$

The discriminate function coefficients are positively correlated with discriminant function. So, AID, S and U are suitable for discrimination of class inheritance hierarchies. More than 61% grouped cases are correctly classified in all the class hierarchies present in the six mentioned projects from sourceforge.net [24].

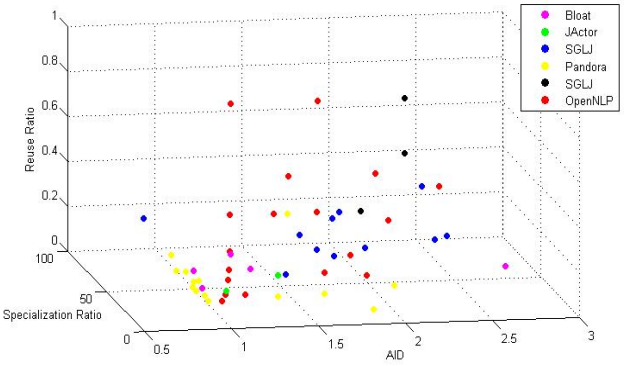

Fig. 2. A visualization of DIPVs of class hierarchies

Figure 2 shows a plot of DIPV for a total of 128 class hierarchies taken from six projects from sourceforge.net as mentioned in Section 2. For the project Pandora, one may observe that specialization ratio of one of the class hierarchies is around 70, while its reuse ratio is well within the limit (0.5) and AID is close to 2. Such hierarchies may lead to higher maintenance and testing efforts and should be reviewed at the design level only. Moving onto the class hierarchies of the Project Oxygen, all the class hierarchies have specialization ratio within the interval range of [1, 4], highest reuse ratio is 0.6 and the highest AID is 2.25. Thus the project is well designed; maintenance and testing efforts would be within the manageable limits. Thus, in general, the triple gives a good understanding of the hierarchical pattern and helps in visualizing the blueprint of the design for analysis and refinement, if needed. However, it cannot be claimed that in all cases, DIPV will always give a good insight on maintainability, reusability or testability. Nonetheless, it may be treated as a primary measure to identify class hierarchies that may create issues in the project maintenance.

4 Conclusion

Different inheritance metrics are analyzed for discriminating the class hierarchies to understand their effect on maintenance efforts. The metrics MaxDIT, AID, reuse ratio, specialization ratio can be used for discriminating the hierarchies. However, a single metric does not suffice to give any insight on the inheritance pattern. To enhance the understanding on the inheritance pattern and related software attributes such as average depth and breadth of hierarchical structure and their overriding complexities a vector DIPV has been proposed in this paper. It helps in providing a better picture of the blueprint of the inheritance pattern present in a software system. Accordingly, the design could be reviewed from the point of view of maintenance, testing efforts and reusability. A tool ClassIN is developed to visualize the inheritance patterns present in a software system. This helps in quick analysis of the software design and the inheritance patterns used in the system. The tool can be downloaded from https://sites.google.com/site/brcreddyse/Tools

References

1. Chidamber, S.R., Kemerer, C.F.: A Metrics Suite for Object Oriented Design. J. IEEE Trans. Soft. Eng. 20(6), 476–493 (1994)
2. Harrison, R., Counsell, S.J.: An Evaluation of the Mood set of Object-Oriented Software Metrics. J. IEEE Trans. Soft. Eng. 21(12), 929–944 (1995)
3. Sheldon, F.T., Jerath, K., Chung, H.: Metrics for Maintainability of Class Inheritance Hierarchies. J. Soft. Main. and Evol. Res. and Pra. 14(3), 147–160 (2002)
4. Daly, J., Brooks, A., Miller, J., Roper, M., Wood, M.: Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software. J. Emp. Soft. Eng. 1, 109–132 (1996)
5. McCabe, T.J.: A Complexity Measure. J. IEEE Trans. Soft. Eng. 2(4), 308–320 (1976)
6. Abreu, F.B., Carapuca, R.: Candidate Metrics for Object-Oriented Software within a Taxonomy Framework. J. Sys. and Soft. 26, 87–96 (1994)

7. Mishra, D.: New Inheritance Complexity Metrics for Object-Oriented Software Systems: An Evaluation with Weyuker's Properties. J. Comp. and Info. 30(2), 267–293 (2011)
8. Henderson-Sellers, B.: Object Oriented Metrics: Measures of Complexity, pp. 130–132. Prentice-Hall (1996)
9. Li, W.: Another Metric Suite for Object-Oriented Programming. J. Sys. and Soft. 44, 155–162 (1998)
10. Aggarwal, K.K., Singh, K.A., Malhotra, R.: Empirical Study of Object-Oriented Metrics. J. Obj. Tech. 5(8), 149–173 (2006)
11. Cartwright, M., Shepperd, M.J.: An Empirical Investigation of an Object-Oriented Software System. J. IEEE Trans. Soft. Eng. 26(8), 786–796 (2000)
12. Basili, V.R., Briand, L.C., Melo, L.W.: A Validation of Object-Oriented Design Metrics as Quality Indicators. J. IEEE Trans. Soft. Eng. 22(10), 751–761 (1996)
13. Dallal, J.A.: Measuring the Discriminative Power of Object-Oriented Class Cohesion Metrics. J. IEEE Trans. Soft. Eng. 37(6), 788–804 (2011)
14. Dallal, J.A.: The impact of Inheritance on the internal Quality Attributes of Java Classes. Kuw. J. Sci. and Eng. 39(2A), 131–154 (2012)
15. Elish, M.O., AL-Khiaty, M.A., Alshayeb, M.: An Exploratory case study of Aspect-Oriented Metrics for Fault Proneness, Content and fixing Effort Prediction. Inter. J. Qua. and Rel. Mana. 30(1), 80–96 (2013)
16. Harrison, R., Counsell, S.J., Nithi, R.: Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems. J. Sys. and Soft. 52, 173–179 (2000)
17. Li, W., Henry, S.: Object-Oriented Metrics that Predict Maintainability. J. Sys. and Soft. 23(2), 111–122 (1993)
18. Zhang, L., Xie, D.: Comments On the applicability of Weyuker's Property Nine to Object-Oriented Structural Inheritance Complexity Metrics. J. IEEE Trans. Soft. Eng. 28(5), 526–527 (2002)
19. Makker, G., Chhabra, J.K., Challa, R.K.: Object Oriented Inheritance Metric-Reusability Perspective. In: International conference on Computing, Electronics and Electrical Technologies, pp. 852–859 (2012)
20. Han, A., Jeon, S., Bae, D., Hong, J.: Measuring Behavioral Dependency for Improving Change-Proneness Prediction in UML-based Design Models. J. of Sys. and Soft. 83(2), 222–234 (2010)
21. Vernazza, T., Granatella, G., Succi, G., Benedicenti, L., Mintchev, M.: Defining Metrics for Software Components. In: 5th World Multi-Conference on Systemics, Cybernetics and Informatics, Florida, vol. XI, pp. 16–23 (2000)
22. Basili, V.R., Briand, L.C., Melo, L.W.: How Reuse Influences Productivity in Object-Oriented System. Commun. ACM 39(10), 104–116 (1996)
23. Briand, L.C., Wst, J., Daly, J.W., Porter, D.V.: Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems. J. Sys. and Soft. 51(3), 245–273 (2000)
24. Java Projects, `http://www.sourceforge.net`
25. Radjenovic, D., Hericko, M., Torkar, R., Zivkovic, A.: Software fault prediction metrics: A systematic literature review. J. Inf. and Soft. Tech. 55, 1397–1418 (2013)
26. Zhou, Y., Yang, Y., Xu, B., Leung, H., Zhou, X.: Source code size estimation approaches for object-oriented systems from UML class diagrams: A comparative study. J. Inf. and Soft. Tech. 56, 220–237 (2014)