Discovering the Distribution of Program Complexity from Software Repositories

Hongyu Zhang

Tsinghua University, hongyu@tsinghua.edu.cn

Hee Beng Kuan Tan

Nanyang Technological University, ibktan@ntu.edu.sg

Abstract

Large object-oriented systems are complex systems that may contain hundreds or thousands of inter-related programs. We perform an empirical study of program complexity by analyzing large Java systems in public software repositories. We discover that the complexity is not evenly distributed across programs. An interesting pattern is that a small percentage of programs are highly complex whereas most classes have small complexity measurement values. We call this phenomenon the small class phenomenon. We measure the program complexity using Lines of Code, McCabe's cycolmatic complexity, and CK's WMC metrics. In this paper, we also discuss the cause and implications of the small class phenomenon. We show that the adoption of object-orientation is a possible cause of the small class phenomenon, and that the distribution of program complexity can be used for quality prediction. We believe our study reveals the regularity that emerges from large-scale software development practices.

Categories and Subject Descriptors: D.2.8 [Software Engineering]: Metrics – product metrics.

General Terms: Measurement, Experimentation.

Keywords: program complexity, complexity distribution, small class phenomenon, software measurement, software quality, defect prediction.

1. Introduction

Handling the ever increasing complexity of software has been one of the fundamental drives of the software engineering discipline. The term software complexity is often defined as “the difficulty of performing tasks such as coding, debugging, testing and modifying the software” (Kearney, 1986). Software researchers and engineers attempt to know the complexity of the software undertaken quantitatively and to find out the relationships between the complexity and the difficulty of development/maintenance task. In general, higher complexity is likely to lead to more defects, more development and maintenance efforts.

A large software system is usually composed of a large number of inter-related programs files. For example, the Eclipse 3.0 and the JDK5.0 systems consist of 10593 and 6545 program files respectively. There have been relatively few empirical studies published that investigate issues relating to the distribution of complexity across the programs in a large software system. We believe it is essential and interesting to perform such a study to understand more about characterizes of program complexity. By doing so, we hope that we could achieve a better understanding of complex software systems and thus we could be able to better control or predict their behavior.

In recent years, mining software repositories has become an active research area. Much of the existing research has been devoted to the analysis of code changes across version histories (e.g., Zimmermann et al., 2005). We believe that the existence of large open source repository also makes a dedicated study of complexity distribution possible. We could discover regularities behind software construction by analyzing systems in software repositories.

In our work, we firstly perform detailed analysis of two well known large-scale Java systems JDK5.0 and Eclipse 3.0, focusing on the distribution of program complexity within each system. We discover that in a large system, many programs have only low complexity measures whereas a small number of programs are highly complex. For example, if we measure the program complexity based on its size, 47.56% of the JDK5.0 programs and 38.03% of the Eclipse 3.0 programs are smaller than 32 LOC, and 63.16% of the JDK5.0 and 56.42% of the Eclipse 3.0 programs are smaller than 64 LOC. We call this phenomenon the *small class phenomenon*[1]. To check the generality of the finding, we also collected 16 different large Java systems from open source repositories such as Apache[2] and Sourceforge[3], we observe the small class phenomenon in all the systems.

In this paper, we also discuss the cause and implications of the small class phenomenon. The small class phenomenon indicates that in the design of object-oriented systems, responsibilities are distributed to many classes rather than dominated by a few classes. We believe that the adoption of object-oriented decomposition and reuse techniques leads to this phenomenon, and that the large number of small programs is a natural consequence of good object-oriented development. We also show that the distribution of program complexity can be used for quality measure and defect prediction. Using the public Eclipse defect dataset, we show that a small percentage of highly complex programs are responsible for a large number of defects, therefore understanding the distribution of program complexity provides a simple yet effective method for locating defects.

2. The Small Class Phenomenon

In this research, we study the distribution of complexity across programs. Program complexity can be measured based on program size, structure, or control/data flow (Kearney, 1986; Zuse, 1990). Many complexity metrics have been proposed over the years. In this study, we choose three commonly used metrics, namely Lines of Code (LOC), McCabe's Cyclomatic Complexity (V(g)), and CK's Weighted Method per Class (WMC). A brief

[1] As in Java, a program file (.java file) can only include one public class. The complexity of private and inner classes can be counted into the associated public classes. Therefore, the complexity of program files actually represents the complexity of class.
[2] www.apache.org
[3] sourceforge.net

description of these metrics is given in the Sidebar. Many open source and commercial metric tools are available to automate the data collection process. In our study, we use the *Understand for Java*[4] tool to collect complexity measurement data for LOC, V(g), and WMC.

Sidebar: Program Complexity Measures

Software complexity has been a subject of considerable research. Software complexity can be measured at design level and program level. Design-level complexity metrics focus on the structural properties of a design, such as cohesion, coupling and depth of inheritance tree, etc. In this research, we focus on program complexity and choose the following three most commonly-used metrics:

Line of Code (LOC): LOC a size-based complexity metric. It counts each physical source line of code in a program, excluding blank lines and comments. Although some authors have pointed out the deficiencies of LOC, it is still the most commonly used size measure in practices because of its simplicity.

McCabe's cyclomatic complexity metric V(g): V(g) is a control flow based program complexity metric [3]. McCabe considers a program as a direct graph (*g*) in which the edges are control flow paths and the nodes are sequential (non-branching) code segments. For most programs, the metric is: *V(g) =edges – nodes + 2, where* V(g) is also equal to the number of decisions plus one in a flow graph. McCabe's complexity metric provides a quantitative measure of the logic complexity of a program. When used in white-box testing, it provides an upper bound on the number of test cases that will be required to guarantee coverage of all program statements. We count the V(g) of a program as the sum of the V(g) values of all its methods.

4 www.scitools.com

Weighted Method per Class (WMC): WMC is one of the CK metrics for measuring complexity of object-oriented software [1]. It is defined as WMC = Σc_i, where c_i is the complexity of each different method c_1, c_2, … c_n in class C. Based on [1], if we consider all methods of a class to be unity, then WMC is simply the number of methods defined in the class. It has been show that WMC has some positive correlation with development effort and software quality. In general, the larger the WMC, the more the development effort and the larger the probability of defect-proneness [2].

References:

1. Chidamber, S. and Kemerer, C., A Metrics Suite for Object-Oriented Design, *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476-493, June 1994.

2. Basili, V., Briand, L. and Melo, W., A Validation of Object-Oriented Design Metrics as Quality Indicators, *IEEE Trans. Software Eng*., vol. 22, no. 10, pp. 751-761, 1996.

3. McCabe, T., A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec., 1976.

A Study of JDK and Eclipse

We perform a detailed study of two well-known large-scale Java systems: JDK[5] and Eclipse[6]. JDK is Sun’s Java development toolkit, and Eclipse is perhaps the most widely used Java development platform. We choose one matured version for each system, namely JDK5.0 and Eclipse 3.0. JDK5.0 contains 6545 program files with 834K lines of code. Eclipse 3.0 contains 10593 program files with 1306K lines of code.

Table 1 shows the descriptive statistics of their program complexity data, which describes the central tendency and dispersion of the data. We find that for all metrics, the mode values (the value of the most commonly occurring item) are close to the min (minimum) values, and are smaller than the median values (the value of the middle-ranked item). The median values

5 http://java.sun.com

6 http://www.eclipse.org

are smaller than the mean values (the average value) and the max (maximum) values are much greater than other values. Therefore the descriptive statistics indicate that the distribution of program complexity is a highly skewed one, with most programs have low complexity while only a few classes have high complexity.

		Min	**Median**	**Max**	**Mode**	**Mean**	**Std Dev.**
JDK 5.0	LOC	2	36	6402	5	127.45	279.47
	V(g)	0	8	1786	2	26.83	62.36
	WMC	0	6	1192	2	12.56	26.88
Eclipse 3.0	LOC	3	51	4886	5	123.28	233.49
	V(g)	0	10	1479	0	28.45	61.93
	WMC	0	5	290	1	10.11	16.24

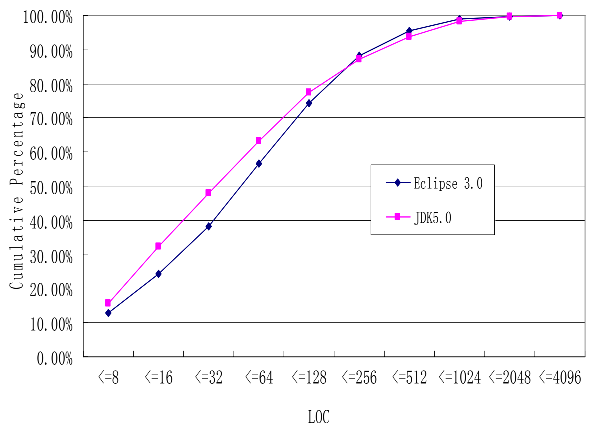
Table 1: The descriptive statistics of the program complexity data

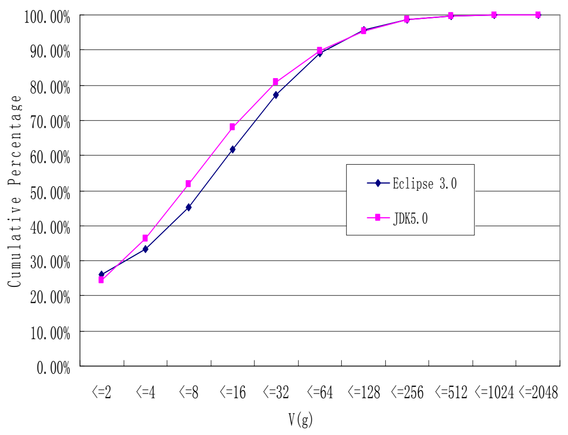
We also analyze the cumulative distribution of program complexity by grouping the measurement data into bins sized in powers of 2 (i.e., 0-8, 8-16, 16-32, etc.), and counting the number of program within each bin. We find that for LOC, 47.56% of the JDK5.0 programs and 38.03% of the Eclipse 3.0 programs are smaller than 32 LOC. 63.16% of the JDK5.0 and 56.42% of the Eclipse 3.0 programs are smaller than 64 LOC. There are also a small number of very large programs: about 6.08% of JDK5.0 and 4.39% of Eclipse 3.0 programs are larger than 512 LOC. 1.89% of JDK5.0 and 1.13% of Eclipse 3.0 programs are larger than 1024 LOC. Figure 1 (a) shows the cumulative distributions of program sizes for the JDK5.0 and Eclipse 3.0 systems.

The same phenomenon is also observed for V(g), we observe that 36.34% of the JDK5.0 programs and 33.46% Eclipse 3.0 programs have V(g) less than or equal to 4. 51.90% of the JDk5.0 programs and 45.08% Eclipse 3.0 programs have V(g) less than or equal to 8. Although many programs have low complexity, a small number of programs are quite complex: 4.59% JDK5.0 and 4.13% Eclipse 3.0 programs have V(g) values higher than 128, and 1.30% JDK5.0 and 1.17% Eclipse 3.0 programs have V(g) values higher than 256. Figure 1 (b) shows the cumulative distributions of V(g) measures for JDK5.0 and Eclipse 3.0 systems.

For WMC, we observe that 27.38% Eclipse programs have WMC values less than or equal to 2, 43.60% Eclipse programs have WMC less than or equal to 4. We also observe that a small number of programs have a large number of methods: 5.58% Eclipse 3.0 programs have WMC values higher than 32, and 1.46% programs have WMC values higher than 64. Figure 1 (c) shows the cumulative distributions of WMC measures for JDK5.0 and Eclipse 3.0 systems.

Our results indicate that in a large object-oriented system, program complexity do not follow the uniform distribution where the data is distributed evenly, nor the normal distribution where the data is distributed around an average value symmetrically. Instead, the distribution of program complexity is a highly skewed one, with most of the data skews towards the left hand side (having lower complexity). We call this phenomenon the *small class phenomenon*. We have also discovered that the distribution of program complexity can be formally described as the lognormal distribution (Zhang and Tan, 2007).





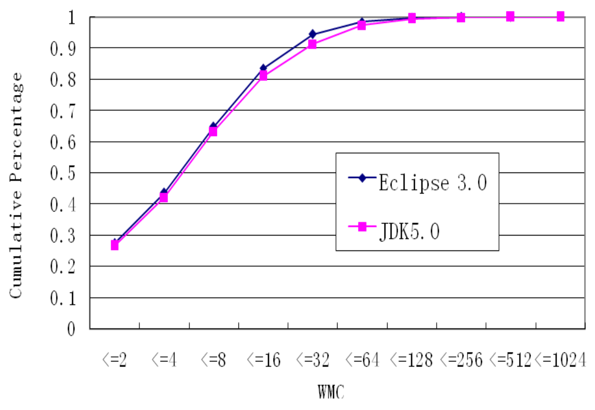


Figure 1. The cumulative distributions of program complexity

A lager experiment

To achieve a better understanding of the general nature of the results, we collect data from 16 different large Java systems as shown in Table 2 (for each system, the latest release version available at the time of the study is used). These systems are from open source software repositories such as Apache and Sourceforge, covering a wide range of domains including graphical packages, editors, web servers, and class libraries. They have 31K classes and 3.7 million lines of code in total, and 1938 classes and 229K lines of code in average. After analyzing their program complexity data, we observe the same small class phenomenon for all 16 Java systems. As an example, Table 3 shows the summary of the cumulative distribution of complexity across programs in each system.

Software	#Class	# LOC	Description
Ant	918	104857	Apache build tool
Zeus	811	105121	Agent building toolkit
Cocoon	836	75653	Apache web framework
Jena	1277	121280	Semantic Web framework
Jetspeed	689	80947	Enterprise Information portal
Jakarta Tomcat 5	1256	160745	Tomcat servlet container
rotege	697	69159	Ontology editor
jBoss	6210	587855	Application server
JDK1.4.0	3878	506965	Sun's JDK (v.1.4.0)
Jetty	314	43578	HTTP server
NetBeans	10888	1440911	Java IDE tool
Jakarta Tomcat 4	764	94496	Tomcat servlet container
ArgoUML	1205	115267	UML modeling tool
myFaces	677	63742	JavaServer face
jEdit	394	88435	Text editor
jHotDraw	195	14611	GUI framework

Table 2: The studied Java systems

3 The Cause of the Small Class Phenomenon

Contemporary software development follows the principle of modularization. In contrasting to "monolithic" software development, modularization decomposes a large solution space into a set of smaller and manageable modules, which could be separately developed and then be composed to form an executable software system. A well-designed module encapsulates certain information or implements certain "concerns".

Size (LOC)	**<= 8**	**<=16**	**<=32**	**<=64**	**<=128**	**<=256**	**<=512**	**<= 1024**	**<=2048**
Ant	6.75%	14.81%	28.00%	49.24%	74.18%	88.56%	97.39%	99.67%	100.00%
Zeus	12.08%	20.59%	36.37%	55.12%	74.60%	88.90%	97.04%	98.77%	99.26%
Cocoon	7.42%	21.53%	41.15%	61.36%	78.47%	93.54%	98.56%	99.64%	99.88%
Jena	12.29%	28.66%	46.36%	66.17%	82.69%	92.48%	96.95%	99.30%	99.77%
Jetspeed	7.84%	20.17%	33.24%	53.56%	72.42%	88.10%	96.66%	98.98%	100.00%
Tomcat 5	6.61%	15.61%	31.13%	52.39%	72.93%	86.54%	95.70%	99.12%	99.84%
Protégé	9.61%	27.26%	48.64%	68.72%	82.64%	91.10%	96.99%	98.71%	99.71%
jBoss	11.69%	29.11%	44.78%	62.91%	79.87%	91.38%	97.50%	99.47%	99.97%
JDK 1.4.0	16.53%	32.18%	46.42%	62.35%	76.74%	86.64%	93.60%	98.07%	99.77%
NetBeans	11.14%	20.74%	33.88%	51.19%	70.05%	85.90%	95.45%	99.18%	99.83%
Tomcat 4	4.84%	12.30%	30.10%	51.05%	75.13%	88.22%	95.42%	98.95%	100.00%
ArgoUML	7.05%	19.42%	43.15%	68.30%	84.32%	92.86%	97.68%	99.17%	99.59%
myFaces	6.50%	18.17%	35.60%	58.49%	78.73%	90.69%	98.67%	100.00%	100.00%
jEdit	4.57%	12.44%	21.32%	36.04%	60.15%	77.41%	90.61%	96.95%	98.48%
jHotdraw	8.72%	20.00%	37.44%	57.95%	86.15%	96.41%	98.97%	100.00%	100.00%
jetty	9.55%	23.89%	37.90%	53.50%	66.24%	82.17%	93.31%	99.36%	100.00%
Average (std dev.)	**9.53% (0.034)**	**21.84% (0.062)**	**37.81% (0.075)**	**57.04% (0.08)**	**75.90% (0.064)**	**88.64% (0.043)**	**96.10% (0.022)**	**99.01% (0.007)**	**99.75% (0.004)**

Table 3: The cumulative percentages of program sizes

In object-oriented systems such as Java systems, the large solution space is decomposed along the class boundary and the modules are implemented as classes. The very concept of “class” and “object” facilitates data abstraction, information hiding and separation of concerns. The concept of interface separates the detailed implementation and its specification. The use of inheritance and composition (delegation) allows one class to implement specific behaviors and to reuse common behaviors provided by other classes. Object-oriented practices, such as the adoptions of design patterns and refactoring also contribute to the formation of small classes. We believe it is the object-orientation that leads to the small class phenomenon introduced in this paper.

In fact, writing small classes is also encouraged by some object-oriented design methodologies. For example, Johnson and Foote (1988) summarized a set of design "rules" for developing better, more reusable object-oriented programs. Among these rules, the "Rule 9: Split large classes" and "Rule 10: Factor implementation differences into subcomponents" advocate the split of large classes into several small classes. Novice OO programmers tend to capture most of the domain and application semantics within a small subset of classes, occasionally within a single object (Lee and Tepfenhart, 2001). Such a solution is no better than the traditional procedural programming. In object-oriented design, responsibilities should be distributed among all classes of the system. Therefore, the large number of small classes actually reflects good object-oriented development, and the degree of class size distribution is an indicator of object-oriented design quality.

The existence of a large number of small programs also implied that object-oriented techniques may lead to more effort in program composition, although the effort put on individual classes is reduced. Therefore, better composition methods and tools are probably needed to reduce the program comprehension and composition efforts in object-oriented development.

4 The Implications of the Small Class Phenomenon on Software Quality

Program complexity as quality control measures

It is commonly believed that there are relationships between external software characteristics (e.g., quality) and internal product attributes (e.g., program complexity). Discovering such relationships has become one of the major objectives of software metrics. There are many regulations/guidelines/standards proposed to control program quality based on its complexity measurement. The results of the measurement could lead to revision or even rejection of the modules. For example, Bennett (1994) reported a case study where a module is rejected if its V(g) value exceeds 20. McCabe has also suggested that, on the basis of empirical evidence, the value of V(g) shouldn't exceed 10.

For LOC, Bowen (1984) reported that US military software development standards specified an average of 100 and an absolute limit of 200 executable statements for module sizes. Sun's Java code conventions suggest "Files longer than 2000 lines are cumbersome and should be avoided" (Sun, 1999). Some authors even suggested that there is an optimal size for software modules. For example, Hatton (1997) suggested an optimal range of 200-400 logical lines or 400-800 physical lines of code. Card (1990) noted that many programming texts suggest limiting module size to 50 or 60 LOC. For WMC, IBM Object Oriented Technology Council (OOTC) published a list of recommended OO metrics to the product divisions in 1993, which suggested that the methods per class measure (i.e., WMC) should be less than 20 (Lorenz and Kidd, 1994).

Our work shows that, the distribution of program complexity in typical large-scale Java systems (such as JDK, Eclipse, etc) is not uniform, nor fall within an optimal range. Instead it is a skewed distribution where most programs are "less" complex and a small percentage of programs are highly complex. We believe that in a large Java system, the uneven distribution of program complexity and the existence of a small number of highly complex programs are natural consequences of OO design. Therefore we suggest that it is not necessary to enforce a threshold or an optimal value for program complexity in practices.

Program complexity and defect prediction

It has been widely believed that programs having higher complexity are more likely to be defective. The skewed distribution of program complexity implies that the distribution of defects is also skewed. Thus, understanding the distribution of the program complexity could help us better estimate the distribution of defects. To explore the relationship between program complexity and defects, we use the public Eclipse defect dataset. The Eclipse dataset is mined from Eclipse's bug databases (BugZilla) and version achieves by the researchers at University of Saarland[7]. The detailed steps of mining defect reports can be found at (Zimmermann et al., 2007). Eclipse 3.0 has 4645 pre-release defects (the number of defects

[7] http://www.st.cs.uni-sb.de/softevo/

reported in the last six months before release) and 1534 post-release defects (the number of defects reported in the first six months after release).

	LOC	V(g)	WMC
# of Pre-Release Defects	0.421	0.389	0.319
# of Post-Release Defects	0.333	0.315	0.140

* All correlations are significant at the 0.01 level (2-tailed)

Table 4. The Spearman correlations between complexity measures and defects

To explore the relationship between program complexity and quality, we compute the Spearman rank order correlation between measurement data and defect data. The Spearman correlation is a nonparametric method for measuring the relationship between variables. The results are shown in Table 4. As all correlations are significant at 0.01 level, the complexity measures are positive correlated with the number of defects.

The existence of significant and positive correlation implies that we can predict defects based on the complexity measures. To do so, we rank the Eclipse programs according to their complexity and compute the number of defects the top *K* (*k* = 5%, 10%, 15%, 20%, 50%) most complex programs are responsible for (Table 5). For example, when we rank the programs according to their LOC values (with rank 1 being the highest LOC value), the top 10% largest programs are responsible for 46.29% pre-release defects and 44.05% post-release defects. When we rank the programs according to their V(g) values, the top 10% most complex programs contain 44.95% pre-release defects and 42.63% post-release defects. Similarly, the top 10% programs with largest WMC values contain 41.57% pre-release defects and 30.65% post-release defects. Figure 2 shows the cumulative percentage of programs vs. the cumulative percentage of defects (when the programs are ranked according to their complexity). We can see that defects are not evenly distributed across programs: a small percentage of highly complex programs contain a large number of defects and a large percentage of less complex programs contain a small number of defects. The three complexity measure, LOC, V(g) and WMC have similar defect distribution curves (except for WMC in ranking post-release defects), with LOC has slightly higher ability of predicting defects. Our results show that in practices, by simply analyzing the distribution of program complexity and identifying the most complex programs, we could quickly locate a large number of defects[8].

	Rank by	Top 5%	Top 10%	Top 15%	Top 25%	Top 50%
Pre-release Defects	LOC	32.93%	46.29%	55.05%	68.93%	87.11%
	V(g)	31.27%	44.95%	53.33%	66.03%	84.90%
	WMC	29.56%	41.57%	49.31%	61.57%	80.34%
Post-release Defects	LOC	29.97%	44.05%	52.41%	67.53%	87.68%
	V(g)	30.35%	42.63%	51.40%	65.88%	85.26%
	WMC	21.16%	30.65%	36.73%	47.26%	68.57%

Table 5: Defect prediction based on program complexity

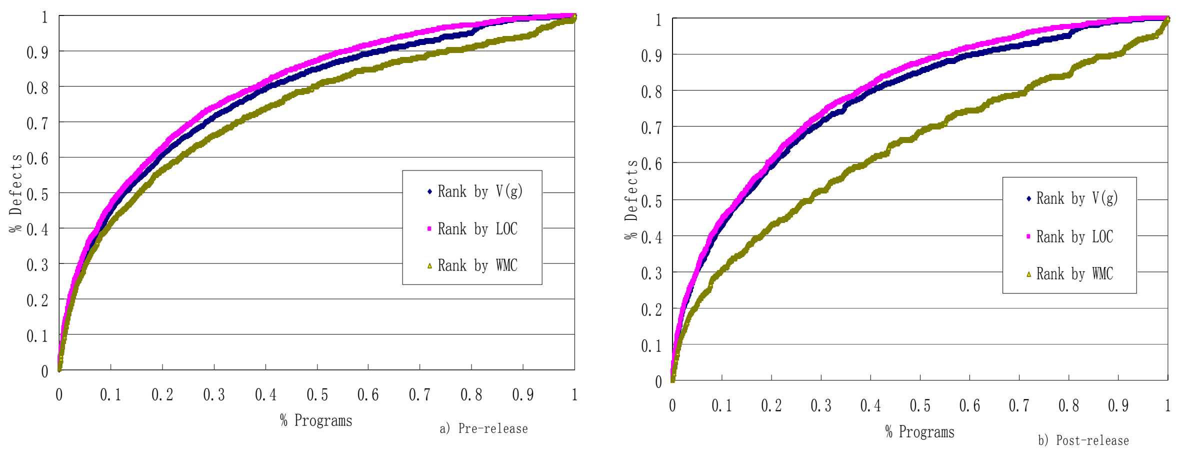


Figure 2. The cumulative percentage of programs vs. the cumulative percentage of defects (programs are ranked in descending order according to their complexity)

6. Conclusions

Large object-oriented software systems are complex systems. These systems are composed of a web of inter-related programs, which have different responsibilities and could be developed by

[8] Note that we do not claim that LOC, V(g) and WMC are the best set of metrics for predicting defects. We use these three common metrics to show the uneven distribution of program complexity and its implication on quality prediction. For guidelines for selecting a subset of metrics for more accurate defect prediction, we refer the readers to (Nagappan et al, 2006).

different programmers at different time. In this research, we have studied the distribution of complexity across programs in a large system. We have collected complexity measurement data from public software repositories and found an interesting statistical pattern: that many Java programs have only low complexity whereas a small percentage of programs have high complexity. We call this phenomenon the small class phenomenon.

We have discussed the possible causes of the small class phenomenon. We believe it is the object-orientation that leads to the small class phenomenon, and the large number of small classes actually reflects good object-oriented development. We also discuss the implications of the small class phenomenon on software quality. As the uneven distribution of program complexity is a natural consequence of OO design, we suggest not enforcing a threshold or an optimal value for program complexity in practices. Using the public Eclipse defect dataset as an example, we also show that a small percentage of most complex programs (e.g., top 15%) are responsible for a large number of defects, while a large number of less complex programs contain a small number of defects. Therefore, when software quality assurance resources are limited, we suggest begin the test/review process with the most complex programs.

Our study shows that we can discover interesting regularities/patterns by analyzing the software repositories, and then use these regularities/patterns to guide software engineering practices. In future, we plan to further analyze the relationship between design quality and complexity distribution. We also encourage the readers to replicate our study on your own software archives, and explore further the implications of the small class phenomenon.

Acknowledgement

This research is partially supported by the 6th Key Researcher Program of the Tsinghua University, the Chinese NSF grant 60703060, and the 863 project 2007AA01Z122.

References

[1] Bennett, P.A., Software Development for the Channel Tunnel: a Summary, *High Integrity System*, 1 (2), pp. 213-20, 1994.

[2] Bowen, J., Module size: A standard or heuristic?, *Journal of Systems and Software*, vol. 4 (4) , November 1984, pp. 327-332.

[3] Card, D. and Glass, R., *Measuring Software Design Quality*, Prentice-Hall, 1990.

[4] Hatton, L., Re-examining the Defect-Density versus Component Size Distribution, *IEEE Software*, March/April 1997.

[5] Johnson, R. and Foote, B., Designing Reusable Classes, *Journal of Object-Oriented Programming*, vol. 1, no. 2, June/July 1988, pp. 22-35.

[6] Kearney J. et al., Software Complexity Measurement, *Communications of the ACM*, 29 (11), 1986.

[7] Lee, R.C. and Tepfenhart, W.M., *UML and C++: A Practical Guide to Object-Oriented Development*, Prentice Hall, 2001.

[8] Lorenz, M. and J. Kidd, *Object-oriented Software Metrics: A Practical Guide*, Prentice Hall, 1994.

[9] Nagappan, N., Ball, T. and Zeller, A., Mining Metrics to Predict Component Failures, *Proc. of International Conference on Software Engineering (ICSE 2006)*, Shanghai, China, 2006. ACM Press.

[10] Sun Microsystems, Inc., *Code Conventions for the Java Programming Language*, 1999, available at: http://java.sun.com/docs/codeconv/

[11] Zhang, H. and Tan, H. B. K., An Empirical Study of Class Sizes for Large Java Systems, *Proc. of 14th Asia-Pacific Software Engineering Conference (APSEC 2007)*, Nagoya, Japan, 2007. IEEE Press, pp. 230-237.

[12] Zimmermann, T., Premraj, R. and Zeller, A., Predicting Defects for Eclipse, *Proc. 3rd Int'l Workshop on Predictor Models in Software Eng.*, 2007. The Eclipse data is available at http://www.st.cs.uni-sb.de/softevo/.

[13] Zimmermann, T., Weibgerber, P., Diehl, S., Zeller, A., Mining Version Histories to Guide Software Changes, *IEEE Transactions on Software Engineering*, vol. 31(6), pp.429-445, 2005.

[14] Zuse, H., *Software Complexity: Measures and Methods*, Walter de Gruyter, Berlin, 1990.