An Empirical Study of Class Sizes for Large Java Systems

Hongyu Zhang
School of Software
Tsinghua University
Beijing 100084, China
hongyu@tsinghua.edu.cn

Hee Beng Kuan Tan
School of Electrical & Electronic Engineering
Nanyang Technological University
Singapore 639798
ibktan@ntu.edu.sg

Abstract

We perform an empirical study of class sizes (in terms of Lines of Code) on a number of large Java software systems, and discover an interesting pattern - that many classes have only small sizes whereas a few classes have large size. We call this phenomenon the small class phenomenon. Further analysis shows that the class sizes follow the lognormal distribution. Having understood the distribution of class sizes, we then derive a general size estimation model, which reveals the relationship between the size of a large Java system and the number of classes the system has. In this paper, we also show that the adoption of object-orientation is a possible cause of the small class phenomenon. We believe our study reveals the regularity that emerges from large-scale object-oriented software construction, and hope our research can contribute to a deep understanding of computer programming.

1. Introduction

Size is an important internal attribute of software, which can be measured statically without having to execute the software system. A number of previous studies showed that size has strong correlations with many software characteristics such as the complexity [15, 23], number of faults [14, 21], development effort [3, 26], and maintenance effort [25]. A large software system is usually composed of many modules of different sizes. In general, modules with larger size are likely to contain more faults, have higher complexity and require more development and maintenance efforts.

In spite of its importance, there have been relatively few empirical studies published that investigate issues relating to the size distribution. We believe it is essential that we perform such a study to understand more about characterizes of software size. By doing so, we hope we could achieve a better understanding of the OO software and thus we could be able to better control or predict its behavior.

In object-oriented paradigm, a large solution space is decomposed into a set of collaborating classes. In this paper, we present our empirical study of class sizes on a number of large Java systems. We discover that many classes have only small sizes whereas a few classes have large size. In average 57.04% of the classes are smaller than 65 LOC and 75.90% of the classes are smaller than 129 LOC. We call this phenomenon the *small class phenomenon*.

In this paper, we also discuss the possible cause and implications of the small class phenomenon. We believe that the adoption of object-oriented decomposition and reuse techniques leads to this phenomenon, and that the large number of small classes is a natural consequence of good object-oriented development.

Further analysis shows that there is a statistical regularity behind large-scale object-oriented development – that the sizes of classes follow the lognormal distribution, which is a widely observed statistical distribution behind many natural and social phenomena. For large Java systems whose class sizes are lognormally distributed, we can then derive a size estimation model that reveals the statistical relationship between software size and the number of classes. This relationship shows that we could estimate size of a large Java software system solely based on the number of classes the system has. Our size estimation model could be useful for refining early size estimates made during analysis and design stages. To our best knowledge, our findings are novel.

The organization of the paper is as follows: in Section 2, we show the collected Java systems on which this study is performed. In Section 3 we describe the small class phenomenon and the possible cause of this phenomenon. Section 4 introduces the discovered lognormal distribution of class sizes and the derivation and validation of our proposed size estimation model. We briefly discuss the related work in Section 5 and conclude the paper in Section 6.

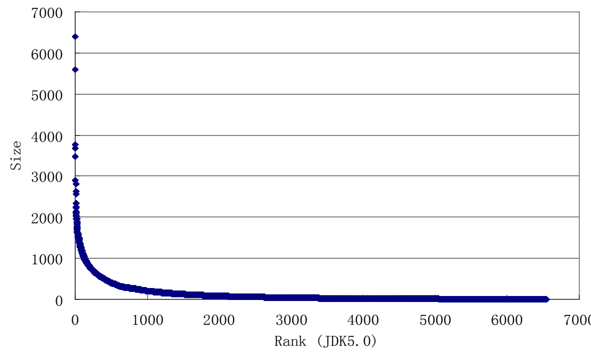
1530-1362/07 $25.00 © 2007 IEEE
DOI 10.1109/ASPEC.2007.64

2. Data collection

To analyze the size distribution of classes, we collect data from eighteen large Java systems as shown in Table 1. These systems cover a wide range of domains including graphical packages, editors, web servers, integrated development tools and class libraries. They have in average 2770 classes and 339K line of code. In this study, we use the Line of Code (LOC) as the size metric as it is still the most widely used metric for measuring size of source code. We count physical source line of code in each source file (.java file), excluding blank lines and comments. We do not count private and inner classes separately as they merely support associated public classes. The sizes of private and inner classes are counted into the associated public classes. We consider Java Interfaces as special cases of abstract classes and count interface files as class files. Table 2 shows the descriptive statistics of the class sizes for all collected systems.

Table 1. The studied Java systems

Software	Number of Classes	Total LOC	Description
Ant	918	104857	The Apache Ant (v1.6.5), a Java-based build tool, available at http://ant.apache.org/
Zeus	811	105121	A software agent building toolkit (v1.03b), available at http://more.btexact.com/projects/agents.htm
Cocoon	836	75653	An Apache web development framework (v2.0.4), available at http://cocoon.apache.org/
Jena	1277	121280	A Java framework for writing Semantic Web applications (v2.2), available at http://jena.sourceforge.net/
Jetspeed	689	80947	An Apache implementation of an Enterprise Information portal (v1.6), available at: http://portals.apache.org/jetspeed-1/
Jakarta Tomcat 5	1256	160745	The Apache Tomcat servlet container (v5.5.9), available at http://jakarta.apache.org/tomcat/
Protege	697	69159	An ontology editor and knowledge framework (v3.0), available at http://protege.stanford.edu/
jBoss	6210	587855	A Java application server (v4.0.2), available at: http://www.jboss.com
JDK1.4.0	3878	506965	Sun's JDK1.4.0, available at: http://java.sun.com
JDK5.0	6545	834185	Sun's JDK5.0, available at: http://java.sun.com
NetBeans	10888	1440911	A Java IDE tool (v4.1), available at: http://java.sun.com
Eclipse (plugins)	12299	1596401	A Java IDE tool (v3.1), available at: http://www.eclipse.org
Jakarta Tomcat 4	764	94496	The Apache Tomcat servlet container (v4.1.31), available at http://jakarta.apache.org/tomcat/
ArgoUML	1205	115267	A UML modeling tool (v0.19.6), available at http://argouml.tigris.org/ .
myFaces	677	63742	An Apache implementation of web application framework JavaServe Face (v1.0.9), available at http://myfaces.apache.org/index.html
jEdit	394	88435	A program text editor (v4.2), available at http://www.jedit.org/
jHotDraw	195	14611	A Java GUI framework (v5.3), available at http://www.jhotdraw.org/
Jetty	314	43578	A Java HTTP server and servlet container (v5.1.4), available at http://jetty.mortbay.org/jetty/



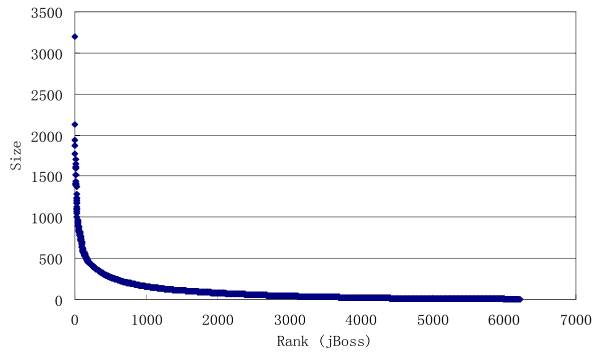


Figure 1. The distribution of class sizes (sorted in descendant order)

Table 2. Descriptive statistics of the class sizes (in LOC)

	Min	Median	Max	Mean	Std Dev.	Mode
Ant	3	65	1262	114.22	148.26	4
Zeus	3	51	4189	128.84	300.96	24
Cocoon	3	45.5	3200	90.49	161.93	9
Jena	3	36	4235	94.97	221.08	9
jetSpeed	3	59	1365	117.32	174.10	13
Tomcat 5	3	58	2930	127.98	211.46	20
Protege	3	34	2888	99.22	232.52	11
jBoss	3	40	3198	94.63	166.16	9
Tomcat 4	3	62.5	1878	123.69	195.32	20
JDK 1.4.0	3	39	3312	130.56	265.01	5
JDK 5.0	3	36	6402	127.45	279.47	5
Netbeans	3	61	5365	132.09	224.79	5
Eclipse	3	53	8255	129.79	276.65	5
ArgoUML	3	38	5647	95.66	264.69	15
myFaces	4	52	999	94.15	118.89	14
jEdit	4	93.5	5320	224.45	466.58	16
jHotdraw	4	49	629	74.93	90.37	69
Jetty	3	57	1348	138.78	194.02	15

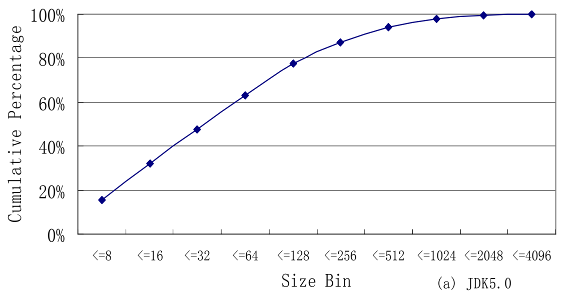
3. The small class phenomenon and the possible cause

3.1 The small class phenomenon

In this research, we study the distribution of sizes across classes. To give a visualized picture of the distribution, for each system we sort the class sizes in descendant order and plot the sizes against their ranks. As an example, Figure 1 shows the size-rank plots for the JDK5.0 and the jBoss systems. The plots show the interesting "long-tail" behavior: that most of the classes having small sizes.

We analyze the collected data by grouping the class sizes into bins sized in powers of 2 (i.e., 0-8, 8-16, 16-32, etc.), counting the number of classes within each bin, and calculating the cumulative percentage for each size bin. The results show that there are many more small classes than large classes. For example, if we consider classes having 32 LOC or less "small" classes, then 47.56% of the JDK5.0 classes and 44.78% of the jBoss classes are "small". If we consider classes having 64 LOC or less "small" classes, then 63.16% of the JDK5.0 classes and 62.91% of the jBoss classes are "small". Figure 2 shows the curves of cumulative percentages of class sizes for the JDK5.0 and jBoss systems. In average, for Java systems shown in Table 1, 57.04% of the classes are smaller than 65 LOC, 75.90% are smaller than 129 LOC and 88.64% are smaller than 257 LOC. We call this phenomenon the small class phenomenon.

Our results indicate that the class sizes 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 class sizes is a highly skewed one, with most of the data skews towards the left hand side (with smaller size). This discovery shows that in the design of these object-oriented systems, responsibilities are distributed to many classes rather than dominated by a few classes.



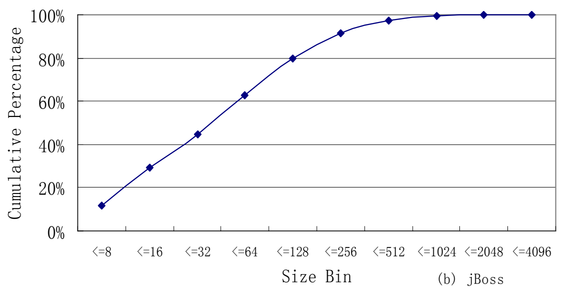


Figure 2. The cumulative percentage of class sizes

The small class phenomenon is also observed by some other researchers. For example, Wilde et al. [28] studied maintainability of three object-oriented systems written in C++ and Smalltalk. Two systems were developed in Bell Communication Research (Bellcore), one system was a PC Smalltalk environment that has an extensive class library. They found that many methods in the classes have small sizes. In all three systems, half or more of the methods are fewer than four Smalltalk lines or two C++ statements. Chidamber and Kemerer [7], during their study of design complexity of two C++ and Smalltalk systems, found that "most classes tend to have a small number of methods (0 to l0), while a few outliers declare a large number of them". In this research, we dedicate our study to the issues relating to class size distribution. We analyze eighteen large Java systems empirically and confirm that a large proportion of classes are small. We will also formally describe the class size distribution in Section 4.

3.2 Small class phenomenon and object-orientation

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”.

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 [20] 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 [22]. 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.

Understanding the nature of the class size distribution has many implications. One implication is that object-oriented techniques may lead to more effort in program composition because of the existence of the large number of small classes, 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.

As size has correlations with software quality [23, 14] and effort [3, 25, 26]. Understanding the distribution of the class sizes could also help us better estimate the distribution of defects and cost. Current software quality and cost models could be revised to take the size distribution into consideration.

4. The lognormal distribution of class sizes

4.1 Lognormal distribution

A random variable X has a lognormal distribution if the random variable *Y= ln(X)* is normally distributed, where *ln(X)* is the natural logarithm of *X*. The lognormal distribution has the following probability density function:

$$f(x) = \frac{1}{\sigma x \sqrt{2\pi}} \exp\left(\frac{-(\ln x - \mu)^2}{2\sigma^2} \right) \quad (1)$$

where μ and σ are the mean and standard deviation of the variable’s logarithm. μ is also called the scale parameter and σ the shape parameter. Taking the natural logarithm on both sides of the equation (1), we get:

$$\ln f(x) = -\ln(x) - \ln(\sigma) - 1/2\ln(2\pi) - (\ln(x) - \mu)^2/2\sigma^2 \quad (2)$$

which has the same form as the quadratic equation:

$$y = \ln f(x) = \beta_0 + \beta_1 \ln(x) + \beta_2(\ln(x))^2 \quad (3)$$

where $\beta_1 = \mu/\sigma^2 - 1$ and $\beta_2 = -1/(2\sigma^2)$. So a lognormal distribution can be seen as a quadratic function curve on a log-log plot. If the logarithm values of empirical data result in a quadratic function curve, we can reason that the data is lognormally distributed.

The lognormal distribution has a wide variety of applications. Many physical, chemical, biological, social and economical processes tend to create random variables that follow lognormal distributions [11, 24]. For example, a study in 1914 reported that inheritance of fruit size fits the lognormal distribution [17].

4.2 The derivation of lognormal distribution of class sizes

In Section 3 we described the small class phenomenon through empirical study of the data collected in Section 2. Further analysis of the data reveals that the class sizes follow the lognormal distribution. To show the lognormal distribution, we perform the following steps:

1. Group the class sizes into bins sized in powers of 2 (i.e., 0-8, 8-16, 16-32, etc.).
2. For each size bin, calculate its geometric mean of the bin endpoints (by multiplying the endpoints and taking the square root).
3. Compute the frequency by taking the total number of classes in each bin and dividing it by the width of the bin.
4. Plot the data on a log-log diagram.

As an example, Figure 3 shows the resulting plots for the JDK 5.0 and jBoss systems. Polynomial regression analysis results in quadratic functions (for JDK5.0: β_2 = -0.2074, β_1 = 0.4879, β_0 = 4.8136, and for jBoss: β_2 = -0.333,

β_1 = 1.4552, β_0 = 3.1571) on the log-log plots. The R^2 values are 0.9942 and 0.9938 respectively, indicating good fitness of the data. Referring to the equation (3), we understand that the class sizes follow the lognormal distribution as the regression curves shown on log-log plots are quadratic function curves. We can also calculate the lognormal parameters μ = 3.5870 and σ = 1.5527 for JDK5.0, and μ = 3.6865 and σ = 1.2254 for jBoss. Applying the equation (1), the probability density functions of the lognormal distribution models for these two systems are obtained:

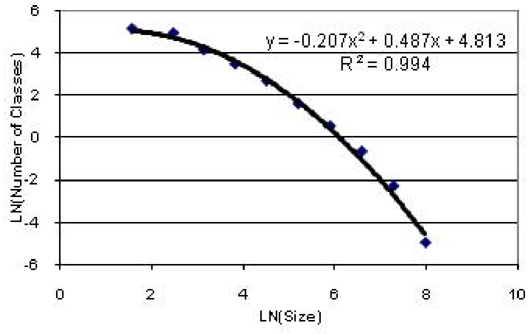
$$JDK5.0: f(x) = \frac{1}{1.5527x\sqrt{2\pi}} \exp\left(\frac{-(\ln x - 3.5870)^2}{2(1.5527)^2}\right)$$
$$= \frac{1}{3.8911x} \exp\left(\frac{-(\ln x - 3.5870)^2}{4.8218}\right)$$

and:

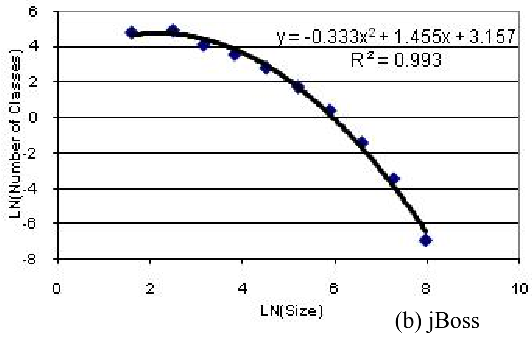
$$jBoss: f(x) = \frac{1}{1.2254x\sqrt{2\pi}} \exp\left(\frac{-(\ln x - 3.6865)^2}{2(1.2254)^2}\right)$$
$$= \frac{1}{3.0708x} \exp\left(\frac{-(\ln x - 3.6865)^2}{3}\right)$$

To confirm the lognormal distribution of class sizes, we draw the Probability-Probability (P-P) plot [12] using the SPSS statistical package. A P-P plot shows a variable's (in this case, the class size's) cumulative probability against the cumulative probability of the test distribution (in this case, the lognormal distribution). The straighter the line formed by the P-P plot, the more the variable's distribution conforms to the test distribution. Figure 4 shows the P-P plots for the JDK5.0 and the jBoss systems. The relationships between the expected and observed cumulative probabilities are close to linear and the slopes are close to unity. Therefore it is reasonable to conclude that the size distributions of these two systems are approximately lognormal.

We repeat the experiment for all studied Java systems and find that their class size distributions exhibit the similar behavior. The lognormal parameters for all the systems are summarized in Table 3. Furthermore, from Table 3 we can see that the μ and σ values for the studied Java systems do not vary much (with mean values 3.8277 and 1.3472; standard deviations 0.2303 and 0.1323 respectively). A t-test shows that with a 95% level of confidence, the mean μ and σ values of the population are within 3.8277 ± 0.1145 and 1.3472 ± 0.0658, respectively.



(a) JDK5.0

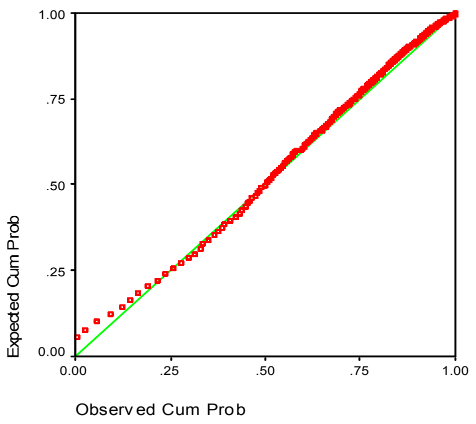


(b) jBoss

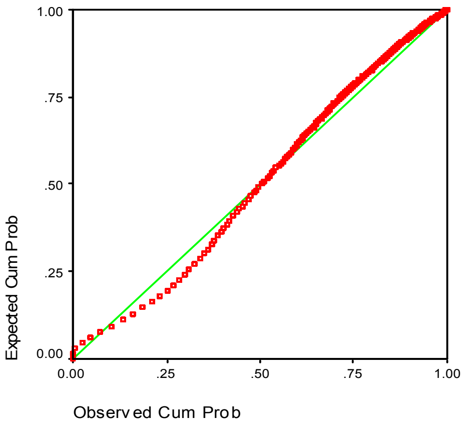
Figure 3. The size distribution of classes, plotted on a log-log scale

	β_2	β_1	β_0	R^2	μ	σ
Ant	-0.3698	1.9491	0.1264	0.9925	3.9874	1.1628
Zeus	-0.2238	0.6718	2.1930	0.9834	3.7350	1.4947
Cocoon	-0.3147	1.3427	1.1625	0.9934	3.7221	1.2605
Jena	-0.2538	0.8000	2.7076	0.9988	3.5461	1.4036
jetSpeed	-0.2786	1.2006	0.9699	0.9962	3.9494	1.3397
Tomcat 5	-0.3173	1.5642	0.8145	0.9988	4.0407	1.2553
Protege	-0.2225	0.5935	2.3487	0.9956	3.5809	1.4991
jBoss	-0.3330	1.4552	3.1571	0.9938	3.6865	1.2254
Tomcat 4	-0.3133	1.5982	0.0659	0.9977	4.1492	1.2637
JDK 1.4.0	-0.2054	0.4769	4.3002	0.9933	3.5952	1.5602
JDK 5.0	-0.2074	0.4879	4.8136	0.9942	3.5870	1.5527
Netbeans	-0.2996	1.3439	3.5641	0.9939	3.9117	1.2919
Eclipse	-0.2483	0.8892	4.5689	0.9979	3.8043	1.4190
ArgoUML	-0.2537	0.8753	2.2617	0.9884	3.6959	1.4039
myFaces	-0.3644	1.8244	-0.0271	0.9939	3.8754	1.1714
jEdit	-0.2518	1.2304	-0.275	0.9920	4.4289	1.4091
jHotdraw	-0.3826	1.8137	-0.9998	0.9906	3.6771	1.1432
Jetty	-0.2574	1.0211	0.5535	0.9781	3.9260	1.3937
Average (std dev.)					3.8277 (0.2303)	1.3472 (0.1323)

Table 3. The lognormal distribution parameters for the Java systems in Table 1



(a) JDK5.0



(b) jBoss

Figure 4. The lognormal P-P plots

4.3 The relationship between software size and the number of classes

Having understood the lognormal nature of class size distribution, we can then estimate sizes of large Java software systems as follows:

$$\text{Size} = \text{N} * \int \frac{x}{\sigma x\sqrt{2\pi}} \exp\left(\frac{-(\ln x - \mu)^2}{2\sigma^2}\right) dx \qquad (4)$$

where N is the number of classes. An analytic solution for this is:

$$\text{Size} = \text{N} * \exp(\mu + \sigma^2/2) \qquad (5)$$

Equation (5) shows that we can estimate software size solely based on the number of classes if the class sizes follow the lognormal distribution. To do this, we use the sample means of μ and σ (which are 3.8277 and 1.3472 respectively as shown in Table 3) to estimate the population means (i.e., point estimates), and use them to construct a generalized class size distribution model. We can then estimate the software size as follows:

$$\text{Size} = \text{N} * \exp(3.8277 + (1.3472)^2/2)$$

$$= \text{N} * 113.88 \qquad (6)$$

Equation (6) reveals the remarkably simple relationship between the size of a large Java software system and the number of classes – that the size is a product of N (number of class) and 113.88. The constant 113.88 is the expected average size of Java classes.

To evaluate the accuracy of the equation (6), we adopt two criteria suggested by Conte et al. [8], the mean-magnitude of relative error (MMRE) and the prediction at level L (Pred(L)). The MMRE is defined as the average of MRE: $MMRE = \frac{1}{n}\sum_{i=1}^{n} MRE_i$ and MRE is defined as $MRE_i = |N_i - \hat{N_i}|/N_i$, where N_i and $\hat{N}_i$ are the ith actual and estimated values, respectively. The Pred(L) is defined as the fraction of estimated data with MRE ≤ L. The value of MMRE is between 0 and 1. The commonly acceptable value is 0.25 (the smaller the value the better the estimation). The value of Pred(L) is between 0 and 1. Pred(0.25) ≥ 0.75 is considered acceptable (the larger the value the better the estimation). We also adopt these criteria in our evaluation.

Table 4 shows the evaluation of the equation (6) for all studied Java systems. The MMRE values is 18.01%, and the Pred(0.25) value is 83.33%. They all fall within the acceptable levels (MMRE <= 0.25 and Pred(0.25)>=0.75).

Table 4. Size estimation for the Java systems in Table 1

	Actual Size (LOC)	Estimated Size (LOC)	MRE	MRE<=25%?
Ant	104857	104545	0.29%	yes
Zeus	105121	92359	12.14%	yes
Cocoon	75653	95206	25.84%	no
Jena	121280	145429	19.91%	yes
jetSpeed	80947	78465	3.06%	yes
Tomcat 5	160745	143037	11.02%	yes
Protege	69159	79376	14.77%	yes
jBoss	587855	707215	20.30%	yes
JDK 1.4.0	506965	441639	12.88%	yes
JDK 5.0	834185	745366	10.65%	yes
Netbeans	1440911	1239961	13.95%	yes
Eclipse	1596401	1400650	12.26%	yes
Tomcat 4	94496	87007	7.92%	yes
ArgoUML	115267	137229	19.05%	yes
myFaces	63742	77099	20.95%	yes
jEdit	88435	44870	49.26%	no
jHotdraw	14611	22207	51.99%	no
Jetty	43578	35759	17.94%	yes
MMRE = 18.01% Pred(0.25) = 83.33%				

Table 5. Size estimation for the test dataset

	Actual Size (LOC)	Estimated Size (LOC)	MRE	MRE<=25%?
DrJava	59187	59674	0.82%	yes
JavaCC	15927	15829	0.61%	yes
jCV	11621	13097	12.69%	yes
Struts	47359	56145	18.55%	yes
Compiere	184255	135635	26.39%	no
James	27003	30748	13.87%	yes
Bluej	9740	5694	41.54%	no
JabRef	22041	20271	8.03%	yes
JMeter	84364	82907	1.73%	yes
InsECT	8103	6149	24.11%	yes
KBVT	32173	41453	28.84%	no
JBooks	12815	13210	3.08%	yes
MMRE = 15.02% Pred(0.25) = 75%				

To further test the equation (6), we also collected 12 different public domain Java systems including program editor, compiler, mail server, ERP software, web application framework, program analysis tool, and personal finance application. The descriptions of these systems are omitted here due to space constraints (Interested readers may obtain these systems from the Internet. For each system the latest version is chosen for this study). These systems have an average of 352 classes and 42.9K LOC. The test results are shown in Table 5. The MMRE is 15.02% and the Pred(0.25) is 75%. These values all fall within the acceptable levels, confirming that the equation (6) can estimate the sizes of large Java systems well.

4.4 Discussion of results

Over the years, many models have been proposed to estimate software size. For example, Halstead [18] gave a size estimation equation based on the number of operators and operands. Albrecht [1] proposed the function points method which measures software size based on a combination of program characteristics such as the number of external inputs and outputs. Recently Costagliola et al. [9] proposed the Class Point method that generalizes the Function Points method for size estimation of object-oriented systems. The total unadjusted class points are defined based on the number of classes and their complexity levels, which can be obtained from early design specification. Comparing with other software size estimation models, our model only requires one variable (the number of classes) thus is significantly simplified. Our estimation method can be applied once detailed design is complete (when the number of classes is known). Therefore, our method could be used to refine the early estimates such as those given by the Class Points method. We should also note that the Equation (6) works well when the class sizes follow the lognormal distribution. It could result in large variations for badly designed systems (those does not exhibit the small class phenomenon).

Some authors suggested that there is an optimal size for software modules and believed that modules of approximately optimal size are least likely to contain a fault. For example, Hatton [19] suggested an optimal range of 200-400 logical lines or 400-800 physical lines of code. Card and Galss [6] noted that many programming texts suggest limiting module size to 50 or 60 LOC. Bowen [4] also noted that military software development standards for module size specified an average of 100 and an absolute limit of 200 executable statements. However, El Emam et al. [14] showed that the theory of optimal class size has no sound theoretical or empirical basis. Our work shows that the distribution of Java class sizes has "long-tail" and follows the lognormal distribution. Furthermore, from Equation (6) we know that statistically, there is an expected average (mean) size of Java class regardless of the nature of the systems, which is about 114 LOC.

5. Related work

Although we are unaware of any prior work on the distribution of class sizes, there have been several studies of general disk file sizes for computer file systems and for the Internet. For example, Satyanarayanan examined 36,000 current files and 50,000 migrated files from a PDP-10 computer at CMU, and discovered that most files are very small and a hyperexponential is a good model for the file size distribution [27]. Crovella and Bestavros analyzed the WWW traffic and discovered that the Web files are currently more biased toward small files than are typical Unix files, and the distribution of Web file sizes is similar in sprit to Pareto distribution [10]. Humphrey suggested that the distribution of program size should be made to match normal distribution [16]. Baxter et al. [2] collected a corpus of Java software and analyzed the structural attributes of the programs (such as Number of Methods, Number of Fields, etc.). They discovered that the distribution of some attributes follow power-law while others do not. In our study, we perform empirical analysis of Java programs and discover the lognormal distribution of class sizes.

6. Conclusions

We studied a number of Java systems and found that many Java classes have only small sizes whereas a few classes have large size. Furthermore, we discovered that the sizes of classes are lognormally distributed. We then derived a novel size estimation model based on the lognormal nature of the class size distribution. Our model shows that the size of a large Java system can be estimated based on the number of classes with good accuracy.

We also 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.

As this study uses public domain software, we encourage other researchers to replicate our study to repeat, refute, or improve our results. In the future, we will further investigate the implications of the lognormal distribution of class sizes and the underlying principles. It is also interesting to explore the synergy between the proposed method and other models that can predict the software size at an earlier stage of the software development, especially the Class Points method. As module sizes correlate with efforts and quality attributes, another possible future work is to analyze the distribution of software cost and quality.

Large object-oriented software systems are complex systems. These systems are composed of a web of inter-related classes, which have different responsibilities and could be developed by different programmers at different time. It appears that these software development activities are arbitrary and there is no "law of physics" behind them [5]. We believe that our study reveals regularity behind the apparent chaotic process of software construction, and hope our research can contribute to a deep understanding of computer programming.

Acknowledgement

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

References

[1] Albrecht A.J., Measuring application development productivity, in *Proc. IBM Application Develop. Symp.*, Monterey, CA, Oct. 1979.

[2] Baxter, G. et al., Understanding the shape of Java software, *Proc. of the 21st ACM SIGPLAN conference Object-oriented programming languages, systems, and applications (OOPSLA)*, Oct 2006, Portland, USA.

[3] Boehm, B., *Software Engineering Economics,* Prentice Hall, 1981.

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

[5] Brooks, F.P., No silver bullet – essence and accidents of software engineering, *IEEE Computer* 20(4), April 1987.

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

[7] 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.

[8] Conte, S.D., Dunsmore, H.E. and Shen, V.Y., 1986. *Software Engineering Metrics and Models.* Menlo Park, CA: Benjamin-Cummings.

[9] Costagliola, G., Ferrucci, F., Tortora, G., and Vitiello, G., Class point: an approach for the size estimation of object-oriented systems, *IEEE Trans. Software Eng.*, vol. 31 (1), pp. 52-74, 2005.

[10] Crovella, M. and Bestavros, A., Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes, IEEE/ACM Transactions on Networking, Vol. 5 (6), pp. 835-846, 1997.

[11] Crow, E. L. and Shimizu, K. (Ed.). *Lognormal Distributions: Theory and Applications.* New York: Dekker, 1988.

[12] Devore, J., *Probability and Statistics for Engineering and the Sciences*, Duxbury Press, 1995.

[13] Douceur, J. and Bolosky, W., A Large Scale Study of File-System Contents, In *Proc. ACM SIGMETRICS Conference*, May 1999.

[14] El Emam, K., Benlarbi, S., Goel, N., Melo, W., Lounis, H., and Rai, S. N., The Optimal Class Size for Object-Oriented Software. *IEEE Trans. Softw. Eng.* 28, 5, May 2002, pp. 494-509.

[15] Gill, G. and Kemerer, C., Cyclomatic Complexity Density and Software Maintenance Productivity, *IEEE Trans. Software Eng.*, vol. 17, no. 12, pp. 1284-1295, Dec. 1991.

[16] Humphrey, W.S., *A Discipline for Software Engineering*, Addison-Wesley, 1995.

[17] Groth, B.H.A., The golden mean in the inheritance of size, *Science* 39, 581-584, 1914.

[18] Halstead, M.H., *Elements of Software Science*, Elsevier, North-Holland, 1977.

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

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

[21] Koru, A. G. and Liu, H., An investigation of the effect of module size on defect prediction using static measures. In *Proceedings of the 2005 Workshop on Predictor Models in Software Engineering* (PROMISE '05), May 2005.

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

[23] Li, H. F. and Cheung, W. K. 1987. An empirical study of software metrics. *IEEE Trans. Softw. Eng.* 13 (6), Jun 1987.

[24] Limpert, E., Stahel, W.A. and Abbt, M., Lognormal Distributions across the Sciences: Keys and Clues, *BioScience* 51, 341-352, 2001.

[25] O'Neal, M., An Empirical Study of Three Common Software Complexity Measures, *Proceedings of the 1993 ACM/SIGAPP symposium on applied computing*, March 1993, pp. 203-207.

[26] Putnam, L. and Myers, W., *Measures for Excellence: Reliable Software, on Time, Within Budget*, Prentice Hall, 1991.

[27] Satyanarayanan, M., A Study of File Sizes and Functional Lifetimes, *Proc. of the 8th ACM Symposium on Operating Systems Principles*, Dec. 1981, Pacific Grove, CA.

[28] Wilde, N., Matthews, P. and Huitt, R., Maintaining Object-Oriented Software, *IEEE Software*, Jan 1993, pp. 75-80.