An Empirical Study of Object Evolution Processes

Tetsuo Tamai
Graduate School of Arts and Sciences
University of Tokyo
3-8-1 Komaba, Meguro-ku
Tokyo 153, Japan
+81-3-5454-6847
tamai@graco.c.u-tokyo.ac.jp

Takako Nakatani
Graduate School of Arts and Sciences
University of Tokyo
3-8-1 Komaba, Meguro-ku
Tokyo 153, Japan
+81-3-5454-6844
tina@graco.c.u-tokyo.ac.jp

Abstract

A number of interesting phenomena can be observed when lifelong processes of object-oriented software are analyzed from the viewpoint of object evolution. This paper reports the results of empirical case studies and discuss about evolution patterns and laws of objects.

keywords Evolution process, object-oriented software, empirical study, statistical model

1 Introduction

We envy people who inherit legacy but pity those who inherit legacy software. But the stock of legacy software keeps on growing and old software systems are aging steadily. The eventual solution for this problem should be re-engineering, typically using the object-oriented technology. In re-engineering an aged system, we may have to look back its evolution history. What we should keep in mind at the same time is that object-oriented systems will also go through their own evolution processes and we should be prepared for that. However, the current wide variety of object-oriented technologies, from OO analysis and OO design to reuse and from design patterns to application frameworks are focusing on the methods of developing new systems and appear to care little about the long range process of evolution.

One of the pioneering works that introduced the word "evolution" to describe the changing process of systems over time is Belady & Lehman[2]. Their "laws" were derived from observation on OS/360's version history. Tamai & Torimitsu [8] examined evolution processes of application systems, especially focusing on system replacement strategies. Analogically speaking, their work treated evolution processes of not just single generation but over multiple generations.

The above two works are dealing evolution at the system level. For the object-oriented systems, evolution at the object level may have the same or even more significance. An object may live within multiple systems concurrently and may keep on living after the death of the system it belongs to, migrating into another system. There are a large population of objects living and evolving in the world, many of which are interacting and moving through internets. Using an analogy to biology again, the object level evolution can be compared to the DNA level evolution whereas the system level is compared to the species level evolution [4].

The objective of our research is to analyze evolution patterns of objects and construct an object evolution process model. When a sound model is successfully constructed, it will be beneficial not only by providing fundamental basis for understanding real evolution processes but also by supporting software engineers over long term object engineering processes through a software environment based on the model.

2 Approach

We intend to learn from biology not just for extracting analogies or metaphors but expecting to borrow rich models. Analyzing a population of objects requires the same observational or "scientific" viewpoint as analyzing a population of life. At the same time, we should not forget that objects are artifacts and their evolution is intrinsically caused by human engineering activities. Results of evolution process observation and analysis have to be useful for object engineering. Our standpoint is to combine these two perspectives, scientific and engineering.

2.1 Case Studies

At the first stage of our research, we have been taking an empirical approach, i.e. through conducting case studies. So far, we took the following three cases for our study.

1. **Heat Exchange Simulation System**
   - Description: a system to simulate heat flow and temperature distribution within a system composed of various heat devices.
   - No. of versions: 4
   - Development period: 8 months
   - No. of programmers: 1
   - Language: Visual Smalltalk
   - Size: 52 classes in Version 4
2. **Cash Receipts Transaction Management System**
   - Description: a system of a service company to manage money reception from customers by matching payments to invoices.
   - No. of versions: 4
   - Development period: 8 months
   - No. of programmers: 1
   - Language: Visual Smalltalk
   - Size: 62 classes in Version 4
3. **Securities Management System**
   - Description: a system to store information of securities possessed by a company: i.e. face value, purchased price, interest, and redemption, and support investment decisions.
   - No. of versions: 14
   - Development period: 3 months
   - No. of programmers: 4
   - Language: Visual Smalltalk
   - Size: 133 classes in Version 14

For each case, data of four or more versions were available. The meaning of "version" is different between the first two systems and the last. In the first two systems, each version was delivered to its customer and the customer returned feedback and new requirements for the next version. A version of the last system corresponds to a snapshot of the system being developed at a certain checkpoint during the development phase. All these systems are rather small but intended for practical use and the first two are actually used now.

2.2 Metrics

We defined a set of metrics and measured the series of version data of the three systems. There have been a number of proposals and discussions on metrics of object-oriented systems [3, 5]. We did not attempt to add totally new kinds of metrics to this stock. We classified metrics into three layers: system, class, and method and measured data including the number of classes or the depth of the class tree for the system layer, the number of methods, instance variables, and subclasses for the class and the number of lines of code for the method. Naturally, aggregated or averaged measures of the lower layer can also be measures for the upper layer, e.g. the total or average number of lines of code of over methods is a metric for a class.

What should be new in our approach are the way data are collected and analyzed:

1. Data are measured through a sequence of versions as time-series data. They are analyzed in time-series.
2. For collected basic statistics, not only means and variances are cared but also their distribution shapes are studied.
3. Not only static data but also dynamic data, e.g. numbers of messages sent or received between object instances, are collected. (But in this paper, we do not discuss about dynamic metrics.)

3 Observed Evolutional Patterns

Quantitative analysis using the above metrics was reinforced by qualitative analysis of tracing class structure and other semantic property changes. We also surveyed project documents and conducted interviews to the developers of the systems. All these results were consolidated in the efforts to relate the users' requirements change and the developers' design intension change to the system and object evolution processes.

We can summarize the major observations into the following four points [7, 6].

1. Fundamental statistics and distribution shapes are relatively stable over time.
2. On the other hand, some peculiar sample points with exceptionally large values exist. They may imply the existence of some design anomalies or exceptional design decisions.
3. Many of measured values have a trend of growing over time but the changes are not continuous; sometimes the growth is rapid and then the growth is slow. Periods of discontinuous change often indicate the occurrence of architectural level change.
4. A unique metric that characterizes class trees exists.

We will especially discuss the first and the the fourth of the above points in detail in the succeeding sections.

3.1 Stable Statistic Model

Some folklore data are known in terms of object system size. A. Aoki, who has developed one million lines of code in his long career as Smalltalk programmer [1], once said in

all systems or libraries he developed the average number of methods per class is 20, the average number of lines per method is 10 and thus the average number of lines per class is 200. Moreover, these values are also stable at the same level even for standard libraries supplied by vendors or other organizations. Maybe, this should not be called a folklore but be accepted as an observation based on solid data and experiences.

Interestingly, our measurement also confirmed this observation. Table 1 shows some basic statistics that are consistent with the above observation. Those values are roughly the same over time(versions) and over systems.

Table 1: Basic Statistics of Heat Exchange Simulation System

Methods per class

Version	1	2	3	4
Mean	15.1	19.4	19.7	18.3
Std Dev.	10.3	16.5	19.4	19.9

Lines per method

Version	1	2	3	4
Mean	8.1	8.5	9.1	9.4
Std Dev.	10.8	16.0	19.5	21.5

Fig. 1 and 2 show typical histograms of size data. It can be seen that not only the mean values are about the same among different versions or systems but also the distribution has a common shape. All graphs appear to imply there exists a common statistical model that explains these distributions.

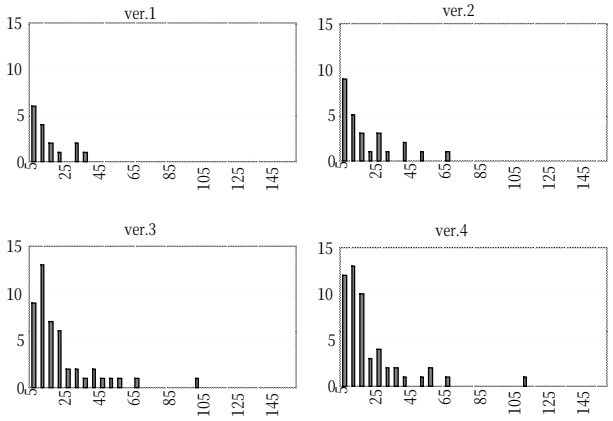

Figure 1: Histograms of #Methods per Class for Heat Exchange Simulation System

At the first glance, the Poisson distribution model appears to fit. However, some trial fitting soon revealed that the Poisson distribution would not fit well. We also tried the geometric distribution model but it did not fit either. Then, we focused our attention on the negative binomial distribution.

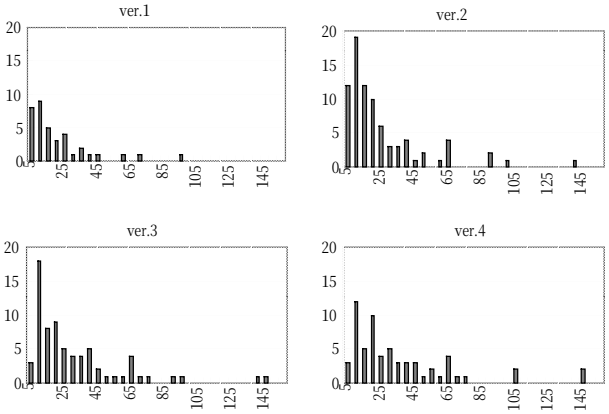

Figure 2: Histograms of #Methods per Class for Cash Receipts Transaction Management System

The reasons the negative binomial distribution is preferred are:

1. Its variance is larger than that of the Poisson distribution when the mean is equal. It is expected to fit better to the distributions like Fig. 1 and 2 that have long right tails.
2. The negative binomial distribution originally has the meaning of *length*, because it is derived as representing the distribution of the Bernoulli trial length when a certain event S occurs exactly a fixed number of times. Thus, it is expected to explain the code length distribution.

Fig. 3 shows the curve fitting of the negative distribution model to the Cash Receipts Transaction system LOC data. The fitting looks quite good but when the Pearson's test of goodness of fit is applied, the result does not support the fit.

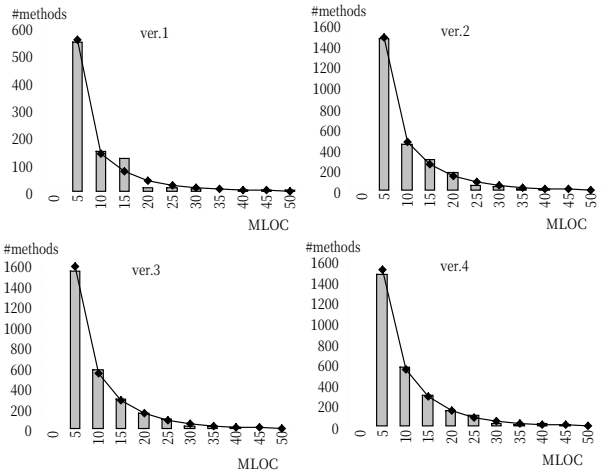

Figure 3: Fitting of Negative Binomial Distribution Model to #Lines per Method Distribution

To refine the model application, we decomposed the set of classes into subsets, each corresponding to a class tree. As

will be shown in the next section, classes belonging to the same class tree generally share some homogeneous properties, which distinguish them from classes of other class trees. Thus, it is expected that the model fitting applied to each set of classes belonging to a tree may give better results. Actually, it turned out that the hypothesis the negative binomial distribution model fits cannot be rejected by level 5% X test for most of the trees of the three case systems.

If the negative binomial distribution model fits method size or class size distributions, then the process of length of code being determined may be interpreted as a stochastic process as follows. Programming activities of a programmer is observed by a third person. To the observer's eyes, the programming looks like repetitive random selections of statements (lines) or methods. When a defined number of statements (or methods) that have specific properties are chosen, it will complete a method (or a class). The probability that a randomly chosen statement/method has this property is constant.

The distribution pattern like this is not new to the software community. The number of lines of code per module should have had a similar shape of distribution before the object-oriented era. However, there seems to have been no previous works that attempted statistical distribution model fitting.

The probability function of the negative binomial distribution is given by

$$p(x) = \left(\begin{array}{c} x-1 \\ k-1 \end{array} \right) p^k (1-p)^{x-k}. \qquad (1)$$

Thus, the distribution is determined by two parameters, p and k. Estimated values of these two parameters should have good information, richer than the pair of means and variance. Fig.4 plots these estimated parameter values for class trees of the Heat Exchange Simulation System, where arrows indicate directions of version advancement. Similar graphs are given for the Cash Receipt Transaction Management System in Fig. 5 and for the Securities Management System in Fig. 6.

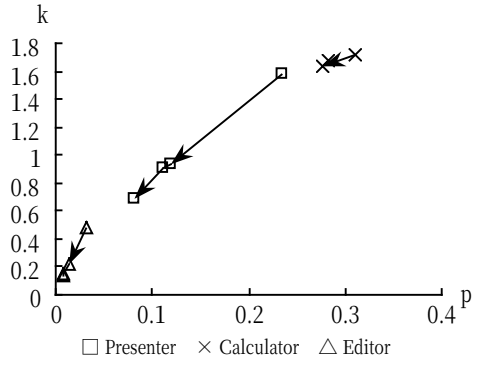

Figure 4: Trace of parameters (p,k) for Simulation System

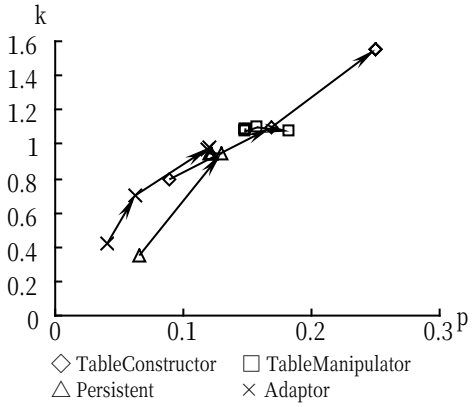

Figure 5: Trace of parameters (p,k) for Cash System

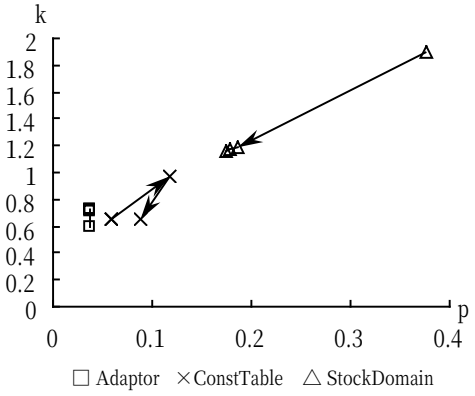

Figure 6: Trace of parameters (p,k) for Securities System

These graphs suggest the following points.

1. There exists a strong linear correlation between the two parameters. The interpretation of this apparently surprising phenomenon can be straightforward, i.e. the mean is constant over versions, because the mean of the negative binomial distribution (1) is given by k/p.

2. As the result of the above linear relation, when the value of k gets larger, also the value of p gets larger and when the former gets smaller, so does the latter. The larger k and p may be interpreted as more patterned coding, stronger convention or uniformly organized programs and the smaller k and p may imply more room for programming decisions.

3. As seen by the arrow direction, k and p are getting smaller as the version proceeds in the Heat Simulation System, getting larger in the Cash Management System and not conclusive in the Securities System. Based on the implication of larger/smaller values of k and p stated above, these trends seem to explain the fact that the Heat Simulation System took the process

of adding new modules according to the users' requirements change, while the Cash Management System followed the process of restructuring the system by the software designer.

3.2 Class Tree Characteristics

As expected, the number of lines and the number of methods per class have a strong correlation. Actually, when we conducted statistical testing, the correlation between the number of lines and the number of methods of a class was validated for each of the three systems.

The scattered diagram of these two values over classes of Heat Exchange Simulation System is illustrated at the top-left of Fig. 7. A conspicuous pattern of this diagram is the existence of multiple lines. Actually each of these lines corresponds to a set of classes that belong to the same class tree, as the other three diagrams show.

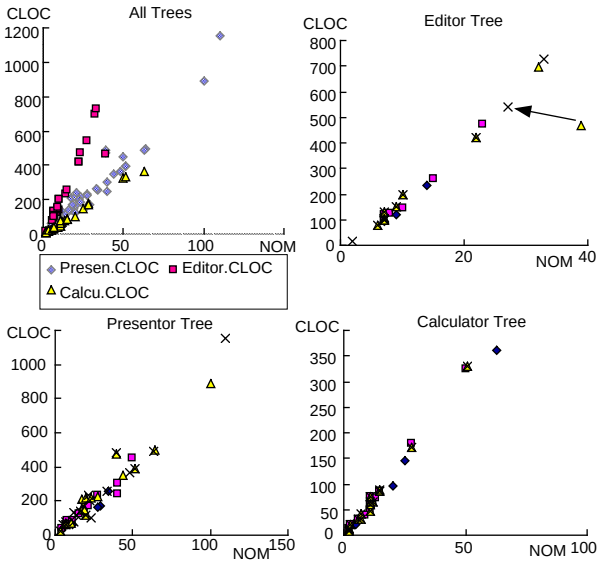

Figure 7: Scattered Diagrams of #Lines vs. #Methods per Class for Heat Exchange Simulation System

The most peculiar phenomenon can be observed in the diagram of Editor Tree (top-right). Here, the arrow represents a move of one class from Version 3 to Version 4. The class had an exceptional value of #Lines/#Methods in Version 3 but it regressed to the "normal" value in Version 4. This phenomenon suggests that the characteristic value of #Lines/#Methods or the regression coefficient between the two metrics for each class tree has a strong constraining power.

To explore the significance of this characteristics of class trees, we set up three hypothesis and conducted statistical testing.

1. *The regression coefficients between different class trees are different.*
   This hypothesis was statistically validated by rejecting the null hypothesis of equality between two coefficients of different trees. It was validated in all the three systems.

2. *The regression coefficient of a class tree is stable over evolution.*
   This was validated by the fact that the null hypothesis of equality between two coefficients of different versions cannot be rejected.

3. *The regression coefficient of a class tree is stable over different programmers.*
   In our data, there is only one case where classes of a single tree were divided into two and developed by different programmers. Data of that case were tested and the result showed that the difference between the two is not statistically significant.

These findings suggest that there exist some kind of design criteria for each class tree that designers implicitly assume. Monitoring this metric will give good feedback to the designers.

4 Conclusion and Future Work

We conducted empirical studies on three object-oriented systems' evolution processes and found several interesting evolution patterns and properties. We plan to collect more cases and validate and extend the findings. Based on such knowledge, we intend to construct a general object evolution model, which can be used for developing long-term evolution design environment.

Acknowledgements The authors would like to thank Atsushi Tomoeda and Harumi Matsuda for their substantive work of collecting and analyzing key data for this research.

References

[1] Aoki, A.: *Smalltalk Textbook*, http://www.sra.co.jp /people/aoki/SmalltalkTextbook/index.html

[2] Belady, L. A. and Lehman, M. M.: A Model of Large Program Development, *IBM Systems Journal*, Vol. 15, No. 3 (1976), pp. 225–252.

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

[4] Dawkins, R.: *The Selfish Gene*, Oxford University Press, 1976.

[5] Lorenz, M. and Kidd, J. *Object-Oriented Software Metrics*, Prentice Hall (1994).

[6] Nakatani, T., Tamai, T., Tomoeda, A. and Matsuda, H.: Towards Constructing a Class Evolution Model, *Proceedings of Asia-Pacific Software Engineering Conference*, December 1997, Hong Kong, pp. 131–138.

[7] Nakatani, T. and Tamai, T.: Evolutional Characteristics of Class Inheritance Trees, *Proceedings of the International Symposium on Future Software Technology (ISFST-97)*, October 1997, Xiamen, China, pp. 44 – 51.

[8] Tamai, T. and Torimitsu, Y.: Software Lifetime and its Evolution Process over Generations, *Proc. Conference on Software Maintenance – 1992*, Orlando, Florida, November 1992, pp. 63–69.