J. Software Engineering & Applications, 2010, 3: 58-64
doi:10.4236/jsea.2010.31007 Published Online January 2010 (http://www.SciRP.org/journal/jsea)

Application of Design Patterns in Process of Large-Scale Software Evolving

Wei WANG, Hai ZHAO, Hui LI, Peng LI, Dong YAO, Zheng LIU, Bo LI, Shuang YU, Hong LIU, Kunzhan YANG

Information Science and Engineering, Northeastern University, Shenyang, China.
Email: wangweiwin1@163.com, zhhai@neuera.com

Received September 15th, 2009; revised September 29th, 2009; accepted October 13th, 2009.

ABSTRACT

To search for the Design Patterns' influence on the software, the paper abstracts the feature models of 9 kinds of classic exiting design patterns among the 23 kinds and describes the features with algorithm language. Meanwhile, searching for the specific structure features in the network, the paper designs 9 matching algorithms of the 9 kinds design patterns mentioned above to research on the structure of the design patterns in the software network. At last, the paper analyzes the evolving trends of the software scale and the application frequency of the 9 kinds of design patterns as the software evolves, and search for the rules how these design patterns are applied into 4 kinds of typical software.

Keywords: *Design Pattern, Feature Model, Software Network, Evolving Trends*

1. Introduction

With the increasing of system scale and complexity, the reliability and maintainability are highly required. Design Patterns describe common problems that frequently occur in the process of the object oriented software development and give the resolutions of these problems. Implication Design Patterns into software development can enhance the open, compatibility, stability and extensibility of software, which makes the development and maintenance much easier [1].

Does Design Patterns improve the quality of software efficiently? Can Design Patterns be widely used? How these patterns are composed reasonably? What is the reasonable scope of the ratio of the times patterns used and software scale? Facing these questions, it becomes an urgent issue to quantify and measure these patterns when they are used in software in the process of software designing.

If software continues to evolve, it needs to be reorganized [2–4]. This is called refactoring and the frames occur during this time. A better understanding of Design Patterns will reduce the time that is spent on refactoring. Looking into how design patterns are implied into some software organized fairly well and evolved continually can direct the design of software system positively. Therefore, it is significant to find out the evolving trends that design patterns are implied into software designing [5,6].

With the help of open-source software *Doxygen*, the object oriented software is abstract into XML. Then with the help of *XmlParse* which is developed in my lab, collect the nodes and edges from XML and abstract the software into software network.

The topology of software system can be represented by topology of network [7–9]. In the network, nodes represent the component of software and the edges represent the relation between nodes. Complex networks theory is applied into software system which mainly refers to open source software, reverse engineering that get the class graph and network model of the source code is taken to get and analyze the organization structure [10,11]. The abstraction process is shown as Figure 1.

2. Abstracting Process

According to the definition of the design patterns, the paper abstracts the structure features and expresses in mathematical language, which is used for designing and realizing the matching algorithm [12–15].

In the software network, nodes present the abstract data; edges present the relation between the nodes. Nodes can be classified into class, struct and interface; edges can be classified into inherit, usage, static, template and

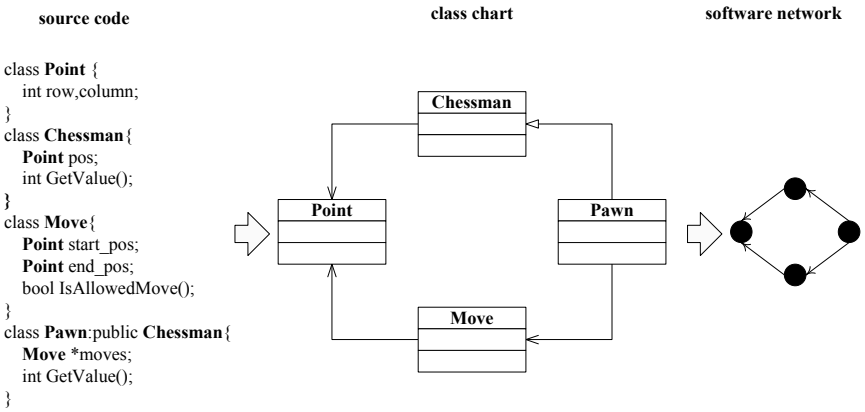

Figure 1. The extraction of the software network

friend. Since software network is digraph and the relation between data are classified into inherit and aggregation, the degree of the nodes are classified in-degree and out-degree of inherit and usage.

2.1 Flyweight

Flyweight supports a number of fine-grained objects with sharing method. The frame of *Flyweight* is as shown in Figure 2.

Abstract the main features of *Flyweight*. Known from the frame of the *Flyweight*, classes in *Flyweight* can be mainly classified into *Flyweight* and FlyweightFactory. FlyweightFactory is an abstract class and the template of FlyweightFactory is defined in *Flyweight*. That is there is one-to-many relation between *Flyweight* and Flyweight-Factory. *Flyweight* at least has two subclasses. The software network of the *Flyweight* is as shown in Figure 3.

All of the nodes in *Flyweight* are class. There is an edge with double value, template and usage, between node *Flyweight* and node FlyweightFactory, and the edge is from node FlyweightFactory to node *Flyweight*. The inherit in-degree of node Flyweitht is more than 1.

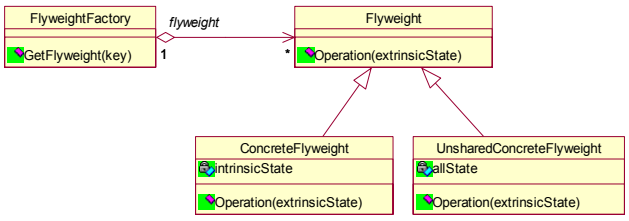

Figure 2. The frame of the Flyweight Pattern

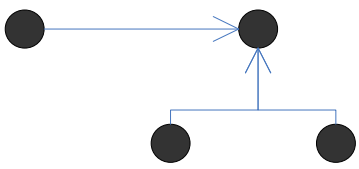

Figure 3. The software network of the Flyweight Pattern

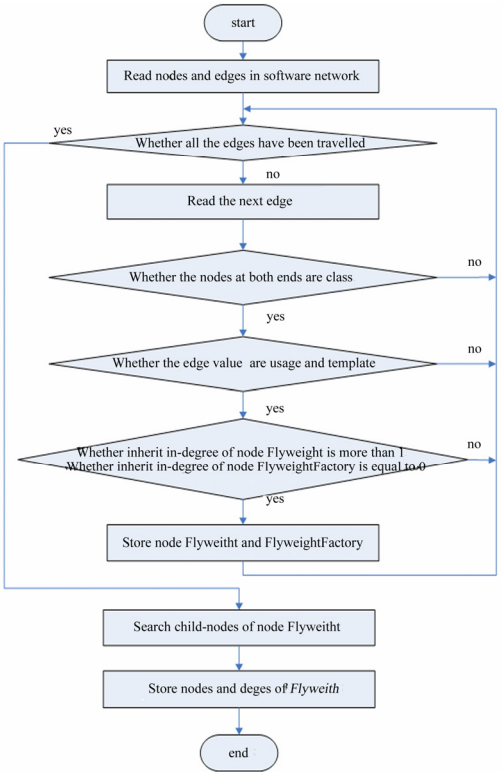

Figure 4. The flow chart of the matching algorithm of the Flyweight Pattern

The key judgment standards are concluded as follows:

(1) The relation between node FlyweightFactory and node Flyweitht are merely usage and template.

(2) The inherit in-degree of node Flyweitht is more than 1.

Figure 4 is the flow chart of *Flyweitht*.

2.2 The Other Eight Design Patterns

The abstracting processes of the other eight design patterns are similar to Flywe*itht*. The software network of the nine design patterns are shown in Figure 5. The abstracting standards of the nine design patterns are shown in Table 1.

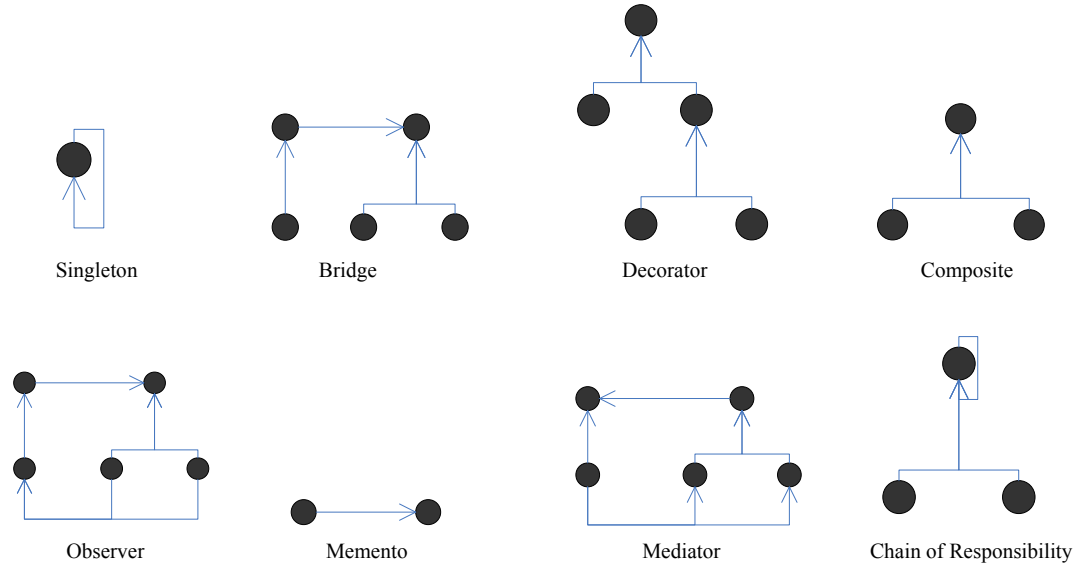

Figure 5. The software network of the nine design patterns

Table 1. The abstraction standards of the nine design patterns

design pattern	the abstraction standards
Singleton	The starting node is the ending node. The starting node of the edge is its ending node.
Bridge	The relation between node Abstract and node Implementor is merely usage. The inherit in-degree of node Implementor is more than 1.
Decorator	The relation between node Decorator and node Component are usage and template. The inherit in-degree of node Component is more than 1.
Composite	The relation between node Component and node Composite are inherit, usage and template. The inherit in-degree of node Component is more than 1.
Flyweight	The relation between node FlyweightFactory and node Flyweitht are merely usage and template. The inherit in-degree of node Flyweitht is more than 1.
Observer	The relation between node Observer and node Subject are usage and template. There is one-to-one usage-relation between child nodes of node Observer and child nodes of node Subject.
Memento	The edge values are friend and usage. The starting node and ending node are class.
Mediator	The relation between node Mediator and node College is merely usage. There is one-to-one usage-relation between child nodes of node Mediator and child nodes of node College.
Chain of Responsibility	The edge value is merely usage. The starting node of the edge is its ending node.

3. The Application and Analysis of Design Patterns in Software Evolving

The paper makes research on four kinds of open-source software: text-processing software *abiword*, image-proce ssing software *blender*, web browser software *firefox*, and language-development software *eclipse*. There are more than one version can be used in these four widely used software, for this reason these software are taken as examples.

3.1 How Software Scale Changes in Software Evolving

Since there is linear relationship between number of nodes and number of edges, the software scale can be represented by the number of nodes. These results can be received: during the software evolving, number of nodes in *abiword* changes smoothly, that in *blender* and *eclipse* increases a little, and that in *firefox* increase first and decrease at last. Through checking software files, we find that the cores of *abiword*, *blender* and *eclipse* hardly change, while the core of *firefox* changes from the versi-on of 3.0 to the version of 3.0.7.

3.2 The Application of Design Patterns in Software Evolving

The evolving trends of the implication of design patterns in *abiword*, *blender*, *firefox* and *eclipse* are shown as Figure 6, Figure 7, Figure 8, and Figure 9. The abscissa is the design patterns being used and the ordinate is times

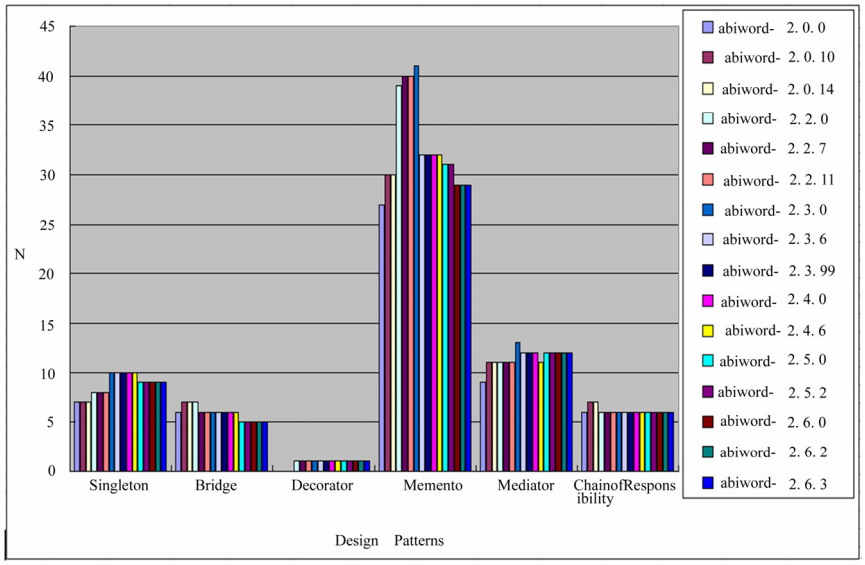

Figure 6. The changes of the application of the Design Patterns in the evolution of abiword

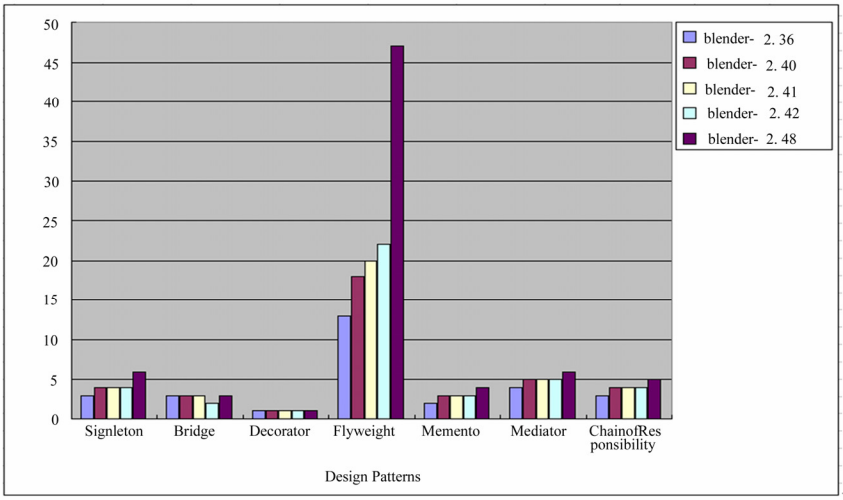

Figure 7. The changes of the application of the Design Patterns in the evolution of blender

N of the design patterns being used.

As Figure 6 shows, six patterns are used in *abiword*: *Singleton, Bridge, Decorator, Memento, Mediator* and *Chain of Responsibility*. The times of these six patterns being used goes up first, and then goes down, at last goes smoothly. Meanwhile, the average using times of each pattern in all versions are far away different. The average using times of *Singleton, Bridge, Decorator* and *Chain of Responsibility* are no more than ten, that of *Mediator* is slightly more than ten, while being different from the other patterns, the average using times of *Memento* is up to 30.

As Figure 7 shows, seven patterns are used in *blender*: *Singleton, Bridge, Decorator, Flyweight*, *Memento, Mediator* and *Chain of Responsibility*. And except *Flyweight*, the using times of the other six patterns are no more than 5. The using times of *Flyweight* goes up along with the software involving, and in the latest versions it goes up so quick that it goes far away from the usual linear growth mode. The other patterns are merely unchanged.

As Figure 8 shows, seven patterns are used in *firefox*: *Singleton, Bridge, Decorator, Flyweight*, *Memento, Mediator* and *Chain of Responsibility*. The using times of *Bridge* and *Decorator* are merely unchanged, and those of *Singleton*, *Memento*, *Mediator* and *Chain of Responsibility* show fluctuations that increase first, and then decrease, at last increase, but the amplitudes is very small. The using times of *Flyweight* increases to a large extent in the latest two versions.

As Figure 9 shows, seven patterns are used in *eclipse*: *Singleton, Bridge, Decorator, Composite*, *Flyweight,*

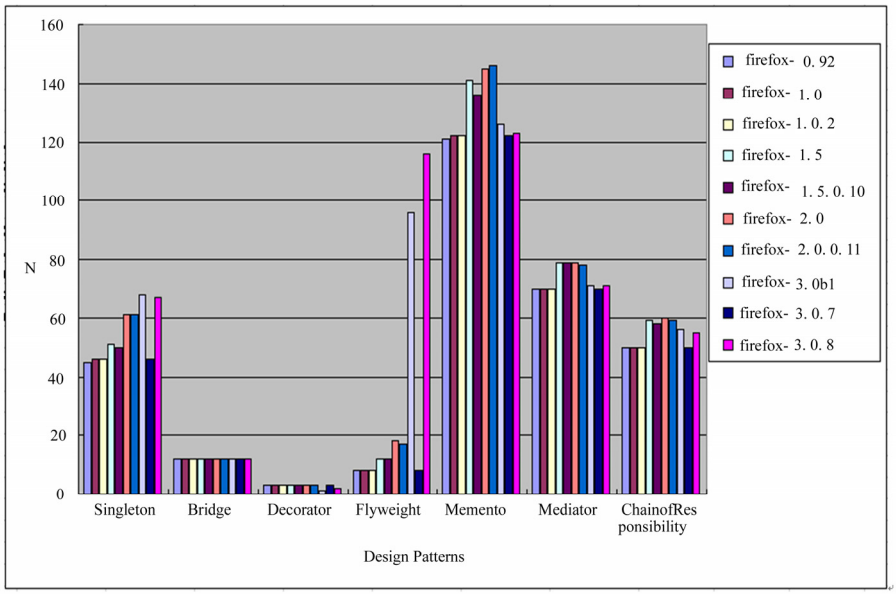

Figure 8. The changes of the application of the Design Patterns in the evolution of firefox

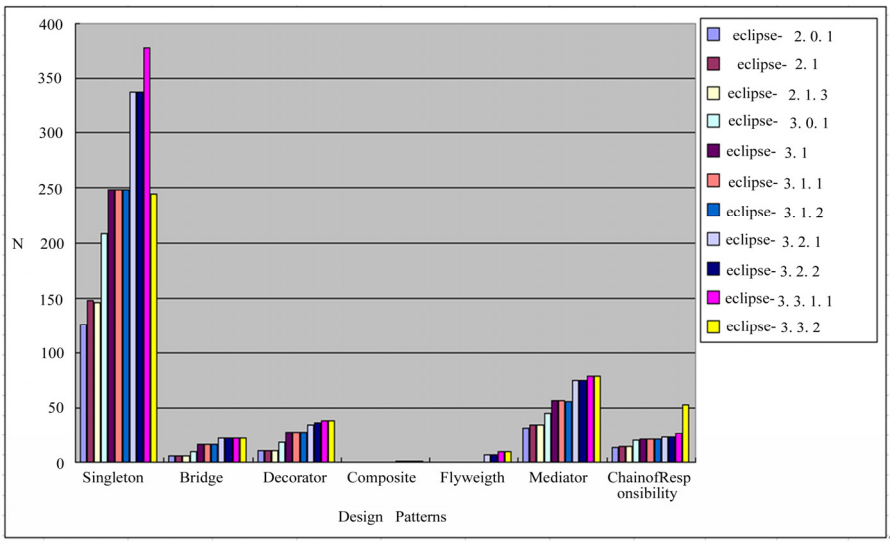

Figure 9. The changes of the application of the Design Patterns in the evolution of eclipse

Mediator and *Chain of Responsibility*. Except *Singleton*, the average using times of the other six patterns are no more than 50. The using times of *Singleton* increase first, and then decrease, but those of the other patterns increase to some extent continuously.

3.3 The Analysis on How Design Patterns are Used in Software Evolving

Known from Table 2 which shows the average using times of each design pattern in the chosen software, the average using times of *Memento* comes up to 32.71 which takes up of 49.7 percent of the sum of the using times of the six patterns used in *abiword*. *Memento* is used to catch the state of an object and store it outside of the object which is up for restoring the object in the future. Text-processing software must remember the state at any moment that can help the users restore the state when it is necessary. For these reasons, *Memento* is used far more frequently than the other patterns in *abiword*.

Known from Table 2, the average using times of *Flyweight* comes up to 24 which takes up of 54.5 percent of the sum of the using times of the seven patterns used in *blender*. Since the subclasses of class *Flyweight* are divided into shared data field and unshared data field, *Flyweight* deals with common graphs and exceptional graphs very well. A large amount of common graphs and exceptional graphs are provided for users in image-processing software. For these reasons, *Flyweight* is used

Table 2. The average using times of each design pattern in the software

Design Patterns	abiword	blender	firefox	eclipse
Singleton	8.75	4.2	54.1	242.8
Bridge	5.875	2.8	12	14.91
Decorator	0.8125	1	2.7	25.64
Composite	0	0	0	0.364
Flyweight	0	24	30.3	3.09
Observer	0	0	0	0
Memento	32.71	3	130.4	0
Mediator	11.5	5	73.7	56.82
Chain of Responsibility	6.125	4	54.7	22.73

far more frequently than the other patterns in *blender*.

Known from Table 2: the using times of *Singleton* comes up to 54.1 which takes up of 15.1 percent of the sum of the average using times of the seven patterns used in *firefox*; the using times of *Memento* comes up to 130.4 which takes up of 36.4 percent of the sum of the average using times of the seven patterns used in *firefox*; the using times of *Mediator* comes up to 73.7 which takes up of 20.6 percent of the sum of the average using times of the seven patterns used in *firefox*; the using times of *Chain of Responsibility* comes up to 54.7 which takes up of 15.3 percent of the sum of the average using times of the seven patterns used in *firefox*. *Mediator* can deal with the communication among the objects implicitly. *Chain of Responsibility* can make these requests a line and pass these requests along the line till the final processing. *Memento* can store the reply data being received for client processing. At the same time, web browser takes C/S model, so users will sent and receive large amount of data request and reply continuously. For these reasons, *Memento, Mediator, Chain of Responsibility* and *Chain of Responsibility* are used far more frequently than the other patterns in *firefox*.

Known from Table 2, the average using times of *Singleton* comes up to 242.8 which takes up of 66.3 percent of the sum of the using times of the seven patterns used in *eclipse*. The developers have to call the system functions through the interfaces provided by *eclipse* when they use *eclipse* and these system interfaces permit being called but not changed, while *Singleton* can prevent the change made by developers when they use these interfaces. For these reasons, *Singleton* is used far more frequently than the other patterns in *eclipse*.

Definition 1: the using times of some design patterns takes up more than 50 percent of the sum of the using times of all the patterns in software, then this pattern is key pattern.

According to the Definition 1, the key pattern of software is similar to the key in database. Since the key pattern decides the main function of software, it can be some kind of symbol of the software. Based on the Definition 1, the key pattern of *abiword* is *Memento*; the key pattern of *blender* is *Flyweight*; the key patterns of *firefox* are *Memento* and *Mediator*; the key pattern of *eclipse* is *Singleton*.

As Figure 6, Figure 7, Figure 8, and Figure 9 show, the using times of *Flyweight* appears abnormal changes in the last versions of *firefox* and *blender*. Through referring to the white books of *firefox* and *blender*, we find that the cores of *firefox* and *blender* changed too where these abnormal changes happens [16].

4. Conclusions

According what mentioned above, the application and rule of software in the process of evolving is abstracted as follows:

The application of design patterns is changed along with the change of software core and the using times of key pattern of software increases first, then decreases, at last swing around a certain number.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Elements of reusable object-oriented software [M]," Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[2] M. M. Lehma, and J. F. Rmail, "Software evolution and software evolution processes [J]," Annals of Software Engineering, Vol. 14, No. 1, pp. 275–309, 2002.

[3] B. Dougherty, J. White, C. Thompson, and D. C. Schmidt, "Automating hardware and software evolution analysis [A]," Engineering of Computer Based Systems, ECBS 2009. 16th Annual IEEE International Conference and Workshop on the [C], Vol. 35, No. 5, pp. 265–274, 2009.

[4] S. N. Dorogovtsev, and J. F. Mendes, "Scaling properties of scale-free evolving networks: continuous approach [J]," Physical Review E, Vol. 63, No. 5, pp. 56125.

[5] N. Zhao, T. Li, L. L. Yang, Y. Yu, F. Dai, and W. Zhang, "The resource optimization of software evolution processes [A]," Advanced Computer Control, ICACC'09, International Conference on [C], pp. 332–336, 2009.

[6] B. Behm, "Some future trends and implications for systems and software engineering processes [J]," Systems Engineering, Vol. 9, No. 1, pp. 1–19, 2006.

[7] L. Paolo, B. Andrea, and D. G. Felicita, "A decomposition-based modeling framework for complex systems [J]," IEEE Transaction on Reliability, Vol. 58, No. 1, pp. 20–33, 2009.

[8] Y. Ma, and K. A. He, "Complexity metrics set for large-scale object-oriented software systems, in proceedings of 6th international conference on computer and information technology [J]," pp. 189–189, 2006.

[9] K. Madhavi, and A. A. A. Rao, "Framework for visualizing model-driven software evolution [A], Advance Computing Conference IACC'09 IEEE International [C]," pp. 1628–1633, 2009.

[10] S. Valverde, and R. V. Sole, "Network notifs in computational graphs: a case study in software architecture [J]," Physical Review E, Vol. 72, No. 2, pp. 26107, 2005.

[11] C. R. Myers, "Software systems as complex networks: structure, function, and evolvability of software collaboration graphs [J]," Physical Review E, Vol. 68, No. 4, pp. 46116, 2003.

[12] A. Potanin, et al. "Scale-free geometry in OO programs [J]," Communications of ACM, Vol. 48, No. 5, pp. 99–103, 2005.

[13] S. Meyers, "Effective C++ (3rd Edition) [M]," Addison-Wesley Professional, pp. 10–50, 2005.

[14] C. A. Conley, and L. Sproull, "Easier said than done: an empirical investigation of software design and quality in open source software development [A]," System Sciences, HICSS'09 42nd Hawaii International Conference on [C], pp. 1–10, 2009.

[15] W. Lian, R. G. Dromey, and D. Kirk, "Software Engineering and Scale-free Networks [J]," IEEE Transactions on Systems, Vol. 39, No. 3, pp. 648–657, 2009.

[16] M. M. Lehma, and J. F. Rmail, "Software evolution background, theory, practice [J]," Information Processing Letters, Vol. 88, No. 1/2, pp. 33–44, 2003.

