December 1998

Object-Oriented Approach as a Potential Silver Bullet

Changsoo Sohn
Southern Illinois University at Carbondale

Apiwan Dejnaronk
Southern Illinois University at Carbondale

Object-Oriented Approach as a Potential Silver Bullet

Changsoo Sohn
Apiwan Dejnaronk
Department of Management
Southern Illinois University at Carbondale

Abstract

Software developers are seeking a silver bullet, if exists, to overcome the software crisis. The silver bullet is expected to enable them to deliver software on time with high quality and low cost. The fundamental cause of software crisis is the nature of software itself. The difficulties derived from the nature of software are complexity, conformity, invisibility, and changeability. In an attempt to search for the silver bullet, this study analyzes five current software engineering approaches: prototyping, spiral model, object oriented approach, cleanroom, and fourth generation language. Through comparison of those approaches based on the four difficulties, this study proposes that an object-oriented approach is a candidate for a silver bullet.

Introduction

Technological advances make users request high quality software with low cost. High quality software cannot help being complex and large. Large and complex software needs high cost. Subsequently, its delivery is delayed. High demand and competitive environment push software developers to hurriedly deliver a software product. If the product is released without adequate quality, more often than not it causes errors in an application system and software becomes more expensive to users than they expect. This situation is referred to "*software crisis*" (Blazer, 1975; Gibbs, 1994; Jeff, 1995).

Thus, software engineers attempt to overcome the software crisis and develop a guideline for the efficient software development. The fundamental causes of the software problems are "essential" difficulties rather than "accidental" difficulties as Brooks (1987) said. If a certain software engineering approach can eliminate essential difficulties, the approach can be a silver bullet. The four essential difficulties that are inherent in the nature of software are complexity, conformity, invisibility, and changeability (Brooks, 1987; Kraut & Streeter, 1995). The software crisis is resulted from the interplay among those four difficulties that are not mutually exclusive.

Therefore, a research question is "is there any current software engineering approach that provides the powerful capability to remove the four essential difficulties of software? If so, why?"

The Nature of Software Difficulties

Complexity: Since software is composed of different parts which none of them are alike, the combination of those parts makes software complex (Brooks, 1987). The complexity is common and inevitable because the software links several conditions multiplicatively (Brooks, 1987). The interaction of the multiple conditions in software leads to the complicate conceptual expressions. Williams (1995) defines complexity as understandability of a piece of software. Thus, whether or not complexity is substantial in software engineering, complexity makes software difficult to interpret, understand, and control.

Conformity: Conformity refers to the concept that all software components must work in harmony. External requirements and interfaces are major drivers of the conformity in software (Mays, 1994). Conformity and complexity are interrelated each other. The rationale here is that software is complex because many parts are involved. Then, to avoid the conflict between those parts, interfaces must be conformed.

Changeability: Software is different from other commodities in that it can be changed after manufacturing. Since the conceptual design of software depends on individuals judgment, different designers with different perspectives may alter the software design (Mays, 1994). Brooks (1987) presents two causes of software change; users' needs and advances in hardware technology.

Invisibility: According to Brooks (1987), the structures of software are highly abstract and not simply explained by a few diagrams. Existing conceptual tools are not capable enough to capture all aspects of those invisible structures. In addition, software cannot be evaluated until it is executed. Because of these facts, we cannot evaluate the performance of a software system until the system is completed (Williams, 1995). Meanwhile, invisibility makes difficult to communicate between users and designers because they do not have a standard model that they can agree on.

Analysis of Software Engineering Approaches

Prototyping: A prototype creates a trial version before the real product is built. Prototyping cannot eliminate the difficulties in software design due to four reasons. First, complexity is not simply removed by the demonstration of a prototype. With or without a prototype, a small software system is always simple. Prototyping is not related with reducing software complexity. Second, conformity interrelated with the software complexity. As prototyping cannot reduce complexity, it cannot eliminate conformity and simplify the interface between those components. Third, the needs for changes are common problems in every software project. The use of prototyping cannot prevent this uncertainty that occurs over time such as change in software specifications. Finally, although the difficulty of invisibility is partially addressed under this approach, a prototype as a tool to increase the understanding between users and designers (Winograd, 1995) does not remove software invisibility.

***Spiral model**:* A spiral model approach is a combination of classical and prototyping approaches with an additional feature called "risk analysis" which is a tool for assessing the possibility of continuing a software project (Pressman, 1992). Although customer evaluation process of spiral model partially focuses on the difficulties in software design, it cannot remove the complexity, changeability, or invisibility of software. In addition, it does not enforce the conformity of the interfaces between different software components. What makes matters worse, the risk analysis not only omits the inherent difficulties of software but also alters costs of the project. Since the costs are reestimated whenever the analysis is conducted, it may be a problem to a fixed-budget project (Pressman, 1992). In addition, prototyping and spiral model approaches conduct a preventive action as oppose to a corrective action. Therefore, it is unlikely that the spiral model approach can be the silver bullet candidate.

Object-oriented Approach: An object-oriented approach is based on two underlying principles: encapsulation and inheritance. First, *encapsulation* or *information hiding* means a design decision is hidden or encapsulated in each software component (Coad & Yourdon, 1991). The interface between encapsulated objects are designed to reveal as little information as possible and to allow limited access to their objects. Unlike the traditional approach where functions are defined separately from a given data structure, the object-oriented approach allows designers to put data together with its operations. As a result, interdependencies between objects are minimized because an existing object can be modified or a new object can be added without affecting the rest. Encapsulation also eliminates duplicate code, which leads to the decrease in the project size and the ease of developing and maintaining the software system (Fichman & Kemerer, 1993). The second principle is *inheritance* that refers to a relationship between classes where one is a parent and the other is its child. This child inherits all attributes that belong to its parent(s). Inheritance supports the concept of reusability which focuses on using existing components to build a new software system rather than creating the whole system from scratch (Korson & McGregor, 1990; LaLonde & Pugh, 1990). The reusability also minimizes the amount of code that needs to be generated. Both characteristics may eliminate the difficulties that are essentially embedded in software design. Inheritance and encapsulation automatically enforce the conformity of interfaces between different parts of software. Since the objects barely interact to each other, the conflict between the interfaces of two or more objects is minimized. On the other hand, an object-oriented approach translates user requirements into abstract data types and operations that can be visualized and easy to understand. Finally, object-oriented software engineering can accommodate the changes by allowing designers to reuse the previously generated components and loosely coupled and strongly cohesive objects facilitate the changeability. From the above reasons, we believe that object-oriented software engineering can eliminate those four inherent difficulties of software.

Cleanroom: Cleanroom software engineering integrates statistical quality control as a tool for improving quality of a software product. Mills et al. (1987) explain that a cleanroom approach is proactive (i.e., preventing a problem) rather than reactive (i.e., correcting a problem). The strict statistical quality control must be performed to detect an error (Mays, 1994). If the error is found, it can be corrected at that moment. Like other approaches, the cleanroom approach only partially addresses the difficulties of software design. The quality control techniques can capture only the errors that occur by chance like human errors. Cleanroom approach can solve the “accidental” software problems, not “essential” problems. Thus, this approach cannot remove the inherent difficulties of software.

Fourth Generation Languages (4GL): The major breakthrough introduced by 4GL is to generate codes automatically (Pressman, 1992). Like other software engineering, the 4GL approach cannot be the silver bullet. Although 4GL enables developers to define software specifications at the high level of abstraction (Pressman, 1992), it only removes complexity in software implementation rather than software design. The design of software is still complicate and it becomes more complex as project size increases. There is a debate that the tools for generating source code create difficulties to a user who is not familiar with how to translate requirements into design specifications (Brooks, 1987). In other words, lack of basic programming knowledge, one may experience a difficulty in using 4GL.

Conclusion

From the above five software engineering approaches, we argue that object oriented approach is a potential silver bullet because object-oriented approach may eliminate four difficulties that software has. In fact, the approach itself still has limitations (e.g., difficult-to-use, and training required) that may increase the accidental difficulties instead of eliminating them (Mays, 1994). However, unlike a conventional (structural) technique, an object orientation totally changes the way we analyze a problem and implement a solution. In the future, it is likely that an object-oriented approach will be a candidate for a silver bullet that solves the problems of software crisis. There is still a lot of improvement that needs to be done in this area and an attention is needed to focus on this approach.

References

Blazer, R., *Imprecise Program Specification Report*, ISI/RR-75-36, Information Science Institute, Dec. 1975.

Brooks, F.P. “No Silver Bullet,” *Computer*, April, 1987, 10-19.

Coad, P. and Yourdon, E., *OOA--objected -oriented Analysis*, 2nd Ed., Prentice Hall, Englewood Cliffs: NJ, 1991.

Fichman, R.G. and Kemerer, C.F., "Adoption of Software Engineering Process Innovations: The Case of Object Orientation," *Sloan Management Review*, Winter, 1993, 7-22.

Jeff, B., *Web BGTCS*, Jul. 1995, [URL:http://www.magicnet.net/~;bryson/swcrisis.html].

Gibbs, W.W., "Software's Chronic Crisis," *Scientific American*, 271(3), Sep. 1994, 86-95.

Korson, T. and McGregor, J.D., "Understanding Object-Oriented: A Unifying Paradigm," *Communication of the ACM*, 33(9), 1990, 76-96.

Kraut, R.E. and Streeler, L.A., "Coordination in Software Development," *Communication of the ACM*, 38(6), 1995, 69-81.

LaLonde, W.R. and Pugh, J.R., *Inside Smalltalk*: vol. 1, Prentice Hall, Englewood Cliff: NJ, 1990.

Mays, R.G. “Forging a silver bullet from the essence of software,” *IBM Systems Journal*, 33(1), 1994, 20-45.

Mills, H.D., Dyer, M., and Linger, R.C., "Cleanroom Software Engineering," *IEEE Software*, Sept. 1987, 19-25.

Pressman, R.S., *A Manager’s Guide to Software Engineering*, NY: McGraw-Hill, 1992.

Williams, N., “The Software Development Process,” Feb. 1995, [URL:http://osiris.sund.ac.uk/~calnwi/html/sdp/intro.html].

Winograd, T., "From Programming Environments to Environments of Designing," *Communication of the ACM*, 38(6), 1995, 65-74.