A Purview of Systematic Software Development

Peraphon Sophatsathit
Advanced Virtual and Intelligent Computing (AVIC) Research Center
Department of Mathematics, Faculty of Science
Chulalongkorn University
Phyathai Rd., Patumwan, Bangkok, 10330 THAILAND
email: Peraphon.S@chula.ac.th

Abstract

Studies of software development process have been widely exercised during the past few decades. The underlying principles, albeit extensively well-known, are surprisingly mismanaged. This article describes some intriguing analogies between the human body and software in an attempt to draw some concise similarities. While human body is undeniably one of the most sophisticated natural "being" ever exists on earth, its impersonating software invention remains to be a far cry from the intended purpose. A purview of systematic approach is summarized to offer a grandiose perspective and taste of how software comes into "being."

keywords: software component, human body, software development, plug-and-play.

1 Introduction

Software in many respects can be regarded as the heart and soul of computer systems, while its hardware counterpart serves as the physical being. This impersonation of software and hardware mimics the phisiology of human body. Unlike computer, the human body is the utmost intelligent life form whose body and soul are intricately interoperate at *subsystem*, *organ*, down to *micro-organism* level. Such intricate granularity of dichotomy renders it one-of-a-kind that no human invention can ever imitate. As technology advances, software research and development, in lieu of its unceasingly efforts, is striving to close the gap between itself and the human paradigm.

This article presents a facet of such systematic endeavor that emphasizes on fundamental software engineering principles, namely, repeatable, economic, and safe. The resulting software products will therefore entail high quality, reliability, yet cost effective to operate as the soul of our futuristic thinking machine. Consequently, a few unconventional configurations and designs will be concocted to revolutionize the existing computing paradigms.

2 An Architectural Framework

Every cell is, in essence, a simple autonomous entity that can function to sustain its survival needs. As cells bind together to form organs, body, and eventually the entire physical being, they interoperate through complex interconnecting networks of control. The blueprint of such a marvelous configuration is believed to be pre-established by DNA. Exactly how it comes into play is still unknown. Despite numerous efforts from various scientific disciplines, researchers are still unable to unlock the secret of nature. Fortunately, software is a well-defined artifact that possesses clear objectives from the outset, though imperfect and susceptible to error in practice. This is an inherent nature of all human inventions that attempt to transform complex abstractions into simple objects. This intangible "soul" of computer operates from the primitive bits to advanced algorithms and artificial neural networks that can *learn* as they go. The complexity grows as the architectural ladder moves away from hardware. To allevi-

ate and cope with such predicament, software researchers and developers alike laboriously strive to refine this invention by mimicking nature. A cell is envisioned as a component, an organ as a package, a body part as a subsystem, and the body as a system. This basic building block scheme lends itself to a wide array of innovative development paradigms, e.g., reuse, plug-and-play, and COTS. Such proliferation of flexible selections does not come without a high price, i.e., component standardization and operational overhead.

At the heart of central control lies intelligent algorithmic methods that regulate the functions of control components or the "brain." Figure 1 illustrates the structure of such component-based software systems. These components are *plugged* onto the control support back plane, which is regulated by the *brain* or central control functions known as the *operationg system*. All system capabilities are performed by this software subsystem or "organ." This analogy continues to instill the evolutionary R&D of software architectural framework.

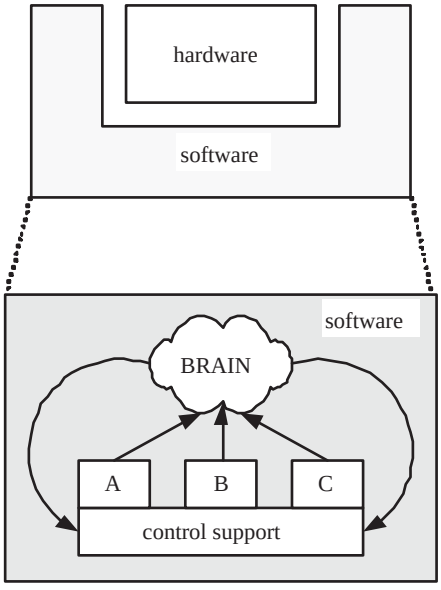

Figure 1: Structure of component-based software systems.

3 Configuration and Design of Software

Based on human analogy, the architectural framework rests on basic *cellular* building blocks or components that serve as the fundamental development units. Each component performs only one function, thereby permitting component replacement, integration, and most important of all, reuse. As the level of abstraction increases, components are combined to form packages through standard user interfaces. This is achieved by means of well-defined interface specifications [1], interconnecting association, and control discipline that form component standard. This is akin to file operation package that is associated with file management sub-systems. Invocation and processing control are administered by the operating systems.

A number of architectural abstraction conflicts may arise according to conventional layered configuration. One prominent and well-known problem is component abstraction hierarchy. The depth of inheritance tree plays a major role in such conflicts. Moreover, the overhead incurred from information transport among layers (e.g., polymorphic operations, multiple inheritance, etc.) may offset the abstraction so preserved, thus rendering abstraction gains to be an unworthy tradeoff.

Another problem that must be rectified (and ultimately eliminated) is software defect. This short-fall has plagued software development since its inception. Modern testing methods, in particular, fault prediction techniques as exemplified by [2] herald state-of-the-art endeavor that is set forth toward achieving fault-free software components and products. A concise discussion of some relevant design aspects will be described to support the *systematic philosophy* of software design principles and practices.

Bearing the aforementioned component-based architecture in mind, the design issues thus rely on many proven principles, namely, abstraction, information hiding, encapsulation, and modularity. These principles will work well only if the underlying support mechanisms are standardized and reliable. In other words, there must be standard interfacing and operating protocols that enable "plug-and-play" to operate smoothly, having as few incompatibilities as possible.

The paradigm of component-based software development has brought about the notion of OOA and OOD for

decades. Modern researches have embarked on hybrid approaches by introducing RAD, AOSD, Agile, or XP, to name a few. These approaches offer flexible means to enhance the design process as development life cycle shrinks, whereby accelerating software product release to suit the fast-changing needs. A simplified design process is given in Figure 2 to demonstrate its iterative, repeatable, methodical, and effective procedures. As such, various design aspects can be verified regularly as development progresses to ensure their correctness, consistency, and compliance with engineering principles.

Some revolutionary design philosophies usually encompass test design during the component (or unit) design stage. Many modern testing techniques are employed at component level to serve the purpose, e.g., finite state machine, architectural description language, hardware description language, etc. Specific test provisions are arranged to exercise particular test cases [3]. As the number of components grows, so does design complexity. The use of design configuration, document control, and design archive permits large-scale software development, where teamwork is essential. Such work flow entails *virtual* cooperation in many software development organizations, thereby overcoming physical barriers for the software teams. Components can be check-in/check-out during different testing stages (unit, integration, and system testing) according to project schedule. Each unit development folder is subsequently integrated from related components to form phase-deliverables. This integration process is performed repeatedly throughout the development life cycle which eventually becomes the final software product.

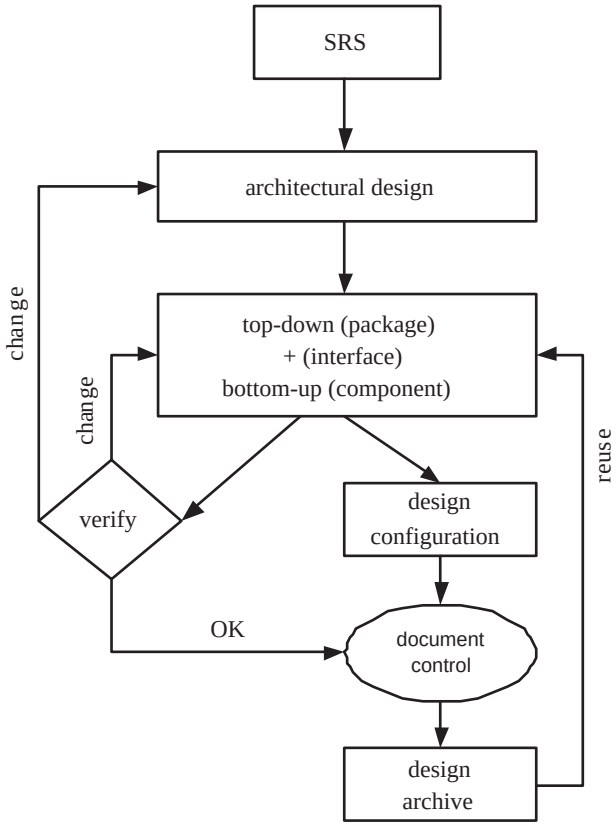

Figure 2: A systematic design process.

4 Ad hoc and Reuse Development

The advent and proliferation of the Internet inevitably creates a paradigm shift on software products and their development process. Internet-based software does not undergo the same development life cycle as its large-scale application counterpart. Turnaround time of each release is swift. Obsolescence is no longer the issue, but rather new capabilities and nifty features are added. Careful considerations on development with/for reuse become essential. Configuration management is undoubtedly called for as complexity grows and time-to-release shrinks. Conventional project team and philosophy are too cumbersome to apply. Instead Agile/XP environments are often proved to be effective. As a consequence, some formality of documentation management may be loosen to ease the burden and keep the project in perspective.

In order to speed up the development process, developers gradually opt for COTS approach to “assemble” software packages and products. This configuration requires standard interface that glues those components together. The problem is no existing de-facto standard that can support the very notion of “plug-and-play” as its hardware counterpart. Consequently, components do not function together as a unified combination and hence induce defects. This, in turn, creates a formidable diagnostic chore or in an infamous programming jargon *debugging*. Such

a process is usually tedious and painstaking, whereby resulting in *left-over* bugs to disrupt normal operations of the desired functionalities.

As the body routines are thrown off from their normal functions, the person suffers and becomes ill. Such malfunctions of one or more organs may trigger a ripple effects on the rest of the body. Software operations are no exception. Regardless of the extent on how careful component encapsulation is done, side-effects seem to be norm and the degree with which components are interrelated increases. This analogy remains to be true as the imitation gets closer. The question is not how to solve the problem, but rather systematically administer it so that when a symptom (defect) pops up, it is cured (fixed) effectively and effortlessly.

5 Conclusion

The human body functions in a systematic, organized, and well-regulated manner. When certain organs malfunction, the body becomes ill. By the same token, software is supposedly performed its functions normally. When software encounters a glitch, it may run into different unpredictable transitions and result in an undesirable behavior.

While human body is built by nature, medical and biological researches attempt to find ways to overcome the inherent short-comings. Software, on the other hand, is built by human. Any defective concoctions can thus be modified, adjusted, or even replaced, as long as it is done systematically. The challenge of this human invention is to continuously improve the development process and the final product. The success of quality software product rests primarily on the aforementioned systematic development and management process. Insofar as human intervention is concerned, the so called "human error" prevails. Researchers are hard at work to get rid of the human factor. As such, the future of software development is likely to be carried out automatically by machines, or perhaps, by the software itself. Who knows?

References

[1] Sathit Nakkrasae and Peraphon Sophatsathit, "A Formal Approach for Specification and Classification of Software Components." *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, in cooperation with ACM-SIGSOFT*, Ischia, Italy, July 15-19, 2002, pp. 773-780.

[2] Atchara Mahaweerawat, Peraphon Sophatsathit, Chidchanok Lursinsap, and Petr Musilek, "Fault Prediction in Object-Oriented Software Using Neural Network Techniques." *Proceedings of the InTech Conference, 2004*, Houston, TX, U.S.A., Dec 2-4, 2004, pp. 27-34.

[3] Sittisak Sai-ngern, Chidchanok Lursinsap, and Peraphon Sophatsathit, "Test Shape Generation of Dynamically Linked Structures." *Proceedings of the ECTI Annual Conference (ECTI-CON 2004)*, Pattaya, Thailand, May 13-14, 2004, pp. 330-333.