A Universal Model of Software Engineering

Peraphon Sophatsathit
Advanced Virtual and Intelligent Computing (AVIC) Center
Department of Mathematics and Computer Science, Faculty of Science
Chulalongkorn University, Bangkok, Thailand.
Peraphon.S@chula.ac.th

Abstract
This paper proposes a universal model that governs the general theory of software engineering and complies with three engineering principles, namely, repeatable, economic, and safety principle. The main idea is to create core components that serve as the basic building blocks to build working software components. Each working software component is made up of core component strings by algorithmically concatenating core components based on a predetermined encoding formula. Such a formulation process imitates the DNA double-stranded helices that make up cells and organs, while maintains a flat structural reference to the core components. These working software components can then grow systematically by splitting their component, mimicking the cell splitting process. Hence, they are interoperable, interchangeable, and standardizable through the common core components. This posits avoidance of efforts on study, analysis, and development of theories and implementation to support any extraneous software artifacts. The benefits of the proposed universal model are conducive toward longevity and unification of software engineering theory and practice.

Keywords: engineering principles, repeatable, economic, safety, core component string, encoding formula.

1. Introduction

From the early era of The Mythical Man-Month [1] introduced by Brooks, software engineering took off in different manner and directions. On the one hand, software engineering in practice seemed to be in chaos without any supporting theory. Several small and adhoc software were developed to serve the rapid growing demands but failed subsequently, despite the wealth of theoretical disciplinary ground work laid down by many experienced forerunners. On the other hand, good practice kept on improving the discipline. Theoretical supports were abundant. Johnson et al. [2] emphasized the existence of theories that supported software engineering, but have yet been realized. After all, it still fell short of applying the supporting theories to software products.

If we look back to software engineering principles, one fact that might help establish a general theoretical framework is the *engineering* discipline. Fundamentally, science makes theoretical discoveries to unlock the secret of the universe, while engineering exploits the theories by putting them into production. Parnas [3] precisely rationalizes between Computer Science Programs and Software Engineering Programs. What sets software engineering apart, in my opinion, is the engineering discipline which encompasses three vital principles governing the development and operations of software. They are (1) repeatable, (2) economic, and (3) safety principle. Software engineering must undergo a systematic process that can not only be repeated, but also performed by any qualified parties to produce the same results. One possible means is good documentation that serves the same purpose as the building blueprint of Civil Engineering. Moreover, the results so obtained must be created in accordance with engineering economic principles, i.e., affordable, cost effective, yet operable by efficient algorithms. Finally, the work product itself must guard against failure, harness the detrimental harmful output for the safety of general users.

The above process framework is not any new discovery or innovation in software engineering. Part of the first principle has been achieved by the supporting algorithms of the software. The questionable part is implementation of the algorithms to be a software product. There are myriad of computer languages, tools, techniques, platforms, implementation specific issues, restrictions, and regulations to adhere. The second principle is a problematic stage of software development where inaccurate cost estimation is still struggling to meet the allotted budget, let alone project failure. The last principle culminates the software process and product through formal approach that unfortunately receives little attention by the software community owing to lack of training and verification rigor.

This paper will present a universal model of software engineering that is built on the above engineering principles. The underlying theorems that form the building block of the proposed model are established in Section 2. A flexible reference framework is described in Section 3, along with a simple example to illustrate the applicability of the proposed model. Conclusion and recommended future work are given in Section 4.

2. Principles of proposed theorems

The proposed software engineering theorems are drawn from the basic computer system configuration consisting of hardware and software. From a religious standpoint, the human body is made up of four substances, namely, earth, water, air, and fire. What makes it work is the soul that controls this living contraption. Hardware is no different from the body that must be equipped with software to run and control it. This analogy is depicted in Figure 1.

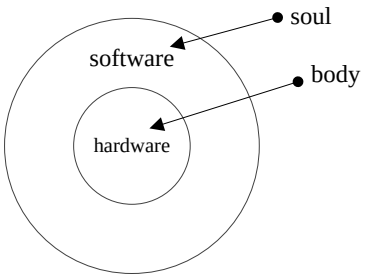

Figure 1. The human body VS the computer system

Bearing the above example in mind, the intricacy of architectural relationships to be drawn from this analogy begins at the highest level of abstraction of the body. The basic five senses, namely, sight, smell, touch, taste, and hearing, functioned by the input organs are the natural marvel that no artificially built devices can substitute. An even more intricacy is their integrated operations that consolidate all forms of input senses for instantaneous processing by the brain, subconscious mind, or instinct. The result is then interpreted, responded, or acted on accordingly. All these activities are completed in splits of a second. A further research deep into the structural construct of these organs unveils one common building block, i.e., the DNA which is fundamentally composed of A, T, C, and G nucleotides. The functional aspect of each organ has long been discovered and established. What remains to be explored is the dynamicity of the control function operated by the brain. Hardware and software exhibit similar imitation of the above natural wonders. From the highest level of hardware abstraction, the VLSI are made up of the basic AND, OR, and NOT gates. It is operational in the presence of the software counterpart. At the highest level of software abstraction, 5GL is on the horizon while the class/object implementation is already realized. Deep down into the basis of software lies the 1’s and 0’s which preserve the linkage with hardware on-off. One important conclusion that can be inferred from this analogy instills the first theorem of the proposed universal model of software engineering.

Theorem 1: *There are simple building blocks that software can be architected to govern the universal usage.*

The theorem can be proven descriptively by examples from three architectural aspects of the simple building blocks, hereafter referred to as core components, namely, structure, function, and behavior. As evident abound in archeology, many creatures were extinct and many evolved over the years. Conjectures on what the world centuries and millennia from now would look like have been attempted. One proposition that stood out from such predictions was the survival of cockroaches [4]. The fact that they are biologically simple renders them survive all adversaries as they could adapt to endure various environmental transformations. By the same token, hardware and software can mimic such endurance by adhering to their simple constructs that are architected by the corresponding basic building blocks, i.e., gates and bits. As such, they can maintain their close relationship to each other. In the meantime, the intermediate and high level constructs fade in and out as architecture evolves. For example, vacuum tube, CRT, on the one hand, and APL, HIPO chart, on the other hand, became rarities and were replaced by integrated circuits and high level software design patterns or languages. If we were to apply Theorem 1 to these progressive levels of abstraction, we could accumulatively create working building blocks in the same manner as DNA that made up the cells and organs. Subsequent discoveries and theoretical findings can be derived from the same ground work, thereby freeing existing dependencies on intermediate architectural constructs, algorithms, and limitations.

Building on hardware gates and software bits, software building blocks will take the form of simple functional typing, namely, integer (X), character (Y), note (A), binary (B), and reserved (Z) that characterize the core components. The X and Y are well known prevalent types, whereas A denotes sound, B denotes bin, and Z is reserved for smell, touch, and taste encoding. The corresponding methods defined to operate on these core components are X: ADD, SUB; Y: order; A: intensity, frequency, spectrum; B: BADD, AND, OR, NOT, XOR, shift, branch, complement; and Z: reserved. Examples of software components to be built in that order are numeric, string, wave, bitmap, and ETC, expressed in the form of a core component string based on an encoding scheme. However, an architectural limit on the component string hierarchy that makes up a software component must be imposed to confine the height of component hierarchy. Such a limitation, at the first glance, could result in larger combinations of core components to represent working software components. This will incur considerable computation loads to decode the composition of software component in the same manner as matching strings of nucleotides to identify the DNA. From the DNA standpoint, it is created by a natural process from unknown origin and conditions which make the DNA matching process time-consuming. On the contrary, creation of software components by the proposed approach can avoid this problem because their formulation starts from a known encoding process. This process is described as "a step by step process which is a general solution to a problem in a finite number of steps[1]", which is known as an algorithm. It is therefore enumerable according to the predetermined encoding formula, yet guaranteed to complete in finite steps unless it is intentionally designed to be irreversible. The outcome of algorithmic software component construction is the core component string that constitutes individual software component, regardless of how long the string extends. Note that the algorithmic formulation must conform to software standards. For example, to create a web page, industrial or de jure UI conventions and component standards must be followed to

[1] Coined by Richard Whitehouse, Mark Mckain, John Tvedt, and Peraphon Sophatsathit

derive the representative component string combinations. However, establishment of such standards is beyond the scope of this proposal.

By virtue of algorithmic creation of software components, the behavior can also be methodically designed and laid out in concert with the structural and functional aspects. This is accomplished through the repetitive design of core component methods to build more complex operations. For example, to design a multiply operation for a new executable binary component, we first choose core component B. Subtraction operation is then defined using complement and BADD methods. Finally, Booth Multiplication Algorithm is employed to complete the task.

The above limitations on architectural constructs transpire another theorem that governs the growth of software component under the proposed universal model.

Theorem 2: *Software components must maintain a simple construct to keep a direct reference to the simple building blocks.*

In modern software systems, software component hierarchy grows which in turn causes complexity to increase considerably. This is the culprit of prohibitively unmanageable software products as oppose to the hardware counterpart, whose plug-and-play capability has gone a long way. For this reason, the proposed universal model sets out to maintain a flat structure of component strings so as to keep a direct reference path from working software components to the core components. Such a design philosophy imposes component division mandate as the hierarchy grows. The argument again goes back to principles of cell split to create larger and complex organs. Software component growth or reconstruction will mimic this splitting process to keep the algorithmic formulation simple and straightforward, yet operable as a whole.

The splitting process proceeds in the same manner as cloning. Each software component duplicates itself to forming identical copies, except some component specific information. The actual splitting process can be implemented by the well-established UNIX *fork* and *exec* technique or any similar approaches. Details will be elucidated in the next section.

The above descriptive proof merely argues by exemplifying the software component construction process from core components proposed by Theorem 1 based on the three architectural aspects. The benefit of Theorem 2 is to maintain its durability as software system and technology evolve. More complicated components can be built by concatenating these core component strings. Until the Von Neumann architecture is obsolete, the proposed universal model hopefully will survive as cockroaches have prevailed.

3. A flexible reference framework

Through the core components, an example of working software component is built under a flexible reference framework as suggested in Figure 2.

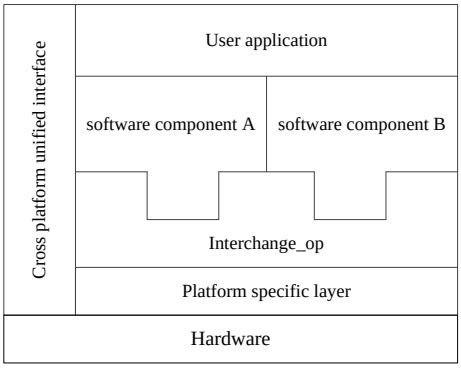

Figure 2. A flexible reference framework

At the lowest level of the flexible reference architecture, *platform specific layer* must be created to shield off hardware dependency, rendering working *software components* to be created, bearing virtually little hardware knowledge in mind. The *interchange_op* layer lies between these two layers to permit various exchanges among different working component configurations. The top layer is *user application* which operates at the highest level of transparency. All these arrangements are interconnected with a *cross platform unified interface* as all components involved are built based on the core component strings. At any rate, this configuration can be rearranged to suit the underlying system architecture and application platform.

A closer look into the software component composition reveals the aforementioned core component strings. For example, to create a graphic working component, one possible configuration can be established as shown in Figure 3.

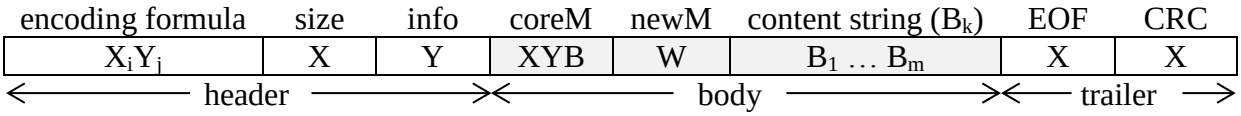

Figure 3. A possible graphic working component configuration

The first field of the header section holds the *encoding formula* represented by X_i and Y_j having the form say, (10)(011), where X_i denotes the 1's and 0's and Y_j denotes the '(', ')' characters. The *size* and *info* fields hold typical information pertaining to a software component. The *content string* B_k, $1 \leq k \leq m$ of the body section makes up the bitmap creating from binary core components based on the encoding formula. This precludes duplications since encoding of another component can be set up using a different formula, while an identical copy created from component splitting process will contain different information in the *info* field. The *coreM* field holds the predefined methods associated with each core component. The *newM* field keeps all newly derived methods W that are necessary for the new graphic working component definition. The last two fields denote *EOF* and *CRC*. Additional security measure may also be incorporated. The rationale behind incorporating all these extraneous "accessories" is to ensure that each component is self-contained and preserves both supporting theorems. Since each software component can be viewed as a cell, it is therefore subject to splitting if there is a need to grow. Having carried all relating information along allows it to split wherever and whenever is necessary. One challenging part of working component splitting is the actual implementation of these methods which could be done by embodying them in the body of the component or maintaining reference pointers to the core component repository. This is an implementation specific issue. As pointed out

earlier, there will be no computation burden to process the excessive length of core component string by virtue of known enumerable encoding scheme.

The proposed theorems which support the above flexible reference framework bring up two essential constructional and operational characteristics of software components.

Corollary 1: *All software can be constructed to be interchangeable and interoperate among their components.*

Corollary 2: *Software engineering artifacts can be standardized in compliance with the underlying architecture, techniques, and operations.*

It is apparent that the proposed universal model abides by the aforementioned three engineering principles. The algorithmic creation of core component strings can be repeated if need be. In fact, the entire software component structure is systematically repeatable. The economic principle is achievable by moving toward machine learning, whereby automatic component generation can be carried out. Finally, the safety principle is straightforwardly implementable since the proposed configuration can be formally verified through the algorithmic procedure. The by-product of this software engineering process is traceable and measurable software product quality.

A probable effective application of this simplistic component architecture is component generation which makes all operating software components dispensable. This implies that non-persistent software components can be immediately discarded after use and new replacement components are automatically generated. On the other hand, persistent software components, after being archived, can also be dispensed. This is similar to old skin cells die and come off, while new cells are generated. Consequently, memory occupation can be considerably reduced under the proposal universal model.

4. Conclusion and future work

Software engineering artifacts have been created in abundant over the years, forming stacks of intermediaries, as well as the supporting theories, that hardly receive full attention. One undeniable fact about software engineering is its flexibility which can be tailored by the designers in any form, set up, and operation. The intricacy of such limitless extension of abundant software artifacts entails a labyrinth of unmanageable treatments to new software development. This paper takes a different perspective to investigate on natural beings that inspire how fundamental building blocks can be assembled to create an extremely complicated and intelligent life form. This life form in turn is able to adaptively evolve itself to sustain various environmental transformations. It is envisioned that software engineering can mimic this ideal natural paradigm to arrive at a universal model encompassing engineering principles and practices. The flat structure of component strings will permit direct jumps from primitive core components to highly complex constructs, thereby enabling subsequent transformations in a straightforward manner. The governing standards, by all means, must also be established to support a systematic development of software products.

References

[1] Pontus Johnson, Mathias Ekstedt, and Ivar Jacobson, “Where’s the Theory for Software Engineering?”, IEEE Software, September/October 2012, pp. 94-96.

[2] Frederick P. Brooks, Jr., “The Mythical Man-Month—Essay on Software Engineering”, Addison-Wesley, 1974.

[3] David Parnas, “Software engineering programs are not computer science programs”, IEEE Software, Volume 16, Issue 6, Nov/Dec 1999, pp. 19-30.

[4] Cockroaches Survive Nuclear Explosion, mythbusters database, http://dsc.discovery.com/tv-shows/mythbusters/mythbusters-database/cockroaches-survive-nuclear-explosion.htm, accessed on September 4, 2013.