Software Development’s Total Cost Evaluation by Applying Complexity Function of Different Orders

Vahid Rahmati
Ayandeh Samin Electronics, Tehran, Iran
vahidrahmati9170@gmail.com

Abstract

The software development (SD) has its own rules of designing, evolving, testing, packaging, and finally releasing which has not been studied widely with regard to its design calibrations to date. This paper introduces SD briefly to the readers by presenting main steps in SD, from need to documentation of a software, for both early and modern methods. At last, the paper suggests a method—with appropriate formulation for measuring the complexity of algorithms in a SD procedure—and solves examples based on it to show the fairness of the new idea. The method can be used generally to evaluate the total cost of a design based on known and unknown parameters that are mainly estimated based on designers’ experiences.

Keywords— Software development, cost evaluation, algorithm complexity, system design

I. INTRODUCTION

Software development (SD) is the procedure of evolving the system through consecutive stages in an arranged set of steps that led to the solution to a specific class of problems at least containing a single solvable problem. This process, despite its different types of definitions, comprises not only the definite scripting of the code procedures in a specific programming language but also the grounding of necessities and purposes of such code, the strategy of what must be implied, and approval that what is developed has met goal of the problem which is the solution [1]-[6].

Earlier to the emergence of systems growing procedures, the improvement of new smart structures or software end-user merchandises was frequently done by using skillful persons. However, in the new era the development cannot rely only on a single developer or a small group of experts trying to solve group-based problem measures in a company [2], [5].

This paper aims at introducing the traditional and state of the art methods of SD for the purpose of comparing and giving a view to the normal modern procedures that engineers need to learn and follow for their successful career.

II. EARLY METHODS

The art of early programming was in conjunction with long time gaping and head scratching making the life harder than expected. Using low level programming languages with gazing eyes to the low quality pixels of old monitors was a passive resistance to the development itself. Development of software was based on the development of software! So how a group of scientists changed this by inventing high level programming languages is not the case, but the way they introduced some important standards is. Plenty more standards like object orientation after the famous structural programming using function definitions and some other basic regular features leading to nice programming also contributed to the modern software management systems. The modern means with improving businesses by easy stages stamped out the low quality programming and system design bringing about a boom in the field. New programming languages such as JAVA, Visual Basic, C#, and ... one by one turned upside tide for long lasting economical SDs [1], [6].

The main quarrel then was over speed, readability, applicability, and generality of the software packages instead of only easy programmability of them. Many companies picked up the gauntlet but lost much of their share and vanished away on the basis of the full control of giants like Microsoft over the warming situation of smart phones market. Discipline plays an important role in development generally and in SD specifically that can be considered as a sufficient parameter in well-formed designs [1].

III. MAIN STEPS

There are indispensable steps in software developing that are studied in this part with the hope of bridging the gap...
between the traditional and the modern methods of SD [1, 6].

II.1 Needs
The main primary step is to figure out what is exactly on the brink of mismanagement which needs a broad new method to gain time and to reduce resource misuse by the SD process [1, 6].

II.2 Definitions
The most important step after understanding the system need is to define the goals of it by clear statements that describe the problem to the programmer in a coherent way. These definitions must not lack finish, meaning the every single detail must be included in the broader titles—however not stated clearly but brought concisely—in order for the designer to simply pass the tight corners of design [1, 6].

II.3 Architecture
This step shows that it’s time to make way for the consumer to choose between possible architectures based on its need and funding resources. Clearly, choosing cheaper architectures lowers the quality of consumer experience, but it avoids high bills and randomly the withdrawal from certain technologies [1, 6].

II.4 Test
No design can be exposed to the users without enough rounds of tests to buckle down possible malfunctions brining off the business for the already used road in future customer requests [1, 6].

II.5 Documentation
The designs cannot always be redone as time and resources should not be wasted, in this case, reusing finished projects is vital based on this fact that they’re well documented for future reimplementations [1, 6].

IV. Modern Methods

Last part explained the old procedures of SD in a concise way to be compatible with many references that explain such design standards and steps. To plant out feet firmly on the ground, let’s look at some modern procedures of SD which certainly will replace old standards of this field [1, 6].

IV.1 Needs
Need is also the first step in modern designing which must be passed as a remark in the consumer problem statement form usually filled before starting any project based on the definitions of the user [1, 6].

IV.2 Definitions
Definitions of the users are those gaols and requirements in the problem statement that must be met accordingly for the clear and full directive programming with almost all details [1, 6].

IV.3 Architecture
Nowadays, the architectures usually divide into web-based (online) and offline services for the software consumer making it easier for him to decide as multi platforms simplified into two categories. For example, instead of choosing among different operating systems (OSs), they choose between two options, however, the offline option makes it necessary to define which platform to be used further. In online option, the type of server side OS cannot be a game changer for the user if he is not going to directly manage it via existing panels [1, 6].

IV.4 Test
Like previous traditional method presented before, the test is needed to ensure the workability of the software in a way that covers every possible bug, and therefore, the appropriate options to fix every hidden gap by different input data and various rounds of tests in multi-platform environments [1, 6].

IV.5 Documentation
The documentation for web-based services such as cloud computing is not done in a majority of the designs as the service has been already documented by the primary company servicing the designer and the user. This is because using application program interface (API) is not always easy for the users and they may need to apply for a software designer to change it according to their requests filled in the form [1, 6].

V. Complexity of Algorithms

The cost of design must be measured via the complexity [6] of it using some parameters. For the present, this paper introduces a way to evaluate the cost of an identical simple program—with one layer routines which means the program has maximum of \( N \) single layer routines—development
based on its complexity, a parameter which gives the fairest way of expenditure calculations. Here, some parameters are introduced [1, 6, 2-7].

1. Number of main routines needed by the program $(N)$.

2. The complexity of every routine defined as $X_i$, where $1 \leq i \leq N$.

3. The time consumed to design each procedure shown as the function $T(X_i)$ measured in minutes $m$.

4. The cost per minute of programming (PMP) defined in the contract is $C$ in some currency unit.

Using these definitions, it’s easy to measure the total cost as

$$C_T \triangleq C \sum_{i=1}^{N} T(X_i) + C_o,$$

(1)

where it uses the sum to evaluate the total number of minutes consumed to produce a program with $N$ main routines each one consuming $T$ minutes based on its complexity $(X_i)$. There are some other costs $(C_o)$ such as testing costs $(C_t)$, fixing costs $(C_f)$, maintenance costs $(C_m)$, and some unknown costs $(C_u)$ added to the formula and defined by:

$$C_o \triangleq C_t + C_f + C_m + C_u.$$ 

(2)

These amounts are used widely by the programmers, however, they may have not gone through as this paper tries to go. It’s as well bound up with the way such calculations are carried out, but mainly, this will be regulated via the market itself as its competitive side forces the programmers to reduce $C_o$ amount in order to absorb more clients and have long lasting business. It’s useful to notify that the complexity is measurable using different methods explained in [5]-[7].

Another point to mention is the program’s complexity as the application of it known as a piece of work. The application is one thing, and the complexity of a software is the other, however, as custom of an engineer is to consider every corner of the project before beginning, the programmer must also consider hidden applicability of the design before start. The performability of the software for the user before releasing it as told before in test step is a must. Here an example is provided to clarify the use of equation 1 and 2 before coming to a pretty pass in SD. The fact about $C$ is that it varies based on the complexity of the program, and it can be classified into low $(C = 1)$, normal $(C = 2)$, and high $(C = 3)$ based on the unit of comparison of prices [1, 6, 2-7].

In this part, paper tries to demonstrate some examples using the equations introduced already to evaluate the total cost of different design procedures. However, the total cost can be measured by different methods like the one using neural networks in [13] or other methods as in [14] and [15] using enhanced networks. An independent method is also presented in [16].

Example 1. Calculate the total cost of a software design with 10 routines evaluated as normal complexity. The routines follow the role $X_i = i$, meaning the complexity rises linearly, and $T : i \rightarrow \sqrt{i}$. Now, Consider the $C_o = 100$ and find the $C_T$.

Clearly, we have

$$C_T = 2 \left\{ \sum_{i=1}^{10} \sqrt{i} \right\} + 100 \equiv 145,$$

as function $T$ was defined according to the unit values.

Example 2. Evaluate the programming costs of the example 1 for $C = 1...3$, $N = 1...10$, and $T : i \rightarrow \sqrt{i}, i^{1/3}, \log i$.

This can be simply illustrated to see how the number of routines and the ability of the programmer—that the complexity of program and the value of $C$ have been set according to it—can affect the total cost.

As shown in Fig. 1 the $C_T$ is calculated for low to high values of $C$ with time conversion function equal to $\sqrt{i}$.

![Figure 1: The evaluated values for $C_T$ vs. $c = 1, 2$ and $3$ with $T : i \rightarrow \sqrt{i}$.](image)

Next in Fig. 2 the $C_T$ is calculated for low to high values of $C$ with time conversion function equal to $i^{1/3}$.
Finally, in Fig. 3, the $C_T$ is calculated for low to high values of $C$ with time conversion function equal to $\log i$.

Clearly, the columns numbered one in all three tests are best fits for us, however, logarithmic design timing was the most efficient as it exhibited the lowest cost per design for every value of $1 \leq N \leq 10$.

VII. Conclusion

A method presented for estimating the total cost of SD using different orders of complexity to resemble various types of real world programming problems that engineers daily encounter. Examples also simplified the theory and its idea behind by graphs representing the rise in total cost. The main parameters involved in the SD simulations were almost 5 that can be changed according to the user’s need.

Acknowledgements

The author would like to thank the ICDST for useful comments of its editors that improved this research. At this point, it’s also necessary to remind the useful advices of Dr. Asadi of Ayandeh Samin Electronics group.

References


