***A Guide To Better Handling Of ‘The Software Problem’ In Engineered Systems***

***Prepared by: Brian McCormick***

***Date: November 19, 2014***

**1.0 Introduction**

It is common knowledge that significant problems often result in engineered systems from faulty, buggy, and difficult to maintain software. We refer to these software issues generically as ‘the software problem’. The software problem has been known to cause cost overruns, schedule delays, and product usage issues. Since systems engineers are responsible for the life cycle of an engineered system, they are primarily responsible for dealing with the software problem [4].

This paper gives guidance on better ways for systems engineers to handle the software problem. It does so by first giving insight into the extent of the problem, next it illustrates why software is so difficult to get right to give a better systems engineering perspective, then it gives specific ways in which the software problem arises, and last it discusses specific techniques to handle each of these issues.

**2.0 The Software Problem in Engineered Systems**

Software issues in the acquisition phase of the product life cycle are quite severe. The 2014 Standish Group’s Chaos report estimates more than half of projects will overrun their cost estimates by 189% [1]. It also estimates a whopping 31% of projects will be canceled before they are completed [1]. One such project cancellation is the Cover Oregon website which cost over $200 million to develop and never worked correctly for even one single customer [5]. Software issues also frequently cause costly schedule delays, often called software slip. The average project slip is approximately 222%. [1]. One such software slip is the year long delay of the U.S. military acquiring the F-35 Joint Strike Fighter aircraft [2].

Software issues in the utilization phase of the product life cycle are also very troublesome. It is estimated 61% of delivered engineered systems do not perform all of their expected software functions [1]. Also, the products are typically buggy and frequently crash, resulting in a reduction of utility of the systems and a loss of confidence in the producer [4]. Sometimes software issues even cause catastrophic losses such as when a software bug in the Patriot missile defense system caused the death of 28 soldiers and injured approximately 100 others [3]. Other effects of software issues during utilization include high maintenance costs and costly software upgrades [4].

**3.0 Why Is Software So Difficult To Get Right?**

Systems engineered really needs to understand one important thing before trying to fix the software problem: even with hard working and talented software developers, getting software to work correctly is simply hard. For example, let’s say you wanted to write a routine to find the mean (aka average) of two numbers. You might decide to write the following C++ code to accomplish the task:

double computeMean(double value1, double value2)

{

double mean = (value1 + value2)/2.0;

return mean;

}

However, you would have just introduced a bug into the engineered system. Did you spot it? The software bug is the possibility that ‘value1+value 2’ could exceed the maximum possible value (or fall below the minimum possible value) for the C++ ‘double’ type. When the above code fails, it could corrupt the system memory and cause the entire engineering system of which it is a part to fail.

So we see it is amazingly easy to get software wrong, but like any good example, this example does point us in the direction of how to get it right. Edge case testing would be a good first step to discovering problems in such code. This code example also illustrates an issue we have discovered over and over again as professional software engineers with 16+ years of experience, that is assumptions must be understood.

This is just an important in systems engineering as well [4]. In the above case, the implicit assumptions are the input numbers into the routine will be ‘nice’ numbers such as 10.2 and 14.0. When the assumption is violated and two numbers such as 225 and 226 are input into the routine then things can go wrong in a hurry.

**4.0 Understanding How Software Fails**

So what specifically can a systems engineer (SE) do to ensure software success in an engineering system? First he/she needs to understand the various ways in which software failures can occur and simply properly handle those issues. Typical software projects suffer from a myriad of problems, including but not limited to [6]:

(1) Lack of understanding of the customer’s expectations (directly under SE control)

(2) Unrealistic schedules (directly under SE control)

(3) Make sure there are no hero developers (not directly under SE control)

(4) Poor and generally unstable software requirements (directly under SE control)

(5) Lack of appropriate software development skill (not directly under SE control)

(6) Lack of testing (directly under SE control)

(7) Poor software development techniques (not directly under SE control)

(8) Software developers jumping straight into coding (directly under SE control)

(9) Using unproven and/or obsolete software technologies (directly under SE control)

Obviously this list is not complete, but it does cover many of the major software issues. The issues can be divided into 2 groups, those that are directly under SE control and those that are not. For the items that are not directly under SE control, the systems engineer still needs to understand the issues and try to persuade the software engineers to follow the guidance presented here. We also note the items in the list are not necessarily independent. In fact, they are sometimes highly correlated.

**4.1 Lack Of Understanding Of The Customer’s Expectations**

Imagine someone hires you to build a house for them. He/she hands you a blank check for building materials and labor and say they will return when you are finished. This scenario sounds ridiculous, but in the land of engineering systems it has happened more than anyone would like to admit [6].

It can be reasonably argued that the most important function of SE in the early stages of the product life cycle is to acquire an understanding of what the customer really wants, not what how he wants to accomplish the task [4]. This means that SE has a critical job in finding out what exactly the customer wants and only then giving the software engineers the requirements they need to satisfy.

We have personally been tasked to work on software for a system that had not yet been defined yet. For 3 long months. Every bit of the effort during that 3 months was wasted. The effort was of no consequence because we could only guess what was needed and we guessed wrong. How such things happen to this day is beyond our imagination.

**4.2 Unrealistic Schedules**

It is an old joke, but it still rings very true: 9 women simply cannot give birth to a baby in 1 month. The software engineering form of this joke is you cannot double the number of developers you have on a software project and finish in half the time (with a corollary joke that throwing more developers onto a late project only makes it later!).

What this software joke is saying is that software takes time to do correctly and should never be rushed (with the corollary joke is saying new developers take time to spin up and take time away from the original developers and ultimately slow everything down instead of speeding everything up, an ironic twist!).

An overly aggressive schedule usually involves introducing software hacks that can destroy a software architecture and often requires a hero developer (see section 4.3) which is also often a bad sign. Also, it often leads to insufficient testing of the software. Finally, it can also hurt moral and have the opposite effect that it tried to achieve, higher productivity, as lower moral can lead to lower productivity [7].

We know the customer will apply pressure on the SE to get things done in the earliest time frame possible. However, the SE needs to remain strong and not cave in to unrealistic schedule demands. If you try to rush the software development, you will often end up paying for it in orders of magnitude in system utilization [4].

**4.3 Make Sure There Are No Hero Developers**

While software teams have different strength developers, there should never be one developer who is considered a hero. This is the type of developer that willingly takes on an extremely difficult task that needs to be accomplished in an unreasonably short amount of time and actually successfully completes the task

While having a hero developer around might seem like a good idea, at best it is generally a sign that tasks are not being carefully planned. At worst, it is a sign that the software planning is random and chaotic. In fact, the Capability Maturity Model (CMM) development model which measures an organization’s software development process maturity has a level called CMM 1, which is the lowest level on their scale [9]. We have attended training in CMM and remember this point well: CMM 1 can best described as ‘we need a hero because our software process is so bad’. So avoid heroes at all costs.

**4.4 Poor And Generally Unstable Software Requirements**

This issue is a close cousin to issue 4.1. The SE needs to make sure all software requirements exists for a reason, that is, they ultimately help to satisfy a customer requirement in some way [4]. Also, the SE needs to make sure all of the requirements are clear and unambiguous. Last, the SE needs to make sure to never introduce changes to the software requirements without proper consideration of the entire product life cycle, especially the software engineering time it would take to handle the change.

**4.5 Lack Of Appropriate Software Development Skill**

Engineered systems are rapidly becoming more and more advanced, requiring more and more specialized software development skills to develop the software. The most obvious suggestions for this, the lofty ‘train everyone’ and ‘value education’ are nice and good for long term team improvement, but they do not address the immediate needs that a software team has. As Spolsky points out, the value of a top notch software developer is many times the value of a good software developer [10]. He further discusses the notion that the only way to create a top notch team software development team is to have only top notch software developers.

While we do not advocate taking such an extreme approach to building a software development team, we do use it as a guide. We assert that there needs to be at least one really strong, top notch developer on a software team that is developing software of any significance. This person needs to be involved in the most complex software development tasks and needs to be someone that can be approached for help with the more mundane tasks by other developers. We agree with Spolsky that the value of such an individual is many times the value of a good software developer and fills a vital need.

**4.6 Lack Of Testing**

As mentioned in section 4.2, an overly aggressive schedule can lead to a lack of testing. However, even with plenty of time for testing many software developers have a bad attitude toward testing. We have heard many comments about testing ranging from ‘the user’s will test it’ and ‘it compiled and ran so it is obviously correct’. Since there is so much negativity toward software testing, we suggest the SE demonstrate the importance of test by revealing personal experiences with testing that have had a significant positive impact on the final product. Attitude has been shown to influence motivation, so a better attitude toward testing can only help to motivate the software developers to test [11].

Testing is a large subject ranging from unit testing and software integration testing of the software modules, regression testing to make sure changes to the software did not affect any of the existing software in unexpected ways, and hardware integration testing [4]. The only additional real advise we can give is that SE needs to ensure there is a software test plan in place and the software developers adhere to it [4].

**4.7 Poor Software Development Techniques**

Any suggestions on best software development techniques almost immediately creates a ‘flame war’, so we simply suggest the SE has the software engineers study design patterns and take their own lessons learned from that. We also suggest the SE make these fairly safe suggestions that are fairly standard knowledge: stay away from global variables unless absolutely necessary, do not hard-code anything unless absolutely necessary, do not copy and paste code from one module to another unless absolutely necessary, strive to minimize the number of connections between interfaces (a concept found in systems engineering and equally applicable software engineering), and to favor composition over inheritance (perhaps the most powerful concept found in the study of design patterns).

**4.8 Software Developers Jumping Straight Into Coding**

Our experience indicates that many software developers simply love jumping straight into the code to ‘get it done’. They do this without any thought of first developing a software architect, use cases, class diagrams, sequence diagrams, state diagrams, etc. Doing these software design artifacts is viewed as something that gets in the way of the ‘real work’. While it is true that one can spend too much time on software design, it is also true that jumping straight into the code with no prior planning can be just as bad.

In our experience, it is good to make the software design process a useful tool, instead of an un-useful master that must be satisfied at all costs even when it does not benefit anything. You should make the software design process as lean as it can be without becoming useless. We suggest developing class and sequence diagrams at a minimum and other design artifacts as appropriate before jumping into coding.

**Works Cited**

http://www.projectsmart.co.uk/docs/chaos-report.pdf

http://rt.com/usa/f35-jet-software-delay-233/

https://www.ima.umn.edu/~arnold/disasters/patriot.html

Blanchard *BS,* Fabrycky WJ. 2011. Systems Engineering and Analysis, 5th ed., Prentice Hall, Upper Saddle River, NJ.

http://en.wikipedia.org/wiki/Cover\_Oregon

http://www.cic.ubn.br/~genaina/ES/ManMonth/SoftwareProjectFailures.pdf

http://www.stevemcconnell.com/ieeesoftware/bp05.htm

http://en.wikipedia.org/wiki/Waterfall\_model

http://en.wikipedia.org/wiki/Capability\_Maturity\_Model

Spolsky, J. 2004. Joel on Software. 1st ed., Epress, Springer-Verlang, New York.

http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2976624/