

Taking Software Requirements Creation from Folklore to Analysis

by Larry Bernstein
Industry Research Professor
Stevens Institute of Technology

Large software systems are too often late, costly and unreliable. Too often the requirements are not well understood or wrong. Understanding and bounding the requirements in a specification is an essential step to solving this problem. As early as 1970 Royce pointed at that unvalidated requirements leads to unmanageable projects. In particular, requirements complexity drives effort required to build software intensive systems, the time it takes to build them and their inherent reliability. Complexity management is critical and by enhancing existing simulation environments used by system engineers to formulate alternative system designs, software engineers can understand the sensitivity of requirements complexity to the likelihood of producing a workable system. Model-Driven Software Realization is a current practice for achieving this understanding. Combining functional and performance simulations with sizing and effort estimation efforts leads to a holistic understanding of feature, form, cost, schedule and trustworthiness.

Concept

The performance of new systems capabilities for government agencies and industry are often examined using simulators. The simulators provide insight into how the new capabilities will perform. The simulators shed little light on reliability, complexity and other software engineering aspects of the proposed changes. The correlation between the complexities of the proposed capabilities with that of an earlier system can be used to bound the trustworthiness, possible schedule and potential costs for implementing the new capability. I call this

the Lambda Protocol. It combines software reliability with software sizing theory. With performance, reliability, schedule and cost-estimates in hand, system and software engineers can make essential engineering tradeoffs as they set a course of action. By the way do we need a new job category, 'Software Intensive Systems Engineering' to describe that vital work that goes on at the beginning of a software subsystem before implementation begins? Some refer to this work as 'Systems Analysis;' but it has a heavy architecture connotation. The effort here is aimed at coming up with 'a' solution to the problem, later the developers will come up with 'the' solution. Many projects have had the seeds of their failure sown at this critical boundary.

When customers present ideas that need system solutions, engineers have an ethical and professional obligation to help them define and simplify their problem. You must build the best solution to the customer's problem, even if the customer does not yet understand how to ask for it.

Here is one way. The customer should be encouraged to write a short prospectus that states the purpose of the system, its value and any constraints essential to making it useful. This should not be confused with a complete set of requirements, which will emerge only through, simulation, analysis, prototyping and validation with an iterative process.

WHAT CAN GO WRONG WITH REQUIREMENTS

The requirements and design phases are important steps in a software project. If these steps are not well done, the quality of the final product will almost certainly be low. At this stage, it is good to maintain a fluid, dynamic synthesis methodology. Any production computer investment activity during this interval, including early coding, imposes a psychological

reluctance to change anything already created. Unrestrained ability to change is necessary to developing a quality design. Investing in prototyping and modeling at this stage is very helpful, but both customer and designer must remember that the artifacts produced will not necessarily find their way directly into the product.

Without an iterative plan for approaching the development of requirements, the design organization can find itself, months along on the project, **developing the wrong software functions**. For example, the designer of an ordering system for a grocery could not guess that suppliers' invoices would not be directly related to orders because suppliers grouped orders for their own delivery convenience. The customer would not mention this because "Mildred always knew how to square things" and nobody ever thought about it.

A formal process has a cutoff point. This prevents the **continuing stream of requirements changes** that can prevent coding and testing from moving along. Changes can be made in an orderly way in future releases after evaluation, but not by altering the requirements document without formal and elaborate change control procedures.

The sales team can sometimes infect both the customer and the design organization with the desire to **gold plate** the product and provide what is *desired* rather than what is *required*. The design team needs to be fully aware of this tendency born of enthusiasm and resist it, without being negative or disheartening. The attitude must be to get the core functionality right.

Finally, many design organizations do not have the necessary human factors specialists to analyze the users' tasks. Without specific attention to the people who will use the product, the organization can **develop the wrong user interface**.

When some service is especially critical or subject to hardware/network failure, the application engineer needs to build software fault tolerance into the application. These are typical issues:

- *Consistency*: In distributed environments, applications sometimes become inconsistent when code in a host is modified unilaterally. For example, the code in one server software component may be updated and this change may require sending out new versions of the client application code. In turn, all dependent procedures must be re-compiled. In situations where a single transaction runs across several servers a two-phase commit approach may be used to keep the distributed databases consistent. If the clients and servers are out of step there is a potential for a failure even though they have been designed and tested to work together. The software in the hosts need to exchanged configuration data to make sure they are in lock step before every session.
- *Robust Security*: Distributed application designers need to ensure that users cannot inadvertently or deliberately violate any security privileges.
- *Software Component Fail Over*: The use of several machines and networks in distributed applications increases the probability that one or more could be broken. The designer must provide for automatic application recovery to bypass the outage and then to restore the complex of systems to its original configuration. This approach contains the failure and minimizes the execution states of the complex of systems.

The reliability of a system varies as a function of time. The longer the software system runs the lower the reliability and the more likely a fault will be executed to become a failure. Let $R(t)$ be the conditional probability that the system has not failed in the interval $[0, t]$, given that it was operational at time $t = 0$. The most common reliability model is:

$$R(t) = e^{-\lambda t},$$

Where λ is the failure rate and the Lambda in our requirements protocol. It is reasonable to assume that the failure rate of a well maintained software subsystem is constant with time after it is operational, with no functional changes, for 18 months or more, even though faults tend to be clustered in a few software components. Of course bugs will be fixed as they are found. So by engineering the Lambda and limiting the execution time of software subsystems we can achieve steady-state reliability. Lui Sha of the University of Illinois and I define

$$\text{Lambda } (\lambda) = C/E\varepsilon.$$

This is the foundation for the Lambda Protocols and the reliability model becomes

$$R = e^{-k Ct/E\varepsilon}$$

This equation expresses reliability of a software system in a unified form as related to software engineering parameters. To understand this claim, let's examine Lambda term-by-term.

Reliability can be improved by investing in tools (ε), simplifying the design (C), or increasing the effort in development (E) to do more inspections or testing than required by software effort estimation techniques. The estimation techniques provide a lower bound on the effort and time required for a successful software development program. These techniques are based on using historical project data to calibrate a model of the form:

$$\text{Effort} = a + b (\text{NCSLOC})^\beta$$

$$\text{Schedule} = d (E)^{1/3}$$

Where a, b and d are calibration constants obtained by monitoring the previous developments using a common approach and organization. NCSLOC is the number of new or changed source lines of code needed to develop the software system. Barry Boehm's seminal books

Software Engineering Economics, Prentice Hall, 1981, ISBN 0-13-822122-7 and Software Cost Estimation with COCOMO II, Prentice Hall, 2000, ISBN 0-13-026692-2, page 403 provide the theory and procedures for using this approach.

NCSLOC is estimated by computing the unadjusted function points of a first cut architecture needed to build the prototype. With the number of unadjusted function points in hand a rough estimate of the NCSLOC for each feature package can be obtained. The prototype validates the features in the context of expected system uses. The features are grouped into logical feature packages so that their relative priority can be understood. The simplified Quality Function Deployment method (sQFD) is used to understand feature package importance and ease of implementation.

The NCSLOC can be estimated by multiplying a conversion factor by the unadjusted Function Points. Nominal conversion factors are available on the web or can be derived from previous project history.

The effectiveness factor measure the expansion of the source code into executable code. Software tools, processes and languages have their own expansion factors that can be combined to characterize the investment in the development environment. As you would expect, higher level languages have higher expansion factors. Processes like regression testing, code reviews and prototyping also increase the expansion factor. Investments in the expansion factory is like investing in design and factory tooling to increase hardware productivity,

Now, if we only knew the complexity of the feature packages the effort equation can be used. to find the staffing. In this equation, β is an exponent expressing the diseconomies of scale for software projects, the complexity of the software and factors related to the developers and their organization. The COCOMO and SLIM estimation models can be applied here but be careful to use average values for terms dealing with the software tools so that productivity gains are not counted twice. So, if we can estimate complexity then we can estimate the schedule, reliability and cost for each feature package.

Estimating Complexity

Modern society depends on large-scale software systems of astonishing complexity. Because the consequences of failure in such systems are so high, it is vital that they exhibit trustworthy behavior. Trustworthiness is already an issue in many vital systems, including those found in transportation, telecommunications, utilities, health care and financial services. Any lack of trustworthiness in such systems can adversely impact large segments of society, as shown by software-caused outages of telephone and Internet systems. It is difficult to estimate the considerable extent of losses experienced by individuals and companies that depend on these systems.

The primary component of complexity is the effort needed to verify the reliability of a software system. Typically reused software has less complexity than newly developed software because it has been tested in the crucible of live operation. But, this is just one of many aspects of software complexity. Among other aspects of software engineering complexity are elements that be mapped onto a normative scale for each feature package. The relative complexity between the feature packages can then be used for estimation. These factors are:

- a. The nature of the application characterized as

1. Real-time, where key tasks must be executed by a hard deadline or the system will become unstable or software that must be aware of the details of the hardware operation. Operating systems, communication and other drivers are typical of this software. (complexity factor = 10) Embedded software must deal with all the states of the hardware. The hardest ones for the software engineer to cope with is the 'don't care states.'
 2. On-line transactions, where multiple transactions are run concurrently interfacing to people or to other hardware. Large database systems are typical of these applications. (complexity factor = 5)
 3. Report generation and script programming. (complexity factor = 1)
- b. The nature of the computations including the precision of the calculations, Double precision is 3 times more complex than single precision.
 - c. the size of the component in NCSLOC
 - d. the extra effort needed to assure correctness of a component,
 - e. and the program flow using Cyclomatic Complexity Metric

For each feature package estimate the effort, time and reliability. Weight these factors with the performance and features in the feature package and then choose the best engineering solution that fits the customer's original goals written in the prospectus. Iterate the results with the customer to establish willingness to pay and potential simplifications before implementation begins.

This material is extracted from the author's new book that contains specific processes and case histories, including

extensive material from missile systems software. See
"Trustworthy Systems Through Quantitative Software
Engineering," Lawrence Bernstein and C.M. Yuhas, Wiley,
2005, ISBN: 0-471-69691-9

Source: Ubiquity