ISSN: 2643-8224

International Journal of Aeronautics and Aerospace Engineering

Research Article

Open Access

# When is the Design Complete?

**Neil G Siegel***

*The IBM Professor of Engineering Management, Department of Industrial and Systems Engineering, University of Southern California, USA*

## Article Info

***Corresponding author:**
**Neil G Siegel**
The IBM Professor of Engineering Management
Department of Industrial and Systems Engineering
University of Southern California
USA
E-mail: siegel.neil@gmail.com

**Received:** January 12, 2019
**Accepted:** February 1, 2019
**Published:** February 8, 2019

**Citation:** Siegel NG. When is the Design Complete? *Int J Aeronaut Aerosp Eng.* 2019; 1(1): 10-18.
doi: 10.18689/ijae-1000103



Published by Madridge Publishers

## Abstract

Almost every software-development and software-intensive system-development methodology calls for the design to reach some level of maturity before the team moves into software and system implementation. Yet data from many sources indicate that a large percentage of software and system-development programs encounter significant difficulties. Data from my own work fixing such problem programs indicates that the major recurring theme across such problem programs is that the design was inadequate for the task at hand; this, of course, should have been detected during the design phase, before the program moved on to implementation, integration, and test. In this paper, I examine indications from a number of real programs to determine why the design is so often inadequate; one of the key findings is that our standard methods, processes, indicators, and metrics for determining if the design is complete are seriously flawed. A proposal for better design-completion indicators is provided, and the implications for practice discussed.

**Keywords:** Software Development Practices; Software Development Metrics; Software Development Procedures; Software Design; Software Management; System Architecture Skeleton; System Design; System Development; Design Metrics; System-Development Metrics; Independently-Schedulable Software Entities; Software-intensive systems; Software-intensive system development.

## Background

The world of software and software-intensive systems development is full of metrics, covering many different topics: estimation of time and effort before the development process starts (a field pioneered by Boehm [1]), indicators of software quality and complexity[i], and so forth.

One would expect, therefore, for there also to be a robust set of metrics for measuring the technical progress of a design, and how one could use such metrics to determine when a design is complete.

There is, in fact, quite a lot of guidance for how to *conduct* the design phase of a development programs: dozens of checklists, suggested representations, suggested reviews, and so forth. Included in this guidance for the design phase are activities designed to assess progress. However, most of this guidance (whether from INCOSE, the IEEE, corporate processes, government standards documents, etc.) for assessing progress is based on the use of ***management*** indicators to evaluate progress through the design phase, things like "create these representations", "hold these meetings", "conduct these reviews".

But the design is not solely a management activity; at heart, design is a ***technical*** activity! One can therefore ask where are the *technical* guidelines, methods, metrics, and indicators that tell us when the design is mature enough to as to allow a development team to know

i Such as http://searchsoftwarequality.techtarget.com/guides/Quality-metrics-A-guide-to-measuring-software-quality, but many others, as well.

that they are ready to move into an implementation phase, and can reasonably have confidence that the multiple people building the individual components of the software will create entities that, when brought together and integrated, will perform with the functionality, capacity, reliability, and other characteristics that are appropriate for this software program or system?

We all are aware that software and system development is a difficult business; many, many development programs fail. For example, Glass [2] is one of many sources that report ***more than half*** of all major software development programs fail.

In this paper, I raise the hypothesis that the root cause of many of these software and software-intensive-system development failures ("failures" in the sense of having significantly late deliveries, inadequate capabilities, significant cost overruns, cancellation, rejection by the users, and often, some combination of all of these factors) are due to ***inadequate design***, especially, ***inadequate software design***. I raise the related hypothesis that one of the causes of these inadequate software designs is that we have few ways to tell if the software design is actually complete; if the design is in fact not yet complete when we enter the implementation phase, what happens is that the design is completed informally by dozens (or even hundreds) of separate programmers, working most likely without effective design coordination. This is sure to lead to design inconsistencies that show up during integration as those hard-to-find and hard-to-solve problems that drive programs into significant cost and schedule overages.

The literature has examined the question of why software development programs fail. A typical view is as follows (Figure 1):

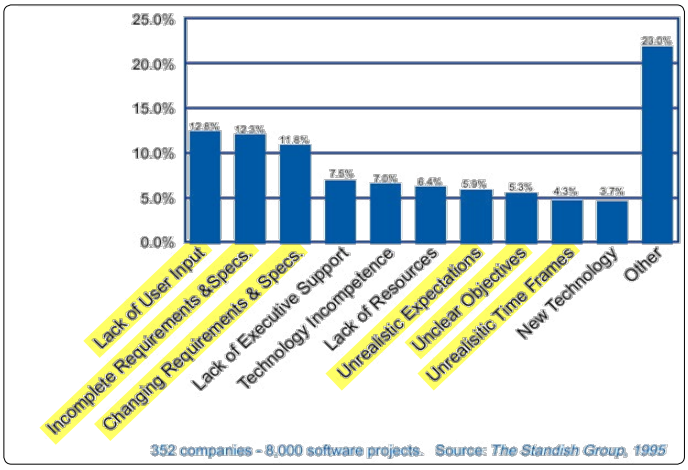


**Figure 1.** Data from the Standish Group (via Barry Boehm [1]) regarding their view of the root-cause of failures in software-development projects.

This is perhaps not as illuminating as it might seem, with the largest category by far being "other". If there is a trend or lesson in these data, it is that ***the authors believe the problem probably lies in the requirements process***; many of the items highlighted in yellow (highlighting in Boehm's original) pertain in one way or another to requirements. The literature even has a favorite phrase: "requirements creep", the notion that programs get into trouble because we let the user keep adding new requirements to the system specification, even after we have moved past the initial requirements phase. This is also a favorite conclusion of textbooks (such as Flowers [3]), university courses, corporate guidance documents (such as Northrop [4]), and Government lessons-learned reports (such as Army [5]).

The data in Figure 1 dates from 1995, but more recent data provides similar conclusions; for example, Symonds [6] provides the following list of her 15 most-common causes of system-development project failure (Figure 2).

1. Poorly defined project scope
2. Inadequate risk management
3. Failure to identify key assumptions
4. Project managers who lack experience and training
5. No use of formal methods and strategies
6. Lack of effective communication at all levels
7. Key staff leaving the project and/or company
8. Poor management of expectations
9. Ineffective leadership
10. Lack of detailed documentation
11. Failure to track requirements
12. Failure to track progress
13. Lack of detail in the project plans
14. Inaccurate time and effort estimates
15. Cultural differences in global projects

**Figure 2.** Data from Symonds: her list of the most-common causes of system-development project failure.

Items 1, 3, 10, and 11 on her list all relate to requirements. She rates items 1, 2, and 3 as the most important; note that 2 of these 3 explicitly relate to requirements: requirements are a major portion of the definition of project scope, and the place where we identify most of the key assumptions, including such items as data and components to be provided by the customer.

In summary, according to the literature, the fairly consistent "villain" of the failed-software-development (and, therefore, also of the failed software-intensive system development) story is *requirements creep*.

I find this to be a suspect conclusion, because it does not account for the observed highly non-linear outcomes. For example, a program team which had been predicting for many months that they were going to finish the project within its planned budget might suddenly make a new prediction that their system will cost five times as much money to build as was in its original estimate. They might attribute this increase in cost to "requirements creep".

Unfortunately, this is not a rare occurrence; not only do we frequently see such very large adverse changes to the predicted development cost of a system (and also, to the development schedule), we also all-too-often see such adverse changes in the predictions to key performance factors, e.g., systems that are now predicted to perform 100 times slower than promised, be 100 times less reliable than promised, and so forth[ii].

ii I don't cite development programs by name here (although I could!), in the interest of not embarrassing people. But anyone familiar with the literature will be able to provide their own examples.

I find the conclusion that the root-cause of these adverse changes is requirements creep to be suspect, because my experience is that requirements creep is fundamentally a *linear* factor: adding 10% additional requirements to a program might increase cost by somewhat more than 10%–some requirements are harder to implement than others–but because requirements are generally decomposed into a large number of "small" statements, my experience is that the effect over a large number of changes tends to be approximately linear.

The change that we find–as noted above–to the prediction for cost (and usually, schedule), however, is often highly *non-linear*: the project has in fact added 10% to the number of requirements, but the predictions about cost and schedule have increased 500%, and the predicted system performance capacity has degraded 10,000%, and so forth.

The true root cause ought to be one that inherently can explain such *non-linear* changes; requirements creep does not offer that explanation.

I spent several years of my career as a sort of a designated "fixer of problem projects" at a large aerospace company. Most of these projects displayed predictions similar to those cited above: a few percent of changed requirements, but many multiples of increase in development cost and schedule, and even more radical decreases in predicted (or measured) system performance.

These assignments allowed me to dig deep into the actual root cause that caused the adverse changes. What I saw as the consistent root problem was quite different than requirements creep; my findings were that in some fundamental sense, the design was almost always inadequate, and therefore the implementation effort was doomed. I also found a consistent root cause in the designs of these systems that could explain the non-linear behavior of the predictions.

## My Experiences and My Findings

During my career, I have found that, by count, the largest portion of a typical system requirements specification are the *functional* requirements; e.g., "use this algorithm", "in response to this action, display this information", and so forth. Since these functional requirements form the largest portion of the requirements specification (usually, well over 95% of the requirements, by count), they receive the most attention from the design and test teams, and in general, the *resulting designs are effective at implementing those requirements*. These design elements share one important characteristic: they can be designed and analyzed through essentially *static representations*, such as functional decomposition, implementation hierarchies, flow-charts, algorithm development, algorithm modeling, and so forth.

The remaining (and generally, by count, quite small) portion of the requirements specification deal with what are sometimes called the *quality characteristics* of the system: performance rates, capacities, timing, reliability, and so forth. These design elements share one important characteristic, too: they can ***only*** be designed and analyzed through representations of the ***dynamic behavior*** of the system. My conclusion from my experience fixing these problem programs was that our systems are too often burdened by an ***inadequate design for the dynamic behavior of the system***.

The design for controlling the dynamic behavior of a system needs to come in two parts: (a) the mechanisms for ***implementing*** the dynamic behavior we ***want*** (e.g., threads, inter-process communications, signals, rendezvous and synchronization, and so forth), and (b) the mechanisms for ***excluding/prohibiting*** the dynamic behavior that we ***do not want***. My findings (Siegel [7,8]) are that designers sometimes do an adequate job on the first of these items, but ***seldom do an adequate job on the second***. This lack turned out to be the root cause for almost every problem program I have been called upon to fix over the course of my career, and is therefore my candidate for the true (but often, unnoticed) root cause of many system-development failures. I have coined the term "unplanned dynamic behavior" to denote the actions that take place within a system that cause highly non-linear degradations in capacity (100x is not uncommon), reliability (1,000x is not uncommon), port-to-port timing, and so forth (Siegel) [7].

This leads to the following *typical failure scenario for an engineering project*:

- During the *requirements*, *design*, and *implementation* stages, all of the measurements indicate that the project is completely on time and on budget. All is well.
- Then, the project enters the *integration* stage.
- During the integration phase, multiple pieces are put together into larger and larger sub-assemblies, and these sub-assemblies are executed. It is at this point that major problems start to occur: things that appeared to work well in the individual parts start showing signs of very significant problems. The system crashes every ten minutes; the system processes data 100 times slower than it is intended to do so; and so forth.
- The team laboriously tracks down these issues one-by-one. Each problem takes far longer to find and fix than thought it would; as a result, our predicted project end-date starts "slipping to the right" as fast as the calendar progresses, or even faster. Each problem turns out to be a problem of *unplanned dynamic behavior*: not a problem with the individual parts, but instead with the way that the parts *interact*. Typical problems are that, at times, processing steps accidently occur out of their planned sequence; processing steps take longer than planned, and timing margins are missed; processing steps queue up unexpectedly, causing conditions akin to turbulence on communications paths, which in turn degrades performance and causes delays in timing; off-nominal data (outliers) or unexpected actions by the users cause the system to behave badly.
- The team works hard to fix each issue. But there seems to be no end to the occurrence of such problems; fixing one does not prevent new examples

of such bad behavior from occurring. No prediction about progress towards the completion of our project turns out to be justified; the predicted end-date just keeps slipping. And people get discouraged: they work very hard and fix one such problem, but 3 days later, a new and equally-difficult and equally-detrimental problem is found . . . and none of the previous corrections fix that problem; we have to start the diagnosis process entirely afresh.

- The project is soon cancelled, because the customer has lost confidence in your ability to manage and deliver the system. Or you are fired, and someone else is given a chance to finish the project. Neither of these outcomes is good![iii]

How did this happen? My conclusion is that the team *did not measure the right things about the design* (specifically, they tried to assess the progress of their design through the use of *management* metrics, rather than through the use of *technical* metrics), and therefore, did not think about the right things in the course of their design. As a result, they did not actually know if their design was going to work or not, and not surprisingly, did not create a suitable, credible, and effective design.

How do you avoid such a situation? Through (a) paying attention to the system's dynamic behavior in the design, especially the part that above I called "preventing the dynamic behavior that you do not want", (b) the creation of good technical measures for assessing the design, and most especially, for assessing the dynamic behavior of the design, (c) creating a work-plan that addresses the difficult portions of the design early in the project, rather than doing just the easy parts first, (d) creating a strong risk-management process, and (f) employing good techniques for monitoring the progress of your project (one that forces a periodic and rigorous assessment of the technical characteristics of the design). As I stated above, I have found that most engineering projects that fail do so because they have *bad designs*; the projects had bad designs in large part because they did not do these things well.

At the center of improved practice, therefore, are three techniques:

- Techniques to create a design that adequately accounts for the dynamic behavior of the final system, both in the *positive* aspect ("design to implement the dynamic behavior that your want"), and in the *negative* aspect ("design to avoid and prevent the dynamic behavior that you do ***not*** want").
- Techniques for credibly predicting the performance, capacity, and other aspects of the resulting design.
- Technical metrics that will allow for a credible assessment of the *progress* of the design.

I address each of these briefly in the sections that follow.

**Techniques to create a design that adequately accounts for the dynamic behavior of the final system**

I have previously written about my prescription for this aspect problem: the systems architecture skeleton (SAS) methodology. In the next section, I summarize that methodology, and in the section after that, I get to the key point of ***this*** paper: how one can be better at implementing the SAS approach through the use of strong metrics and indicators that help one understand ***when the software design is complete***.

The SAS methodology has deep roots; an early formulation was made by Walker Royce [9] in the mid 1980's[iv], and shortly thereafter implemented by a team he led that included Peter Blankenship, Chase Dane, and Ben Willis. It was then extended to operate over multiple heterogeneous computer processors around 1988 by David Bixler[v]. I built a complete engineering and management methodology around these techniques in the early 1990s [described by Siegel [7,10]], and subsequently applied that methodology to more than a dozen major system-development programs.

The original motivation for the SAS was the observation that many system development programs (such as decision-support systems, combat aircraft, business-automation systems, and many others) routinely experience significant cost and schedule over-runs in their software-development portion. Not only does the program incur the cost impact of these over-runs, but if – as has often been the case – the software schedule extends enough that the software becomes the pacing item on the program's critical path, very significant addition cost impact can be incurred due to the delays imposed by software onto other activities (this effect is sometimes called "the marching army").

Therefore, one of the principal goals of the entire program design and management effort is to create software estimates for both cost and schedule that are credible, and can be met. To accomplish this, I have conducted analyses aimed at understanding the root-causes of the software cost and schedule over-runs on previous development programs. Through such analyses, I have identified the following as the principal root-cause of these problems: ***The design fails properly to control the "dynamic" behavior of the system*** (e.g., control-flow, signaling, timing, capacity, race-conditions, etc.). Standard systems engineering and software-development processes tend to stress the "static" aspects of the design (e.g., algorithms, ICDs, etc.), but our case-studies (described below) show that it is the dynamic, rather than the static, aspects of the design that are generally in fact the source of these significant cost and schedule over-runs in the software portion of the program.

iii There is a bit of engineering folklore called the "90/90 rule of project management", which says that "The first 90% of the project's work accounts for the first 90% of the project's schedule. The remaining 10% of the project's work accounts for the next 90% of the project's schedule.". These paragraphs explain why this happens!

iv Described in his 1998 book, cited above as (Royce 1998); the particular implementation that this team did was called "network architecture services" (NAS), and later somewhat generalized to "universal network architecture services (UNAS).

v David coined the term hIPC (heterogeneous inter-process communications) for this implementation, and it has appeared in various places in the literature under that name.

This root-cause have been validated not only by the case-studies of previous actual programs (such as Royce [9]), but also by academic studies (such as Siegel [7]). Having achieved an understanding of the root-cause, I developed a "design-based technique" that is explicitly intended to mitigate these two root-causes. Finally, I have now had the opportunity to implement a number of real system-development programs using this design-based technique, and have ***achieved predictable cost and schedule software deliveries***. This combination of case-study results, root-cause analyses, academic studies, and program results using these particular mitigations provides confidence that the assessment of the root-causes of previous software-development program causes is correct, and that our mitigation methods have proven effective.

The following is offered as an example of the confirmatory evidence available: a study examined six actual large-scale software-intensive, real-time system development programs. The figure 3, below, depicts both a measure of quality (latent defect rate, expressed as defects above a certain severity level discovered after fielding per month per 1,000,000 SLOCs), and a measure of development-program cost performance (cost of the software at development-program completion, expressed as a percentage of the original bid cost). As can be seen, the six programs show a marked "clumping" into two distinct groups. Three programs show low latent defect rates, and the software for these programs also were all completed within 3% of their original bid cost (average: completed for 98% of their original bid cost). These are the programs that incorporated the approaches (to be described below) intended to mitigate the two root-causes listed above. On the same graph, three other programs show high latent defect rates, and it is also the case that the software for these programs all required at least 175% of their initial bid cost to complete (average: ~200% of their original bid cost); these were programs that were similar in complexity, scope, and problem domain to the three successful programs, but were executed without the mitigation measures. The result is that the mitigating measures both improved quality, and also improved the program's cost performance, that is, they reduced the variance between the original cost estimate and the cost at completion.

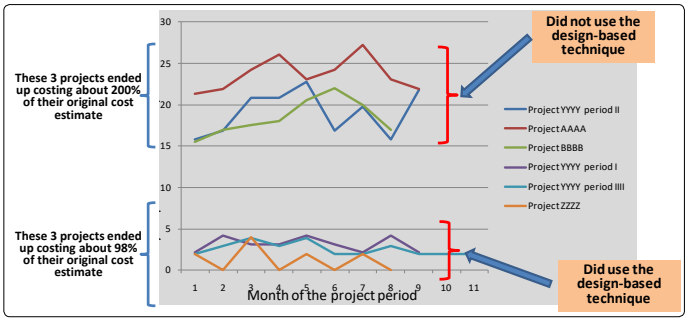


**Figure 3.** The design-based technique mitigates the risk of cost and schedule over-runs in the software development, and also results in higher-quality software.

**The business case:** Although it clearly cost money to implement the mitigating measures we have developed, it is clear from the above data (and also from the other data are available) that the cost of such implementation is more than recovered through being able to credibly complete the program's software-development effort within the original cost estimate (and, not shown in the figure, the programs were able to complete within the bounds of their original schedule estimates).

In summary, using this "design-based technique to control unplanned dynamic behavior in complex software-intensive systems both (a) improved quality (~6x improvement in latent defect rate is depicted) ***and*** (b) decreased the achieved development cost by about 2x, and allowed the programs to complete both within their original cost estimate and within their original schedule allocation.

**Root causes:** The following provides a more detailed description of the two root-causes.

Recall that above we identified that the first of the two identified root causes was problems arising from ***inadequacies of the design to control properly the "dynamic" behavior of the system*** (e.g., control-flow, signaling, timing, capacity, race-conditions, etc.). For example, the design of the software might be such that control signals and the data to which they refer can get out of synchronization. Or activities might get out of their planned sequence, violating implied requirements for processing validity. Or queuing might build up in unexpected ways, causing radical decreases in processing capacity and/or significant increases in critical port-to-port timing threads. Or off-nominal data conditions in ways not foreseen can ***cause software control to get "lost", causing crashes and resulting in low software mean-time-between failure***. All of these effects are routinely observed in actual system development programs that incorporate large amounts of software. The software business has, in fact, created an entire vocabulary of terms that describe such problems with this "unplanned dynamic behavior", terms such a "dead-lock", "race condition", and so forth; the existence of these terms is an indicator of the pervasiveness of these problems in large software-intensive systems.

Detailed investigation of thousands of actual software problem-reports on real systems (Siegel) [7] has in fact validated my hypothesis that adverse or uncontrolled dynamic behavior – I use the term "unplanned dynamic behavior" – is the actual root-cause.

Furthermore, it can be shown that the effects resulting from such unplanned dynamic behavior can be highly non-linear, that is, seemingly small errors can cause huge degradations in the performance and quality of the system; this is an important indicator that we have identified the actual root-cause of the problems.

Consider a simple example: a disk drive operates by having a metal disk coated with a storage medium spinning at a planned, constant rate. The surface of the disk is organized as concentric rings of storage medium, each called a track, which are partitioned into angular sectors of data, with control information placed in between each sector. If the design goal

is to read all of the data on a track in a single revolution of the physical disk, there is a clear timing budget for performing the necessary data transfer and processing for each sector, derived from the rate of revolution of the disk and the angular size of the sector and inter-sector control data. If the process to read and process the data from sector 1 takes even a small amount longer than this timing budget, the system will not be ready to start reading the data from sector 2 when the head is over the appropriate location, and the disk will have to be allowed to spin all the way around again before the data from sector 2 can start to be read and processed. As a result, to read the data from sector 1 and sector 2 would take much longer than budgeted, because the time for the platter to spin all the way around intervenes between the two sector reads; if there are 64 sectors per track, it will take 64 revolutions to read the track, rather than the design goal of reading the track in a single revolution, and hence, it will take 64 times longer than planned to read a track of data. The result is a highly non-linear degradation of performance from the expected level, due to what could be a relatively minor overage – literally, a few microseconds – in a timing budget.

This potential for extreme non-linearity in degradation of performance due to an unexpected dynamic in system behavior, even a minor such variation, is what makes the management of system dynamic behavior (and the avoidance of unplanned dynamic behavior) such a fruitful one for system development. The undesirable behavior could manifest itself as timing/performance/capacity degradation, as in our simple example, or as reliability/mean-time-between-failure degradation, or in some other fashion. Often, of course, the causal relationships are far more complex and subtle (and hence, harder to find) than in our simple disk-drive example.

Problems such as these tend to show up only late in a conventional integration cycle (since they involve the interaction among system components, the behavior does not appear when the components are being tested alone; it only appears as the integration stage nears completion), and also can be very difficult to find and correct, and worse, attempts at correction (since they involve changes to those software elements that are controlling the complex dynamic behavior of the system) can often result in introducing other errors, e.g., "one step forward, two steps backwards", and therefore, the cost and schedule to correct such problems can be difficult to predict.

There are two additional ways in which this effect can become non-linear, that is, incidents that appear relatively minor can result in significant increases to program cost and schedule:

- "Positive feedback loops" of increasing problems: e.g., problems found (including the problems in controlling system dynamic behavior) causes patches and modifications to be required to one portion of the code ... which "ripple" into other portions of the code ... which cause more software development to be undertaken than planned, and more modification to be undertaken than planned. Each additional unplanned modification introduces a new possibility for somehow "breaking" yet another portion of the design, and the cycle repeats.
- Software development gets onto the program critical path, and therefore causes adverse cost implications across the entire program, not just in the software-development effort. This is so common that a joke – the "90/90 rule of software development", cited above – has been created to memorialize it! This is a prominent effect: because problems such as those that we are considering arise in the integration phase, it is often the case that schedule delays that arise only this late in the program cannot be made up, and late delivery of the software impacts other, often even more expensive, elements of the program that cannot be completed and tested without the software (e.g., flight testing).

## Description of the Method

In order properly to control the dynamic behavior of the system, I have employed a set of design and process techniques that provide guidance for correct implementation of the SAS; these comprise what I term a "technology of integration", a set of technical approaches that focus on exactly this aspect of software-intensive-system risk, with the goal of mitigating exactly these risks; specific techniques include what we call our "software backplane" and our "systems architecture skeleton". This approach was invented for a U.S. government program (the Cheyenne Mountain missile warning system), and has a long list of other program successes (with many different customers, military and civilian) on large, complex, real-time systems.

The key characteristics of this methodology to implement the SAS are as follows:

- Connection-oriented definition of work-flow through the system, e.g., defines the dynamic behavior that we want in the system.
- Use of a "white-list" methodology for control of system behavior: only those actions, sequences, and players that are registered in advance of an actual instance of execution are allowed; all other attempts or requests for execution are denied. The method does allow for the potential of dynamic updating of these registered and allowed events.
- Separation of the work-flow management mechanism from the functional software, so that it can be implemented and managed by experts in that field
- Use of the data signals as the actual control signals, thereby eliminating the potential for data and control signals to become out of synchronicity with each other
- Use of a "white-list" methodology that excludes and prevents other sorts of unplanned dynamic behavior

Having provided evidence that unplanned dynamic behavior is the root cause of many of the significant software development

problems that are seen in so many programs, my proposal therefore is to establish a methodology that aims to ensure that the ***design adequately controls such unplanned dynamic behavior before we enter the implementation phase***. The following are some of key steps in this methodology:

- Separate the implementation of the control structure from the implementation of the system's functionality. My personal practice often is to do this through the use of middleware that is responsible for implementing the system's control structure (usually through dispatch and control of mission threads), and for ***preventing*** any other combination of stimuli and processing from taking place; there are of course other means to the same result. In the middleware-based approach, one can enforce compliance by implementing this middleware in software modules separate from those that implement the system's functionality, and through the use of code auditors that ensure that there are no coding constructs that attempt to implement control constructs (e.g., rendezvous and synchronization, dispatch, etc.) in the software modules outside of the middleware.
- Ensure that the control structure is inherently robust. One technique that we often use, for example, is to avoid the use of separate signals or channels for data and control; having those separate allows the possibility that they will get out of synchronization. Instead, we tend to prefer designs where the arrival of a data packet ***is*** the control signal.
- Of critical importance is incorporating features into the control structure beyond just those that are intended to implement the dynamic behavior you want; you must also include design features to ***prevent the dynamic behavior you do not want***! We often accomplish this latter point through a white-listing methodology, where behavior (threads, connections, service calls, etc.) is allowed only if it is on a pre-defined list.
- Identify every potential stimulus that will commence processing of some sort within your system; identify every ***independently-schedulable software entity*** in the system; use those independently-schedulable entities to define explicitly the mission threads you wish to have within the system, each mapped to their stimuli (time triggered, data triggered, user-action triggered, etc.). Define how these interactions are mediated and enforced: e.g., the control structure used to start and control processing within the system. Define the timing budgets for each mission thread (and for each step along each mission thread), and how those are monitored and enforced. Then program the middleware to control the implementation of exactly those threads and relationships, disallowing any attempt to execute other threads, combinations, or sequences. This is in effect a "white-listing" methodology. I find it revealing that no software or systems engineering text that I have consulted ever talked about the concept of the independently-scheduleable entities within a system; but those are in effect the "moving parts" of your system – how could one sensibly proceed without such an explicit definition?
- Since the design and implementation of the control structure is completely separate from the functional and algorithmic aspects of the system, it can be implemented even before those other elements are available, using a stimulator to create the external stimuli and message load to the system in a realistic fashion; one can in essence run the system's dynamic skeleton at full load long before the mission applications are available. This creates schedule time – a project manager's most precious resource – to analyze the dynamic behavior of the system (using technical metrics such as port-to-port timing, processor loading, and so forth), allowing one to isolate anomalies, and to correct them. Since this can be done without the large volume of mission applications software being present (at the beginning of the process, those can be represented by stubs), it is easier to "see" the dynamic behavior, since it is not masked by the existence of all of those mission applications. This is the "skeleton" portion of the phrase "systems architecture skeleton". Integration can proceed then by removing selected stubs and replacing them with real modules as they become available. One can also see why this approach reduces the risk of re-using prior software; one can control the introduction of such re-used modules, doing them one-by-one, and seeing places where these re-used modules disrupt, violate, or tend to "fight" the desired control structure, making it easier to adapt those re-used modules to the new system.
- Doing the above also creates a very significant management opportunity: to assign all of the control structure design and implementation for a system to a small yet skilled team that is implementing the middleware. Correct design and implementation of complex software controls is a rare skill, and spreading such responsibility across a large portion of a team has proven to be a poor management practice. I have written about this in Siegel [7].

For completeness, the following assumptions and limitations of the system architecture skeleton (SAS) methodology are provided:

- The SAS methodology, of course, assumes that it is in fact possible to separate the implementation of the control structure of a system from the implementation of the algorithms and other functionality of a system. This has proven to be the case for a variety of types of systems: military command-and-control, industrial process control, decision-support, and others. But we do not exclude the possibility that there are systems for which this assumption is not valid, or not practical.

- The "white-list" approach that is inherent in the SAS does not allow for general use of software objects by any and all requestors. Such an "open" execution model is fashionable in some industries and application domains; the SAS – which limits or prevents such general use –is not likely to be a satisfactory design solution for designers for whom such general use of software objects is deemed necessary or desirable.

Interestingly, we have found that this approach addresses both the problem of unplanned dynamic behavior, but also addresses the problem of not achieving planned software re-use rates, as our analysis has indicated that misunderstanding the dynamic interactions among those re-used and new software elements is the often the root cause of failing to achieve the planned level of software re-use.

**Techniques for credibly predicting the performance, capacity, and other aspects of the resulting design**

When we can, we ought to measure actual performance of our system; this process is often called benchmarking. But of course, we need to make predictions about achieved system performance before we have completed the system. For this purpose, we use models and prototypes.

Models and prototypes are often *nested* or *chained*. An example of such nesting might be:

- A physics model of radio-frequency propagation feeds . . .
- A model of an antenna, which feeds . . .
- A model of the antenna-mast height, which feeds . . .
- A model of the received signal quality, which feeds …
- A model of successful packet completion rate, which feeds . . .
- A model of message completion delay (average and variance), which feeds . . .
- A model of end-to-end completion time and accuracy for a specific capability, which feeds . . .
- A measure of some system operational performance measure

There are, however, many different ways to interconnect these models. Sometimes, they are all put together into a model-of-models, with fully automated interactions between each model. Other times, they are run separately, but the outputs from one are automatically fed into the next model in the chain (these are usually called *federated models*). Other times, the models are completely disjoint, and the outputs from one are manually transferred into the next model in the chain.

It is my experience that it is important that each model be maintained and operated by its actual creator; that creator is the expert who knows the limits of credibility for their own model better than anyone else, and having them maintain and operate that model, in turn, contributes to achieving better and more credible predictions. This motivates me usually to prefer the use of separate models with manual transfer of data! That sounds old-fashioned, but better accuracy and credibility in my view is more valuable than automated interconnection.

We use the models to analyze our system and its candidate designs; that is, how well does each of our candidate designs perform. Since we are concerned with whether it meets the needs of the *users*, the model must finally reach the level of being able to make predictions about the *operational performance measures*, not just about the technical performance measures. This is a common failing of system models; many are designed only to make *technical* predictions.

Of course, the system architecture skeleton described in the previous section provides a new and important way to predict system performance, too: we implement the actual system architecture skeleton early in the project, we populate it with models or prototypes of each independently-schedulable entity, and this creates predictions based on realistic system dynamic behavior, since the system architecture skeleton is the mechanism for implementing and controlling the system's dynamic behavior.

**Technical metrics that will allow for a credible assessment of the *progress* of the design**

"Measuring progress on the design" is, in my view, of significantly more importance than books, standards, and training manuals generally recognize. Few texts, for example, talk at all about "measuring design progress" (Siegel) [11][vi].

To the extent that these source documents advocates the use of technical measures, they only advocate those that measure the design as a "black box"–its visible performance and capacity, for example. Seldom do they advocate technical measures of the "goodness" of structure and form; there are seldom "white box" measurements of the design.

I certainly advocate the use of management measures about design progress, and the use of black-box measures of system performance. But I have found that these are not sufficient! I therefore also advocate the use of direct technical measures about the internal suitability of the design. I advocate this because this is what I have consistently seen as the real failure mechanism in important engineering projects.

As discussed above, the need to prevent in advertent adverse, unplanned dynamic behavior is a key success factor in design, and also is an indication of why *simpler is usually better* in design. Everyone advocates the KISS[1] principle, but what is it that you actually measure in order to achieve a simple design? The sources seldom tell you what to measure, in order to see if your design is simple or not. Here is my favorite example of a tangible design parameter that you can aspire to keep simple: *the number of independently-schedulable software entities within the mission software* for a large, complex system. I *always* measure this parameter when I design or evaluate a system. One of my best systems (still in use 30+ years later – and that is an eternity in the software

vi Although my forthcoming text "Engineering Project Management", Wiley, 2019 Siegel [11], will do so!

business!) had only *seven* independently-schedulable software entities in the mission software.

At the same time that I was building this system with seven independently-schedulable software entities in the mission software, the same customer had another company building a system for a slightly different mission, but one that shared many of the same operational conditions and constraints. That contractor was having problems, and the customer asked me to take a look at their work. It turned out they had no count or list of the independently-schedulable software entities within their system! How could they expect to control adverse dynamic behavior? At my suggestion, they made such a list; it turned out that they had more than 700 independently-schedulable software entities within their system's mission software. What human being could understand the potential interactions and implications of so many independently-schedulable parts in a complex system? Their system was *never* fielded; it was about 100x less reliable than needed (and more than 1,000x less reliable than my similar system).

In the end, the next system that I built for this same customer (which had 9 independently-schedulable software entities within the system's mission software; I was unhappy that we went from 7 to 9!) was eventually adopted to take over the mission intended for the other company's project. The other company spent nearly $1,000,000,000.00 and yet produced nothing useful. They failed to implement the KISS principle, and they failed to assess the progress of their design using any sort of white-box technical metric.

The above, although simplified from the versions presented in the references (especially Siegel [7] and Royce [9]), demonstrates that a set of steps can be provided, and a set of technical items that can be instrumented, observed, and measured, resulting in an ability to declare that in fact the design is complete in some meaningful, tangible, and objective fashion.

## Implications and Conclusions

I spent many years of my career as the designated "fixer of system-development programs that were in trouble". My insight from that experience was that lots of systems have ***bad designs***. Why might that be?

- Designs often take place in an abbreviated period of competitive proposal selection. Once you have won, it is psychologically hard to recognize that your design might not be ideal. After all, it won!
- Fitting a design to the mixture of social and technical constraints is hard, and we tend not to iterate enough times, or to select the design based only on technical criterion ("effective"), rather than both technical and operational criteria ("effective and suitable").
- We trim the design trade-space far too quickly. People hate ambiguity, and are therefore quick to down-select to a design that seems feasible, in order to stop the pain of carrying a lot of true alternatives.

Whether these are the correct causes of poor designs or not, in this paper I presented evidence from actual system development programs that what I have termed herein ***unplanned dynamic behavior*** is a key root cause of many system development problems; in essence, the design is not adequate for the task at hand.

This insight allows one to create a methodology, sequence of steps, and ***technical*** metrics that can in fact provide far higher assurance that one asserts that the design is done, you are truly in a state where the implementation can proceed without excessive risk.

In my view, the design is ***not done*** until you have accomplished these items:

- Explicit definition of all of the independently-schedulable entities within the system (and likely, only a relatively small number of these), and a clear plan for how they will interact to form threads.
- Design mechanisms to control unplanned dynamic behavior (e.g., white-listing allowed behaviors, etc.)

These (and the technical metrics derived from them, such as port-to-port timing variance, etc.) form the ***key indicators of a mature, complete design***.

Evidence was presented that using this software design approach, one can consistently complete complex software and software-intensive system development efforts, meeting cost, schedule, and performance parameters.

## References


1. Boehm BW. Software Engineering Economics. Prentice Hall; 1981.
2. Glass RL. ComputingFailure.com. Prentice Hall; 2001
3. Flowers S. Software Failure: Management Failure. 1996.
4. Northrop G. Northrop Grumman Systems Engineering Handbook. CTM-101, Tenets of program success. 2010.
5. Army. Landwarnet. A report by the U.S. Army Science Board. 2007.
6. Symonds M. 15 Causes of Project Failure. 2011.
7. Siegel NG. Organizing Complex Projects Around Critical Skills, and the Mitigation of Risks Arising from System Dynamic Behavior. USC, 2011.
8. Siegel NG. Organizing Projects around the Mitigation of Risks Arising from System Dynamic Behavior. *International Journal of Software Informatics*. 2011; 5(3).
9. Royce W. Software Project Management: A unified framework. Addison Wesley. 1998.
10. Siegel N. A Manager's Perspective on the Benefits of Ada. TRW Data Technologies Division Technical Notes Series. 1994.
11. Siegel NG. Engineering Project Management. Wiley Publishers. 2019.