Software engineering processes for Class D missions

Ronnie Killough (ronnie.killough@swri.org), Debi Rose (debi.rose@swri.org)
Southwest Research Institute, 6220 Culebra Rd., San Antonio, TX, USA, 78238

ABSTRACT

Software engineering processes are often seen as anathemas; thoughts of CMMI key process areas and NPR 7150.2A compliance matrices can motivate a software developer to consider other career fields. However, with adequate definition, common-sense application, and an appropriate level of built-in flexibility, software engineering processes provide a critical framework in which to conduct a successful software development project. One problem is that current models seem to be built around an underlying assumption of "bigness," and assume that all elements of the process are applicable to all software projects regardless of size and tolerance for risk. This is best illustrated in NASA's NPR 7150.2A in which, aside from some special provisions for manned missions, the software processes are to be applied based solely on the criticality of the software to the mission, completely agnostic of the mission class itself. That is, the processes applicable to a Class A mission (high priority, very low risk tolerance, very high national significance) are precisely the same as those applicable to a Class D mission (low priority, high risk tolerance, low national significance). This paper will propose changes to NPR 7150.2A, taking mission class into consideration, and discuss how some of these changes are being piloted for a current Class D mission—the Cyclone Global Navigation Satellite System (CYGNSS).

Keywords: software engineering processes, NPR 7150.2A, Class D mission, CMMI

1. INTRODUCTION

Software engineering processes are often seen as anathemas; thoughts of Capability Maturity Model Integration®[1] (CMMI) key process areas and NASA Procedural Requirement (NPR) 7150.2A[2] compliance matrices can motivate a software developer to consider other career fields. One of the reasons for the common disdain is that current models seem to be built around an underlying assumption of "bigness," and assume that all elements of the process are applicable to all software projects regardless of size and regardless of tolerance for risk. This problem is illustrated rather clearly in NPR 7150.2A, *Software Engineering Requirements*, in which, aside from some special provisions for manned missions and other safety-critical software, the defined software processes are required to be applied based solely on the criticality of the software to the mission, completely agnostic of the mission class itself. That is, the processes applicable to a Class A mission (high priority, very low risk tolerance, very high national significance) are essentially the same as those applicable to a Class D mission (low priority, high risk tolerance, low national significance).

However, with adequate definition, common-sense application, and an appropriate (but often missing) level of built-in flexibility, software engineering processes can provide a critical framework in which to conduct a successful software development project. Having proposed a rather boring title in the abstract submission, we now propose perhaps a more apropos title (an obvious play off the name of a popular Clint Eastwood movie) that will also serve to structure the remainder of this paper: *The Ugly, The Bad, and The Good.* In these sections, we will first outline some of the less "attractive" elements of two software engineering process frameworks (NPR 7150.2A and CMMI), discuss the challenges that many face when trying to scale software processes down, and then highlight some of the positive aspects of these software engineering process requirements and models. We will then conclude with a section entitled *Digging Up Gold,* in which we will propose changes to NPR 7150.2A, taking mission class into consideration, and discuss how some of these alternative practices are being piloted for a current Class D mission—the Cyclone Global Navigation Satellite System (CYGNSS).

2. THE UGLY: BASIC FACTS AND THE INTIMIDATION FACTOR

I know of very few projects, software or otherwise, that cannot be organized and conducted according to the following basic steps:

1. Determine what needs to be done
2. Figure out how to do it
3. Do it
4. Make sure you did it

Sure, there are some additional activities that usually have to be included, such as planning and managing the effort, delivering and supporting the product, and perhaps creating user documentation and conducting training. But somehow when it comes to software engineering processes, the simplicity of these four basic steps tends to get lost among the myriad details that can cause even the most ardent software process practitioner's eyes to glaze over. To someone that is new to software engineering processes, or to a project manager that is not a software developer himself, these processes can be downright intimidating.

2.1 The CMMI

The "ugly" of software engineering processes is readily illustrated in some facts and figures. We'll begin with the CMMI:

- The first widely-used version, Version 1.1, of the Capability Maturity Model (CMM, which was the precursor to the CMMI), was at least seven years in development, outlined 18 key process areas (KPAs) and 150 "top-level" activities![3]
- One of the more widely used books documenting the CMM and its application consumed 344 pages, excluding appendices.[4]
- The most recent version of the CMMI, Version 1.3, is documented in a book with 552 pages, and the CMMI has grown from the original 18 to now 22 process areas.
- The latest book requires 161 pages just to explain what the model is, to define terms, and to describe how to use the book and the model, before the elements of the model itself can be described.[5]

Quoting what is probably the most ironic statement in the book: "A Capability Maturity Model (CMM), including the CMMI, is a simplified representation of the world."[6]

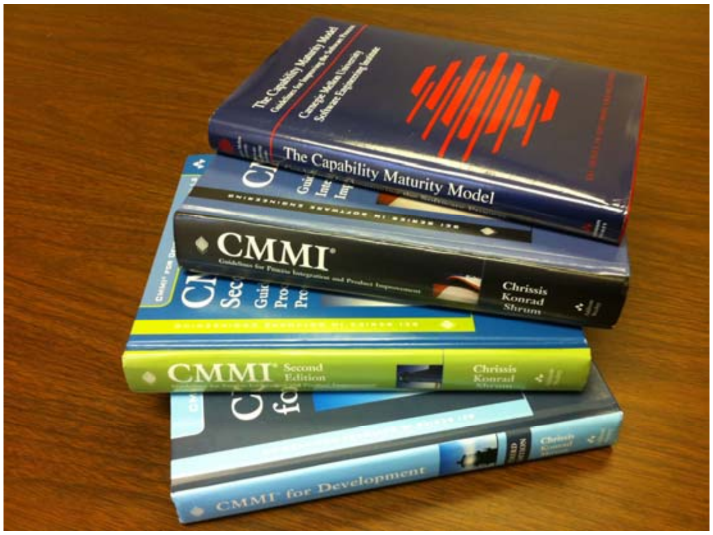


Figure 1. This stack of CMM and CMMI books illustrates the "bigness" often associated with software engineering processes.

2.2 NASA's NPR 7150.2A

Turning our attention to NASA's NPR 7150.2A, we see some reduction in sheer quantity of pages (a paltry 48 pages vs. hundreds in the CMMI), but we encounter a new set of statistics that can be equally intimidating. A few examples include:

- NPR 7150.2A requires as many as nine *types* of plans related to the software development effort. If one considers that a mission may have eight to perhaps fifteen or more organizations involved in software development, the sheer number of software plans can quickly multiply, not to mention the numerous programmatic and system engineering plans that other applicable NASA procedures require, that contain overlapping/duplicated content to that required in the software plans such as configuration management and risk management.[7]
- The Software Development and Management Plan alone is required to address 24 individual categories of planning items.
- The procedures describing the Software Requirements Specification (SRS) outline 18 different *categories* of requirements that must be specified in the SRS.

The complexities of software engineering requirements and frameworks can have a rollout effect in the amount of process oversight that is applied to software development projects. In the case of one mission, it reached an extreme level, which at the time created extraordinary frustration, but with the benefit of the passage of time can now be viewed with hilarity. Years ago one of the authors of this paper was leading the development of software for an instrument on a NASA mission. We were conducting the project under the aegis of our internal CMM Level 3 software processes, which had been mapped to NPR 7150.2. As part of our internal processes, we were subject to process oversight by our internal software quality assurance (SQA) department. NASA also applied Independent Verification and Validation (IV&V) oversight to the project, and there were regular teleconferences associated with the IV&V group. The overall mission software systems manager was also providing some SQA-type oversight for the project and conducted regular teleconferences with our team. Finally, there was additional supervision provided by NASA mission and quality assurance. At some point I realized that the instrument flight software development effort was being overseen by more quality assurance staff than I had developers on the project—literally, there were a total of seven staff involved in quality and process oversight (meaning we were on regular teleconferences and/or getting email inquiries from seven different people) for this one development effort, versus a software development team consisting of myself and three developers! Ultimately, we presented these facts to NASA and were able to eventually realize some relief, but this example serves to illustrate what can happen when software process requirements (including process oversight) are implemented "blindly" without considering cost/benefit, mission significance, technical risk, team experience, and other factors.

2.3 Process stack-up

The story presented above is an example of what I refer to as "process stack-up"—that is, when multiple organizations are involved, each of those organizations may have their own software process requirements. Given the ability to apply professional judgment, the experienced software engineer could plot a path through the collective set of software engineering process requirements and create a plan that satisfies the *intent* of the processes established by each of those organizations. However, this can take significant effort and so it is not uncommon to see a project manager just resign himself to complying with all of them to avoid the pain of weighing through reams of process documents and writing waivers. This can be made more difficult because each of those organizations may be required to be in "compliance" with their own internal processes, so that, for example, they can "take credit" for full process compliance on that project in their next independent CMMI assessment. As such, there is little motivation for any given organization to back off on some of the technicalities associated with their own processes for the good of the project. The hapless software project manager can easily find himself in the throes of "process stack-up"—having to satisfy process requirements from multiple organizations that have equivalent objectives but differing procedural steps, documentation requirements, or reporting requirements.

3. THE BAD: PITFALLS OF PROCESS SCALING, TAILORING, AND WAIVING

One additional “ugly” fact of NPR 7150.2A that was not presented in the prior section, is that NPR 7150.2A defines eight “classes” of *software*, ranging from Class A (Human Rated Software Systems) to Class H (General Purpose Desktop Software).[8] These eight software classes are intended to provide a framework by which the process requirements defined in NPR 7150.2A can be tailored and adapted based on the criticality of the software to the mission. Just to keep things confusing, NASA also uses a lettered nomenclature to denote different classes of *missions* – ranging from Class A missions (high priority, very low risk tolerance, very high national significance) to Class D missions (low priority, high risk tolerance, low national significance).[9] In the last section of this paper, we will discuss ideas for scaling NPR 7150.2A based on mission class. However, for now, we will focus our attention on the problems associated with scaling and tailoring software processes in general, using the eight classes of software defined in NPR 7150.2A as one specific example of how process tailoring itself can sort of get out of hand.

3.1 The problem with the color grey

Developing good software engineering processes and organizational procedures is not an easy task. Defining processes and procedures that are general enough to be widely applicable to an organization’s various types and sizes of projects while being specific enough to be of practical utility is hard enough. But these are just the beginning of the difficulties—building in the types of flexibilities that the “real world” requires while maintaining compliance with the underlying model (e.g., CMMI) multiplies the challenge. As a result, most software engineering procedures give a head-nod to tailoring and adaptability but often fail to deliver any real flexibility, and the path leading to a project-appropriate set of processes can be long and winding.

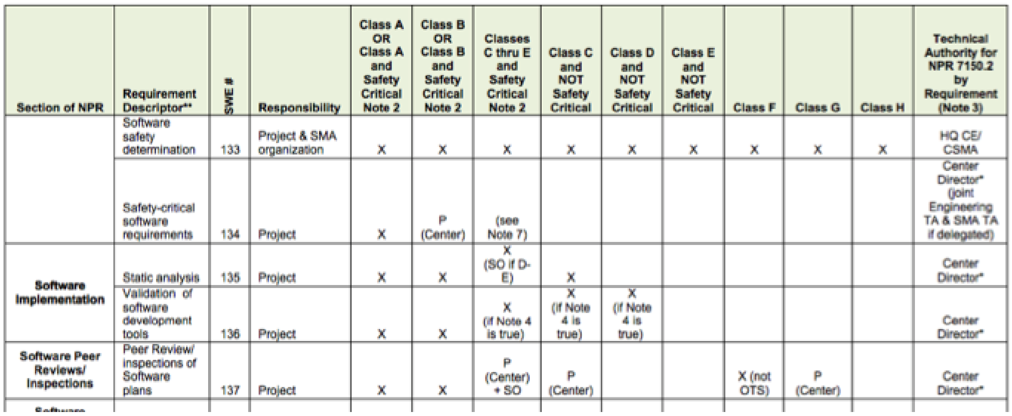
The real world is often grey, including the world of software development and management. No processes are perfect; and the size, criticality, and cost of individual projects can vary greatly. As such, software processes need to allow for flexibility and adaptability; that is, they need some “grey-ness” built into them. As most practitioners know, and as we discussed in the prior section, software engineering process documents have a well-deserved reputation of being rather wordy (although “comprehensive” might be more polite) and at least a perception of rigidity. And, where flexibility does exist, the tailoring instructions may be as complex and confusing as the processes themselves. I have wondered why computer scientists and software engineers seem to have such difficulty defining software engineering processes, and particularly those elements that would provide needed flexibility and adaptability, in any concise format. Perhaps part of the problem lies in the fact that we deal in ones and zeros. I often lament that many of today’s computer science graduates are of what I refer to as the “point and click” generation. The software development tools of today enable programmers to create complex, multi-threaded software applications without requiring the developer to have an understanding of how that application gets translated into ones and zeros, and what happens at execution time, “under the hood,” so-to-speak. In spite of that concern, it remains true that in the world of computers and software, everything is ultimately a one or a zero—there is not much room for anything grey in that world. And so, creating the color grey in the land of computers requires LOTS of ones and zeros to be placed in close proximity to one another. That does not really result in the color grey, but it appears to.

Figure 2. The color palette above appears to show shades of grey, but it is an illusion—the only way to create shades of grey in the world of computers is to put a lot of alternating ones and zeros (or, black and white pixels) next to each other. Similarly, if flexibilities are provided for in software process requirements, it often only results in an appearance of flexibility—in reality, the models typically require all or most of the boxes to be checked at least in some fashion.

This is too often true in the world of software processes. If any flexibility is built into a set of software process requirements, it can be difficult to parse through the quantity and complexity of the tailoring guidance provided – all those pages of procedures and compliance and tailoring matrices translate to lots of ones and zeros, too. And just as in the simulated color grey, if one does take the time to wade through it all, it is not uncommon to ultimately discover that the anticipated level of flexibility and adaptability is not really there—it just appears to be—it is all an illusion.

3.2 The eight classes of software in NPR 7150.2A

Let's refer back to the eight classes of software defined in Appendix E of NPR 7150.2A. Appendix D of that document is comprised of a matrix that uses those eight classes of software to provide tailoring guidance for how to apply the software engineering requirements defined in the body of the document to these various software classes. However, the tailoring matrix is 132 rows long and consumes eight pages. The matrix is further supplemented by 18 footnotes providing additional interpretation and clarification guidance. Even the column headings are not entirely straightforward—instead of the columns being just "Class A," "Class B," "Class C," etc., as one might expect – tailoring guidance is even embedded in some of the column headings, such as in Column 6: *Class B OR Class B and Safety Critical Note 2*, and in Column 9: *Class D and NOT Safety Critical*. Do you need help in determining the applicability of one of these 132 process items to your project? No problem—the matrix includes a final column that defines no less than five different technical authorities having jurisdiction over the various portions of the process requirements (but of course one must also reference the footnote labeled "Note 3" for additional clarification in determining final technical authority).



Section of NPR	Requirement Descriptor**	SWE #	Responsibility	Class A OR Class A and Safety Critical Note 2	Class B OR Class B and Safety Critical Note 2	Classes C thru E and Safety Critical Note 2	Class C and NOT Safety Critical	Class D and NOT Safety Critical	Class E and NOT Safety Critical	Class F	Class G	Class H	Technical Authority for NPR 7150.2 by Requirement (Note 3)
	Software safety determination	133	Project & SMA organization	X	X	X	X	X	X	X	X	X	HQ CE/ CSMA
	Safety-critical software requirements	134	Project	X	P (Center)	(see Note 7)							Center Director* (joint Engineering TA & SMA TA if delegated)
Software Implementation	Static analysis	135	Project	X	X	X (SO if D-E)	X						Center Director*
	Validation of software development tools	136	Project	X	X	X (if Note 4 is true)	X (if Note 4 is true)	X (if Note 4 is true)					Center Director*
Software Peer Reviews/ Inspections	Peer Review/ inspections of Software plans	137	Project	X	X	P (Center) + SO	P (Center)			X (not OTS)	P (Center)		Center Director*
Software													

Figure 3. A small excerpt from the NPR 7150.2A Appendix D Requirements Mapping Matrix illustrating the complexities often associated with tailoring software process requirements.

So, to determine how to tailor the processes for your particular software development effort, you need to pour through 132 rows of information and associated footnotes. And, I have failed to mention Appendix A, where one can find definitions for what NPR 7150.2A means when it uses various terms, including such terms as "Software" and "Uncertainty." Assuming you spend the time to parse the matrix, what you will find is that, ignoring safety-critical software, the difference in process requirements among at least the first three classes of software is rather trivial—you will not find many blank spaces in the first few columns of the matrix.

3.3 The four classes of missions in NPR 8705.4

As was asserted in the introduction to this paper, the processes defined in NPR 7150.2A are required to be applied based *solely* on the criticality of the software to the mission, completely agnostic of the mission class itself. That does not seem reasonable—most would probably agree that the size, cost, and criticality of the mission itself should play a significant role in the level of software processes that should be applied. One might argue that the mission class, as defined in NPR 8705.4[9], is even more important in dictating required software processes than the class of the software. However, if mission class were added to the tailoring provided for in Appendix D of NPR 7150.2A, the result would be a three dimensional matrix with rows representing process requirements, columns representing software classes, and the third dimension representing mission classes. Be careful what you ask for.

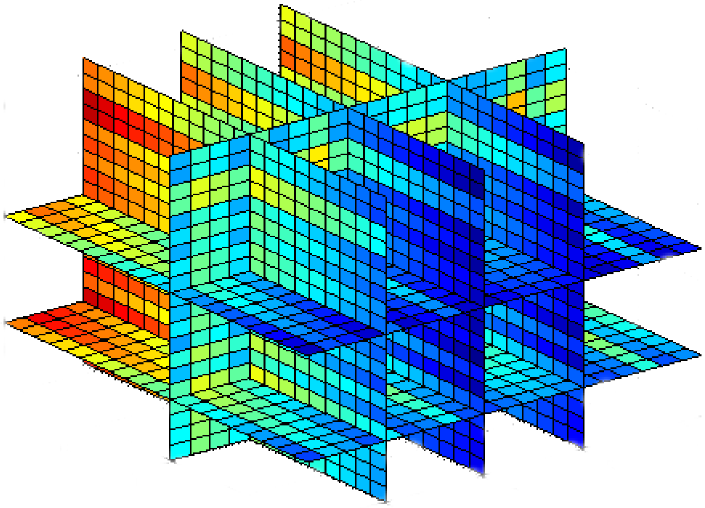


Figure 4. Be careful what you ask for: adding mission class to the tailoring matrix in Appendix D of NPR 7150.2A might result in a 3D matrix that would be impossible to navigate.

3.4 Issues of formality and professional judgment

One final challenge associated with scaling software processes to a given application is the difficulty in identifying any one process area or activity that, taken alone, seems unreasonable or irrelevant. Should a software project ignore configuration management? What about risk management? Peer reviews? Requirements? As such, a common approach to scaling software processes is not to eliminate activities, but rather to make adjustments to the level of *formality* in their implementation. But therein lies the difficulty—defining levels of formality in a concise and unambiguous way can be elusive, because formality is most readily expressed and understood in subjective terms; a process may be "formal" or "not formal," or perhaps "less formal" or "more formal." Then, there is the problem of documentation. For a software quality assurance auditor or CMMI appraiser to believe that you are, in fact, performing a given software process activity, you have to produce some evidence. Evidence means documentation; and, to many, documentation equates to formality.

Take peer reviews as an example in the context of a low-cost or low-risk software effort. Perhaps a project is, in fact, conducting peer reviews. But maybe the results of those peer reviews were captured in redlines to some printed code, and those printouts were discarded once the redlines were incorporated. Or in today's world, maybe the developer had his laptop with him in the peer review and made the simple corrections to the code during the peer review and added some temporary comments into the code that he referenced to implement the remainder of the changes back in the lab, after the peer review concluded. Come audit time, where is the peer review evidence? Even if some of the evidence is kept by the project, does it comply with the process requirements? What level of evidence will be considered acceptable? Someone's word? Or perhaps a project management note that a peer review of module X was conducted on date Y, even though no defect logs were kept?

A book describing one of the early versions of the CMM included a chapter entitled "Applying Professional Judgment."[10] That is what is needed—the ability to apply professional judgment. But professional judgment does not get you very far in an effort to achieve or sustain a particular CMMI level, or to pass a process compliance audit. It is not that a qualified and experienced software project manager needs significant help in making those judgment calls; but rather, it is difficult to ascertain which types of judgment calls can be made that will still, for example, allow your organization to pass a CMMI Level 3 appraisal, or keep your project in compliance with NPR 7150.2A. It is perhaps telling, in this regard, that the more recent revisions to the CMM book (those describing the CMMI[1]) exclude this chapter on professional judgment. To be fair, I am certain that the intent of the CMMI is to describe a framework that will accommodate the use of professional judgment in the definition and application of organizational-specific processes and procedures. However, the level of effort required to define processes in a way that enables the professional to comfortably apply this judgment while remaining compliant is at issue.

4. THE GOOD: IT ISN'T ALL BAD

In truth, not much needs to be said here. Most agree that in spite of the shortcomings, there is good to be found in software engineering processes, including CMMI and NPR 7150.2A. Outside the context of adequate planning, programmatic structure, and discipline in software engineering activities, the success of a given software project is impossible to predict. Historically speaking, programming and software engineering were synonymous. The software development process began by popping open an editor and banging out code on a keyboard. Requirements and design existed only in the mind of the programmer, debugging and verification were the same thing, and somehow the software was always 90% complete.[11] To be sure, some still live in that world, and many others probably wish they did. Part of the problem was sourced in our universities—software course assignments consisted mostly of small short-term projects, and the resulting program was expected to be turned in the next class period, or perhaps a week later for larger assignments. This resulted in several generations of software developers that knew programming languages, data structures, and even topics like compilers and processor architectures that, as I stated earlier, are no longer required in many modern programs. However, these developers lacked any real sense of sound engineering practices. Progress has been made—some schools now incorporate semester-long software engineering courses where students get an introduction to software engineering concepts and at least one semester to practice what they learn in the context of a larger assignment. However, a degree plan only has so many hours available to allocate among all the various necessary courses. Properly and practically defined and implemented, software engineering models and processes provide the framework to train staff in the "engineering" and "programmatic" aspects of software development that, in spite of the progress, are still not always taught well in our universities.

In preparation for this paper I sat down and read NPR 7150.2A cover-to-cover in one setting. I either need to be committed to a mental institution, or perhaps I deserve some sort of award for bravery. However, setting aside some of "The Ugly" facts discussed earlier, it really is not that unreasonable a document. Could it be better organized? Yes. Could it be laid out in a more compact format? Yes. But it does provide some specific process requirements without succumbing to the temptation to dictate excessive implementation specifics (such as document outlines) like some of the older military software standards did.[12] I must admit it is a little harder to say the same about CMMI—it is easy to read 50 or 100 pages of text without coming away with much of anything in matters of practicality. Of course, the CMMI is a software process model outlining process *areas* and goals, rather than a specific set of processes and procedures. I have also long taken issue with the CMMI's rather draconian definition of "organizationalism"—it seems to conflict with the principle that software processes are really tools in a toolbox. Further, I have seen it result in processes that tend toward excessive "sameness" that create barriers to process acceptance by the skeptics, which works against the very organizational goals the CMMI wants us to achieve. However, the CMMI does provide a framework to guide the development of processes and procedures and is arguably the most influential international standard for software engineering processes. It has now been around for over two decades and, according to the CMMI Institute, the CMMI serves as the underlying model for the software and system engineering procedures for more than 5,000 organizations in over 70 countries.[13]

5. DIGGING UP GOLD: SOME IDEAS FOR CHANGE

It is an easy observation to make that, in spite of so many efforts (only a few of which have been referenced herein) by so many people spanning over three decades to reach nirvana in the definition of software engineering processes, much work still remains. Of course, we cannot hope to address this larger issue in this small paper, even if we were to boldly (and foolishly) claim to have all the answers. While some of the propositions put forward here could be applied to other efforts and in other contexts, our recommendations in this concluding section will focus narrowly on NPR 7150.2A and its applicability to Class D NASA missions. We will also not tackle the larger issue of how the body of NPR 7150.2A could be better written or organized—rather, we will posit our suggestions under the assumption that NPR 7150.2A will remain essentially as it exists today. Finally, since we cannot solve the obvious double-overloading of terms that exists in NASA's use of lettered classes to define both software types and mission criticalities, the reader will have to pay very close attention when software and mission classes are both used in the same sentence or paragraph. To aid with this, in this section the word "class" will be capitalized when referring to mission classes (e.g., "Class D mission"), but will be lowercase when referring to software classes (e.g., "software class B").

5.1 Some strategies for scaling software engineering process requirements

Scaling down software engineering processes that were created with an assumption of bigness can be challenging. Before diving into some specific recommendations for Class D missions, some general strategies for scaling processes down are offered below, along with some examples. Most of the recommendations in the next section are derived from one or more of these strategies.

1. *Eliminate selected process requirements*. Completely eliminating a given process activity can leave behind some lingering heartburn as it can be difficult to determine what activities are of no importance at all even on smaller missions. So, when selecting a process requirement to do away with, two guiding principles may help. Guiding principle #1 is to select an activity in which, due to the small size of the project, the value of the activity can likely be realized without any formal documentation or reporting. In this way, the Software Project Manager (SPM) can conduct the activity "informally," with "informal" defined as "no required documentation." Guiding principle #2 is to select an activity that effectively represents a secondary check on another same or similar activity that is already being done by someone else.

   Example: An SPM can easily ascertain the status of a small development effort without formal trending of metrics; as such, the process requirement to analyze and trend metrics could be eliminated, or perhaps the requirement to collect and report formal metrics at all might be eliminated. A second example is Independent Verification and Validation (IV&V)—by its very name, this is an activity that serves as a secondary check and could be selected for elimination (NPR 7150.2A already makes some provision for selected application of IV&V).

2. *Reduce the scope of process requirements*. If a given process requirement is deemed important enough to be included even for smaller or less risky efforts, the scope of the requirement could be reduced in some fashion.

   Example: Continuing the metrics example above, rather than requiring five types of metrics to be collected, perhaps only two or three may be required.

3. *Merge some software process activities with system or mission-level ones*. As previously noted, most software process requirements are written with an assumption of "bigness"; but they are also written with the assumption of "alone-ness." That is, a given process activity may be necessary and important, but that activity may not need to be implemented as an activity distinct from the equivalent activity at the subsystem, system, or mission level.

   Example: Risk management is an important process activity. However, requiring separate software-level risk management plans, processes, and reporting chains is not always important, particularly on smaller efforts. A second example may be requirements verification. Clearly verification is a critical activity on any mission; however, for some missions, software verification could be merged with subsystem verification testing with little to no impact to mission risk, particularly if thorough software unit testing is enforced.

4. *Reduce the level of detail in some software engineering activities*. When considering the task of designing hardware, it can be difficult to identify a step to eliminate—a certain minimum amount of detail is required in order to realize the implementation at all. Software is not that way—in fact there remains substantial dispute among the software engineering community as to what constitutes an appropriate level of detail in pre-coding design activities. As such, adjusting the level of detail required in the software design is a good candidate for de-scoping software engineering process activities.

   Example: While some amount of formal software design work is important, the level of detail can be easily and reasonably adjusted for various efforts. NPR 7150.2A already allows for reducing the level of detail in the software design for certain classes of software. A common approach is to require an architectural design for the software but to eliminate the requirement for detailed, module-level designs.

Additional similar scaling strategies exist just in the context of required plans, specifications, and other documents:

- The required content for some plans and specifications could be reduced.
- A number of otherwise distinct software plans could be combined into a single plan and/or merged with upper-level plans.
- Selected documents could even be eliminated altogether for some missions.

With regard to the reduction of required content, it has been observed that NPR 7150.2A requires 24 distinct items to be addressed in just one of the nine software plans and 18 types of requirements to be captured in the Software Requirements Specification. For smaller, lower-risk missions, reducing the number of types of software requirements to just four or five seems achievable.

Previously, it was noted that NPR 7150.2A requires as many as nine types of software plans. For smaller efforts, most of these plans could be combined into a single Software Development Plan. NPR 7150.2A already allows for software plans to be combined and for software-level plans to be combined with system-level plans. However, this is not made obvious; and no guidance is provided for when plans should and should not be combined.

This last point brings up an important observation as we lead into the concluding recommendations for a Class D mission appendix to NPR 7150.2A. Even if a given set of software engineering requirements does *allow* for some project-specific adaptations, very often there is not sufficient time to research those options, develop an approach that is both appropriate for the project and that will find acceptance among whatever process authorities exist, and still meet initial project deadlines. This is a key lesson learned on the Class D CYGNSS mission, and one that extends beyond software. There existed only approximately six months between contract award and System Requirements Review (SRR), and most of the various software and system engineering planning documents were due at SRR. The CYGNSS team was quite aware of the need to “think outside the box” with regard to process efficiency in the context of a mission of this type; and, in fact, we were encouraged to do so by NASA. However, with such a short time between contract award and SRR, and without the benefit of any existing guidance, there was little time to evaluate options when it came to eliminating, merging, and/or reducing the scope of required planning and mission assurance documents. This observation drives the need for some “pre-tailoring” of system and software engineering processes in order that the implementing organization can successfully and maximally implement scaled down, efficient processes for Class D missions. Perhaps a more effective approach to dealing with Class D mission process requirements would be to produce an entire self-contained document outlining requirements for system (including software) engineering and mission assurance activities for Class D missions that supersedes the other NASA Procedural Requirement documents; but this paper will not attempt to address that broader issue except to occasionally make recommendations when software and system engineering activities and/or documents should be merged.

5.2 A proposal for a Class D mission appendix to NPR 7150.2A

In order to avoid the nightmarish (and, of course, unrealistic) idea of the three-dimensional tailoring matrix discussed in section 3.3 of this paper, we propose the creation of a “Class D Mission” appendix to NPR 7150.2A. This appendix would specifically outline requirements, options, and guidance for software process requirements for Class D missions. In the interest of simplicity and clarity, the appendix should address only one or two classes of software—specifically software class B and perhaps class C—since it is not realistic that a Class D mission would ever incorporate class A (man-rated) software. For software at and below software class D, a small set of *recommended* practices could be included to guide those efforts. Process requirements associated with safety-critical software can also likely be ruled out for a Class D mission; if a Class D mission were to include safety-critical software, the existing NASA requirements would apply.

One of the reasons that the software class tailoring matrix contained in Appendix D of NPR 7150.2A is so lengthy is that it begins by listing all of the document sections as rows in the table and then uses the columns (together with some footnotes) to indicate applicability. For the proposed Class D appendix to be concisely useful, it is proposed that the Class D mission appendix instead list the major software lifecycle activities and then specifically list what is required together with recommended practices. Activities and artifacts that are NOT required should not be listed, except when needed for clarity. Finally, the requirements and recommendations should be described in such as way as to avoid, or at least reduce the incidence of, the subjectivity inherent in statements like “do it, but be less formal about it.”

As the recommendations are outlined, reference will be made to the CYGNSS mission (a NASA Class D mission that is currently in development), along with some narrative describing if or how some of these recommendations are being implemented on that mission. The recommendations provided in the following sections are not intended to be excerpted and dropped in "as-is" into a Class D appendix to NPR 7150.2A; rather, they are put forth as an initial starting point for discussions, hopefully leading to the creation of such an appendix.

One final over-arching item that needs attention in the context of software process requirements for Class D missions is the applicability of the CMMI. Currently, NPR 7150.2A requires a non-expired CMMI for Development (CMMI-DEV) Level 2 rating for all class B software. The difficulty with this requirement is that most universities and small businesses do not have any CMMI rating or the infrastructure to implement the processes or the resources to obtain one. If there is any mission class in which universities or small businesses may be involved in the development of class B software, it would be a Class D mission. Perhaps the existence of a CMMI rating could serve as a replacement for any other required processes for Class D missions, but the requirement to hold a CMMI rating for a Class D mission does not seem realistic.

5.3 Software planning

It is important for any mission and any development team to have an understanding of what is to be developed, how it is to be developed, and the resources required. As such, the activity of software project planning (and the document containing the results of that planning) is of vital importance, even on smaller missions. On larger missions, in which many distinct organizations may be involved in software development, various mission-level software plans are needed to guide the development, integration, and lifecycle support for all of the software that will ultimately become part of the spacecraft, ground operations, and data processing systems. Additionally, each organization responsible for software development must also have one or more software plans that contain additional planning detail specific to the software being developed and to the organization responsible for developing it.

Because they typically have much smaller budgets than higher-classed missions, Class D missions are often necessarily comprised of teams that span a limited number of organizations. As such, the need for multiple mission-level software plans, combined with multiple software plans at each organization, is diminished. For Class D missions, it is recommended that *mission-level* software plans be integrated with mission-level system engineering plans, including areas such as configuration management, risk management, review processes, integration and test plans, and operations, maintenance, and retirement plans.

Each organization responsible for one or more computer software configuration items (CSCIs) should produce a single Software Development Plan that includes plans for the following:

- A list and brief description of the CSCIs and accompanying documentation that will be developed by the organization.
- An implementation plan, including the method for deriving and capturing requirements, the design approach, the development resources, and the procedures for handling reused software.
- Review and quality plans, including plans for conducting internal peer reviews, formal technical reviews, and the role of software quality assurance on the project.
- A software test plan, including planned levels of software testing (e.g., unit, integration, verification), planned test configurations and needed resources, and how and when those testing activities merge with subsystem testing. Clearly establishing test levels and responsibilities can be particularly important on Class D missions if some of the recommendations herein are implemented (see section 5.5).
- A software configuration management plan, including tools and procedures for source code control, software and document versioning convention, procedures for baselining software and documents and production of version description documents, and tracking of problem reports and change requests during development and after delivery.
- Software security plan, including how the software and development environment will be protected from accidental or malicious corruption during development and whether any specific software security requirements have been imposed.

Note that programmatic items often found in an SDP (such as risk management, work breakdown structures, staffing plans, schedules, and milestones) are not listed. It is assumed that, on a Class D mission, these items will have been addressed in planning documents at the system/subsystem level containing the software. If this is not true for a given software effort, then these items should also be included in the SDP.

On CYGNSS, a total of five organizations are developing or contributing to the development of flight or ground software. In the case of CYGNSS, the mission system engineering documents were not combined with mission software plans, as recommended here. Because CYGNSS is somewhat of a “pathfinder” for implementing processes appropriate for a Class D mission, a single mission-level Software Management and Development Plan (SMDP) was created to provide some guidance with regard to how to interpret contractual mission assurance requirements and to determine how (and how much) to apply NPR 7150.2A. The CYGNSS SMDP also documents mission-level software plans and procedures and provides some recommended best practices for lower classes of software that may be developed on CYGNSS but that lie outside any contractual software process requirements. Organizational-level SDPs have also been developed for class B and some class C development efforts covering the types of planning items contained in the list above.

5.4 Software design and implementation

The astute reader will notice the title of this section and note there has not yet been any discussion of requirements—that we have skipped straight to design and implementation. As will soon be asserted, requirements are important regardless of the size of the effort. However, recommendations regarding requirements appear in the next section in order to combine them with the topics of verification and test.

A matter of long and continuing debate in the software development community is whether documenting the design of software outside the software itself is important and, if it is, how the design should be documented and what level of detail is appropriate. Many believe that, because software is created using a precise language, the software itself *is* the design. Of course, various so-called “methodologies” for describing the design of software have been created over the years, most notably structured design[14] and object-oriented design.[15] I have always maintained that the specific design notation is not really the most important thing. Rather, identifying the most significant elements of the design and then describing or illustrating those elements in the most effective manner for the element being described is much more useful than remaining “true” to a single software design methodology.

Attempts have been made to compare the discipline of software to other paradigms, such as building construction or electrical engineering. While it is an imperfect analogy, and there is value in comparisons with other paradigms, when it comes to software design, I tend to like the comparison to architectural building design. In any non-trivial software development effort, it is important, before any software coding begins, to first create an overall architecture for the software. This is one of the weaknesses of some instantiations of agile methods—without some idea of the scope of the overall effort, how that scope will be partitioned among a set of defined components, and what the end product will look like, the resulting application can (and most likely will) result in a piece of software that resembles the famous Winchester House.[16]

For Class D missions, the following guidelines are proposed as required content in the software design document:

- An overall block diagram of the software showing key software components and the flow of control and data among those components and the external systems the software interfaces with.
- A list and narrative description of each software module/computer software component (CSC), such as the operating system or executive, device drivers, application-layer interface components, command and telemetry handling modules, and science data processing algorithms.
- A comprehensive memory map for the computer system on which the software will execute.
- A description of the execution model, identification of interrupt service routines, a list of tasks and priorities, and software timing diagrams for nominal and selected off-nominal operational scenarios.
- A list of significant data structures and identification of any global data.

With regard to software coding, there is relatively little that needs to be said. Whether it important to require coding conventions for Class D missions is debatable; it is certainly good practice. What is most important when it comes to code development is the use of documented configuration management procedures and a source code control tool. The proposed required content for the SDP in the previous section includes a plan which addresses source code control. Additionally, implementation of peer review-level source code inspections is strongly recommended.

5.5 Software requirements, test, and verification

In section 5.3, we asserted the importance of planning even for small efforts and outlined in just one page how that planning could be scaled for smaller efforts by combining planning documents and by reducing required content in software plans. It is equally important to understand, document, track, and verify software requirements even on small efforts.

It has been noted that NPR 7150.2A lists 18 different types of software requirements. While most every book on software engineering has its own favorite list of categories of requirements—some short, some long—I have found that the following four tend to spur the software engineer toward considering the most critical elements of the software to be developed:

- Functional requirements – These are the basic requirements and answer the question: "What must the software do?"
- Performance requirements – These requirements answer questions such as "How fast must the software perform its functions?" and "How many resources is it allowed to consume?"
- Interface requirements – These requirements deal with both internal and external interfaces and specify the details of those interfaces or reference Interface Control Documents (ICDs).
- Fault handling requirements – Many consider these functional requirements, and technically that is true. However, without calling them out separately, they are often forgotten or not given adequate attention. These requirements answer questions such as "What types of errors and faults may be anticipated?" and "How is the software required to respond to these faults, and how must the software report them?"

Additionally, a context diagram should be created for each CSCI to be produced that identifies the external interfaces.

For Class D missions, a decision must be made as to whether to derive a distinct set of CSCI-level software requirements for formal verification, or to just capture these as part of the system or subsystem in which the software will reside. It is recommended that, unless there are reasons to do otherwise, the CSCI-level requirements be captured within the containing system or subsystem. From those system or subsystem-level requirements, a more detailed set of requirements for each module, or CSC, can be derived and documented "less formally," such as in a spreadsheet, or on the back of a napkin for that matter. The important thing is when a developer takes up a given software component to begin design and implementation, he first document a detailed set of "design requirements" that can be quickly reviewed with the software lead to ensure the module will provide the needed subset of the overall software functionality, it do so within the required allocation of time, it correctly interface with the other modules, and it report and handle errors in a manner consistent with the rest of the software.

The decision to combine CSCI software requirements with subsystem requirements impacts software test and verification planning. Assuming the overall CSCI requirements are integrated with the containing subsystem requirements, subsystem verification and software verification become one activity. This decision must be coordinated with the cognizant subsystem integration and test (I&T) lead to ensure that incorrect assumptions are not made with regard to the maturity of the software that will be delivered to I&T and that adequate test time is included in the I&T plans and schedules.

Note that if software and subsystem verification are merged, this will place additional importance on the task of software unit testing. The CSC-level requirements that were defined must be verified during unit-level testing, together with the normal unit testing activities of code coverage and boundary case testing. While merging software and subsystem verification can be an effective approach for Class D missions, it can backfire if the software is inadequately unit tested and hastily integrated, as it can result in significant delays in the I&T schedule. Clearly documenting testing levels and lines of responsibility in software and system plans is critical.

On CYGNSS, some software requirements and verification activities are being combined with the equivalent system-level activities, such as the mission operations software. For the spacecraft flight software, the decision was made not to combine software and system-level requirements for the spacecraft, but to instead have a distinct set of software requirements that will be verified separately. This decision was driven at least in part by the fact that CYGNSS is comprised of eight micro-satellites (microsats), and adjusting the spacecraft I&T schedule to accommodate additional flight software testing for all eight microsats would not be cost effective or time efficient. Further, any changes that are found to be needed in the flight software during I&T will need to be tested and verified in an "offline" environment so as not to consume key I&T resources and to minimize disruption of the overall I&T of the eight microsats. For other Class D missions with only one spacecraft, the additional and separate set of flight software requirements and accompanying verification may not be necessary.

6. CONCLUSION

Software engineering processes are difficult to define with clarity and conciseness, and building in real-world flexibilities makes such a task orders of magnitude more difficult. Striking appropriate balances between disciplined implementation and adaptability, thoroughness and conciseness, are significant challenges. Further, writing software processes is boring to most, and internal groups responsible for defining them are often loathe to spend any more time than necessary in their creation. As a result, most defined software engineering processes are either overly prescriptive with little in the way of obvious or pre-defined flexibility, or are excessively abstract in order to make them broadly applicable.

This paper has attempted to take a critical but somewhat light-hearted look at these issues in the context of two software engineering process standards, the CMMI and NASA's NPR 7150.2A. The authors are currently involved in the development of software for a NASA-funded mission called CYGNSS and are actively working to plot a reasonable path through the maze of software engineering process requirements on this Class D science mission. Some initial observations and recommendations for others to consider have been offered, concluding with a recommendation for the creation of a Class D mission appendix to NASA's standard for software engineering processes—NPR 7150.2A.

REFERENCES

[1] Chrisses, M. B., Konrad, M., and Shrum, S., [CMMI for Development, Guidelines for Process Integration and Product Improvement, Version 1.3], Third Edition, Addison-Wesley, Boston (2011).

[2] NASA Software Engineering Requirements," NPR 7150.2A, National Aeronautics and Space Administration, Washington, D.C. (2009).

[3] Paulk, M., Curtis, B., Chrisses, M. B., and Weber, C., [The Capability Maturity Model^{SM} for Software, Version 1.1], Software Engineering Institute Technical Report CMU/SEI-93-TR-024 (1993).

[4] Paulk, M., Weber, C., Curtis, B., and Chrisses, M. B., [The Capability Maturity Model, Guidelines for Improving the Software Process], Addison-Wesley, Boston (1994).

[5] Chrisses, M. B., Konrad, M., and Shrum, S., [CMMI for Development, Guidelines for Process Integration and Product Improvement, Version 1.3], Third Edition, Addison-Wesley, Boston, Pages 1-161 (2011).

[6] Chrisses, M. B., Konrad, M., and Shrum, S., [CMMI for Development, Guidelines for Process Integration and Product Improvement, Version 1.3], Third Edition, Addison-Wesley, Boston, Page 9 (2011).

[7] "NASA Space Flight Program and Project Management Requirements," NPR 7120.5E, National Aeronautics and Space Administration, Washington, D.C., Appendix G (2012).

[8] "NASA Software Engineering Requirements," NPR 7150.2A, National Aeronautics and Space Administration, Washington, D.C., Appendix E (2009).

[9] "NASA Risk Classification for NASA Payloads," NPR 8705.4, National Aeronautics and Space Administration, Washington, D.C., Appendix B (2004).

[10] Paulk, M., Curtis, B., Chrisses, M. B., and Weber, C., [The Capability Maturity Model^{SM} for Software, Version 1.1], Software Engineering Institute Technical Report CMU/SEI-93-TR-024, Pages 77-79 (1993).

[11] Paulk, M., Curtis, B., Chrisses, M. B., and Weber, C., [The Capability Maturity Model^{SM} for Software, Version 1.1], Software Engineering Institute Technical Report CMU/SEI-93-TR-024, Page 22 (1993).

[12] "Software Development and Documentation," MIL-STD-498, United States Department of Defense, Washington, D.C. (1994).

[13] "Who Uses CMMI?", <http://cmmiinstitute.com/results/who-uses-cmmi/> (2013).

[14] Yourdan, E., and Constantine, L., [Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design], Prentice-Hall (1979).

[15] Jacobson, I., Booch, G., and Rumbaugh, J., [The Unified Software Development Process], Addison-Wesley (1999).

[16] "The Winchester Mystery House," >http://www.winchestermysteryhouse.com> (2013).