Should the DoD mandate a standard

Software Development Process? Part 2.

Specific Problems With The DOD-STD-2167 Process. One cause of the software crisis is that the classic software development process has not always been successful on large projects. The proven process, methods, and life-cycle have rarely worked as promised and have often inhibited innovation. The following specific problems have been raised with regard to the DOD-STD-2167 mandated process:

▪ **Life-cycle Constraint.** Many new developmental life-cycle models have been introduced in the last few years, and others will be created as software engineering evolves. Several differ fundamentally from the classic waterfall life-cycle and cannot be mapped into it. Notable examples of methods having life-cycles prohibited by 2167 include certain rapid prototyping, AI, and recursive development methods such as Object-Oriented Development. As Mogilensky noted "The classical 'waterfall' life-cycle is to modern software management as global common data structures is to Ada software design." Thus, requiring conformity to a single standard life-cycle model is an improper restriction. The counter argument that all life-cycle models are minor variations of the waterfall life-cycle and thus permitted within 2167 is simply not true.

Other counter arguments are true, but do not really address this issue because of the 2167 and MIL-STD-1521 requirements that specific products be developed and reviewed during specific life-cycle phases. While more life-cycle models are consistent with 2167 than with MIL-STD-1679, many others are still prohibited.

▪ **Functional Decomposition Constraint.** The functional emphasis of the Software Requirements Specification, the functional aspect of certain design entities of the static software hierarchy of 2167, and the way this hierarchy is tied to the software development process tend to force the developer to use a functional, hierarchical-decomposition software development method.

Thus, the static software hierarchy is tied too closely with the software development process, prohibiting one from first developing the proper Ada structure and only then decomposing it into a static hierarchy for purposes of Software Configuration Management. The static software hierarchy of 2167 does not map well into the network structure of well-designed Ada software, and impacts the order and scope of integration and testing. This, however, should be method, language, and software architecture dependent. This problem is another example of improper "how to" constraints.

▪ **Top-Down Constraint.** Paragraph 4.8 of 2167 states "The contractor shall use a top-down approach to design, code, integrate, and test all CSCIs unless specific alternate methodologies have been proposed . . . and received contracting agency approval." This choice of top-down as the single default development approach implies it is the preferred approach for all software development. Yet the appropriateness of top-down, bottom-up, outside-in, inside-out, or holistic approaches is language, application, methods, and life-cycle dependent. Extensive reuse often implies a bottom-up approach to design and test. The compilation order restrictions of Ada encourage a bottom-up approach to Computer Software Component (CSC) testing. The development of critical software implies bottom-up design and testing, and the development of test suites requires at least a partial bottom-up approach.

The counter argument that top-down is currently preferred is irrelevant. A consensus rarely produces state-of-the-art approaches and any default is subject to obsolescence. The argument that all design methods are top-down is incorrect. Many other approaches exist, and it can be argued that every major project should use a combination of methods. Arguments that most bottom-up examples are subject to debate miss the point that other methods exist and the contractor should be free to use the best methods.

▪ **Program Design Language Constraint.** Paragraph 5.2.1.4 of 2167 makes the use of a PDL the default method for the top-level design of each CSCI and paragraph 5.3.1.5 mandates use of a PDL in development of a detailed design. But the use of a PDL as a top-level design description method is probably inappropriate. Graphics are clearly superior in understandability for presenting top-level architectural designs in terms of software units and their relationships.

The use of a PDL as a detailed design description method is also becoming inappropriate. Due to the high modularity and low complexity of well-designed Ada units, the lack of distinction between Ada PDL and Ada code, and the design aspects of the Ada specification, the nature and purpose of PDL is changing. PDL is not needed to document the logic of the body of many units because of

by Donald G. Firesmith

their trivial size and complexity.

The PDL requirement and default probably resulted from findings that the use of PDLs increased productivity and reduced life-cycle costs compared to the use of flow-charts. They certainly are useful for specifying the internal logic of programs written in low-level languages or resulting from software development methods that produce relatively large and complex unit bodies. PDL, especially when viewed as a program documentation language, may also prove useful in the automatic generation of Software Detailed Design Documents. However, it is inappropriate to imply that the use of PDLs, or any single detailed design methods or tool, is to be preferred under all circumstances. This inhibits contractor innovation.

One frequent argument in government circles is that defaults do not necessarily prejudice the contracting agency. If the customer is truly interested in graphic methods, then the PDL default should not adversely affect this. However, many developers are convinced this is a real problem. Another counter argument is that specifying the use of a PDL does not necessarily specify how it is to be used. Although this is true, it would nevertheless be hard to argue that a graphic method is a PDL.

▪ **Informal Test Constraints.** 2167 currently contains many requirements and defaults regarding the performance and documentation of informal testing. These improper constraints cover such areas as the documentation of unit-level test requirements, responsibilities, schedules, test cases, procedures, and results in the Software Development Files, the default for individual testing and configuration management, the implication that units are integrated individually into CSCs, and the documentation of considerable information concerning contractor-internal CSC integration and testing in the Software Test Plan. Many developers view these requirements as not cost-effective in many cases and as unneeded, unwanted micro-management.

▪ **Review Process Constraints.** The size and complexity of today's systems overwhelms the formal review process of 2167 and 1521. It is impossible to properly review manually-produced gothic-novel-sized specifications. There is often insufficient time for a proper in-depth analysis and correction of errors. Reviews concentrate on superficial formatting problems while important technical issues are buried. By allowing the contractor to limit the scope of any single review, a better analysis would result for these reasons:

▪ Smaller documents and partial documents are easier to review. A less fatigued reviewer will review the end of a document with the same care as the beginning.

▪ Smaller documents and partial documents, requiring less lead time, will be more current when reviewed.

▪ Because smaller documents and partial documents take less time to produce and review and have a more narrow scope, fewer personnel grind to a shorter stop.

▪ A larger number of smaller reviews makes each review less important. As reviews become a routine part of development activities, developers are less impacted by "non-productive" work, and the "dog and pony show" atmosphere is reduced.

▪ "Show stoppers" discovered will likely be limited in scope, delaying development for a shorter period.

▪ Spreading out each review permits better contractor man-power leveling by overlapping the requirements analysis, design, and coding of separate elements. The advantages offered now by the separate review of different CSCIs and incremental reviews would also result if applied to smaller, relatively independent "chunks" of software.

▪ Major problems found earlier, will be easier and less expensive to correct.

Although 2167 allows incremental reviews, the linear nature of the classical life-cycle, with formal review bottlenecks between phases prohibits the use of recursive "design a little, code a little, test a little" methods. Although 2167 permits a small number of incremental PDRs per CSCI per build or release, it does not permit methods such as Object-Oriented Design in which small amounts of code (about 1KLOC) are recursively designed, coded, and tested during each pass through the method. On large projects (over 100KLOC), it is impractical to hold several hundred traditional Critical Design Reviews (CDRs) and PDRs. The bottleneck nature of formal reviews prohibits one from coding and testing as one goes—an important aspect of methods that permits one to incrementally validate the evolving design.

Since not all methods produce the same intermediate products in the same order, the scope of current reviews is sometimes inappropriate. The timing and the scope of the results of certain design activities are set by the timing and nature of the formal reviews.

▪ **Critical Design Review.** The process of 2167 does not account for the evolving nature of the CDR.

Due to the high modularity and low

complexity of well-designed Ada software, the lack of distinction between Ada PDL and Ada code, and the design aspects of the Ada specification, the classic purpose of the CDR (to review and approve unit-internal logic prior to coding) is no longer relevant. Coding the Ada specification is a design activity. PDL is not needed to document the logic of the body of many units because of their trivial size and complexity. One should code the body once started since it involves little added work and allows one to use the compiler to partially check the results prior to any semi-formal review. By performing the unit testing immediately, one can also validate the design as one goes.

Use of the same language for design and implementation causes a non-trivial problem in defining what is design and what is code. This complicates determining the scope of the SCR as currently defined.

Requiring a formal review of the "detailed design," at CDR and prior to coding and unit test, prohibits the contractor from using recursive software development methods that result in the hierarchical top-down design, code, and test of very small amounts of software. A major constraint.

- **Delay Problems.** On large projects, the 2167 process results in a long delay between requirements definition and implementation. During this time, contracting agency personnel are likely to change, causing the project to be subject to different "hot buttons" and large changes in requirements. Contractor personnel turnover is also likely, resulting in the loss of the rationale for certain key decisions.

- **Lack of Intermediate Software.** Useful software does not exist until the end of the development life-cycle. Only at the end of a build, release, or project is there a product that validates the requirements and design, is testable, and is subject to user scrutiny. This is a major argument for prototyping that tempts some contractors to rush into coding without a systematic software development method or adequate preparation.

- **Prototype Inhibition.** The use of prototypes, long recognized as a productive technique in every engineering field, is something for which 2167 does not adequately allow. The development life-cycle is inconsistent with that of several prototyping models. Only by building prototypes can one verify the feasibility of a paper design. Perhaps this is why hardware engineering has advanced beyond software engineering.

- **Reuse Inhibition.** The top-down development process seems to assume that all software will be built from scratch, thwarting a major goal of Ada, production and use of libraries of reusable software. Isolated references to the importance of reusability in 2167 are insufficient if the general process inhibits it.

- **Automation Inhibition.** To increase software development efficiency and the quality of software and its documentation, significant portions of the process must be automated. This includes, but is certainly not limited to, the production of documentation. 2167 seems to assume all software is to be built from scratch by performing all activities of the prescribed process in accordance with the standard life-cycle. Yet automation of significant portions of the life-cycle will have a major effect on the description of these required activities and the associated reviews. It is not at all clear that this can be adequately or efficiently handled by merely deleting requirements from 2167, the only method of tailoring allowed.

A counter argument is that the government will only pay for automation if it can be proven cost-effective. In judging cost-effectiveness, tradeoffs should be performed. Does readibility suffer from automation so that reviewing and maintenance take longer and cost more. If such questions are ignored until too late, the project will suffer.

These valid questions must be weighed against the quality and productivity goals of automation. Manual production opens the door to human error. Much of documentation consists of translating design information from the format working documentation or software structure into the required format of the deliverable documentation. When performed by hand, transcription errors result in an inconsistency between the software and its documentation. Manually produced documentation is not always updated to incorporate changes in design, due to the large effort involved. This can also be said about "translating" requirements into design and design into software. One can argue that automation should eliminate certain classes of errors, making the reviews more productive since reviewers will not need to spend time finding such errors.

These arguments for the current 2167 process have been raised:

- The process ensures production of intermediate products that can be used to restart the process should development be stopped midstream. However, all contractor-proposed processes should do so.

- The classic waterfall life-cycle of 2167, with its phase boundries and

milestones, brings a structured management process to software development. However, this is not the only structured life-cycle that defines phase boundries and the milestones needed for management.

Recommended Solutions. Before looking at specific recommendations, we should consider the DoD needs that formed the foundation of the 2167 process. DoD needs to:

- Understand the process, methods, and life-cycle used to develop the deliverable software.

- Be confident that the process, methods, and life-cycle are cost-effective and will result in product (delivered on time, within budget) that works as expected and is maintainable.

- Properly oversee the development process to be reasonably certain there will be no major surprises.

To solve the problems listed above and ensure DoD needs are met, I recommend that DoD:

- Rename 2167 "Defense System Software Acquisition."

- While keeping the DIDs, modify DOD-STD-2167 from a development process standard into an acquisition process standard.

- Provide industry with financial incentives to do good work.

- Modify the acquisition process to account for differing needs of software and hardware developers.

If DoD considers these too radical, I propose 2167 at least be modified to make it clearly independent of any required default software development process, methods, or life-cycle. To accomplish this:

- Replace paragraph 4.7 with the following: "The contractor shall propose in the SDP and detail in the SSPM a systematic software development process, methods and tools that are appropriate for the application and implementation language. Once approved by the contracting agency, the contractor shall develop all SCSIs in accordance with these methods and tools."

- Remove all method-dependent terms (e.g., PDL, top-down).

- Delete paragraphs 4.1.1 and 4.1.2 and replace paragraph 4.1 with the following: "The contractor shall propose in the SDP a specific, structured, life-cycle model with well-defined phases that provides adequate intermediate products and milestones appropriate for the application, implementation language, and proposed software development process, methods, and tools. Once approved by the contracting agency, the contractor shall develop all CSCIs in accordance with this life-cycle."

- Remove all requirements governing contractor internal processes (e.g., informal testing, controls and visibility requirements concerning non-deliverable items, the use of software development libraries, development configurations).

- Decouple 2167 from the reviews mandated by MIL-STD-1521. Add the following paragraph to 2167: "The contractor shall propose in the SDP, formal and informal reviews based on the phases of the software development life-cycle model. Once approved by the contracting agency, all CSCIs shall be reviewed in accordance with these proposed reviews."

- Decouple the static software hierarchy from the software development and review process so that the contractor first develops the proper software structure and then decomposes it into a static hierarchy for purposes of SCM.

- Add criteria to DOD-HDBK-287 for evaluating the contractor's proposed software development process, methods, life-cycle, and reviews.

Conclusion. DOD-STD-2167 is a tri-service military standard that establishes and mandates a uniform software development process for Mission-Critical Computer Resource software. It will be applied to a great many projects and have a major impact on the way software is developed in the US.

The draft of DOD-STD-2167A contains several improvements that begin to answer some problems mentioned above. During review of this draft revision, members of the Ada community identified numerous problems concerning the compatibility of 2167 with the proper use of Ada and modern software development methods, some of which are specific to the 2167 process, methods, and life-cycle. Others are problems with software process standards in general.

DoD goals would be best served if all process, methods, and life-cycle requirements and defaults were deleted from 2167. DoD should specify what it needs and leave the contractor free to propose the best application and implementation language-specific way it should be developed. Only by promoting innovation and rewarding achievement can DoD ensure the use of the best process, methods, and life-cycle for the application and implementation language. **DS&E**