SOFTWARE ACQUISITION MANAGEMENT

In a Nutshell

James H. Dobbins

Volumes have been written about acquisition management, and part of that mass of information discusses software management. The problem with software acquisition management information is that it is voluminous and scattered all over.

In law school, some of the most popular little books are the "nutshell" series published by West Publishing Co.: *Contract Law in a Nutshell*, etc. The objective of this article is to provide you with a nutshell capsule summary of information you need for software acquisition management. I hope you find it useful.

Why Software Management Is Difficult

Software management is difficult because of uncertainty and risk (big surprise?). It's usually very difficult to recognize software risks as they surface. You generally see software risk later, sometimes much later, when it is no longer a risk but has become a problem and costly or even impossible to correct. But, you should see it and plan for it much earlier. The software risk driven problems are *usually* management, not technical. During acquisition, we seldom consider the things that "kill" us later. We wear cost-proposal blinders. Acquisition managers need to do eight things better.

Mr. Dobbins is a Professor of Systems Management at the Defense Systems Management College and Course Director for the Management of Software Acquisition Course.

Eight Cost-Proposal Blinders

1. Use metrics properly. Understand metrics implications.

2. Understand the implications of software process capability maturity.

3. Understand when we do and don't need an independent verification and validation contractor.

4. Understand system performance implications of software quality.

5. Don't let low software cost blind us to its potential effect on the system.

6. Do software requirements a *lot* better.

7. Learn how to *build visibility* requirements *into the RFP/Contract*.

8. Learn how to establish sensible software source-selection criteria.

Twenty-three Sources of Software Risk and Uncertainty

Having done those eight things better, acquisition managers need to understand the following 23 sources of software risk and uncertainty.

1. Government and contractor lack of understanding of the effect of *software process maturity*.

2. Lack of software experience in top management. Ignorance of the law is no excuse.

3. Lack of understanding of when, how and what to measure (software metrics).

4. Lack of understanding of how best to get current information/visibility.

During acquisition, we seldom consider the things that "kill" us later. We wear cost-proposal blinders.

5. Lack of understanding of how to use measurement (metric) information.

6. Lack of understanding of full spectrum of contractor testing.

7. Lack of understanding of how to utilize the test concept in software fully.

8. Lack of understanding of the fundamental and significant differences between software and hardware concepts for common terms such as reliability and availability.

9. Failure to incorporate proper rigor and knowledge into source-selection criteria.

10. Lack of understanding of how to plan for software risks so the risks stay risks instead of becoming problems.

11. Lack of understanding that software isn't magic.

12. Lack of understanding how software fits into the system engineering process.

13. Failure to understand software architecture and the effects of changes. Why you can add a window to the top floor of the nine-story software building, but you can't add a basement.

14. Lack of understanding that software engineering is a discipline; a process.

15. Lack of understanding of the implications of insufficient time allocated for: Software Requirements, Software Design, Concurrent Engineering, In-Process Quality Analysis, Design for Reuse, PDR/CDR, Error Correction, Error, Analysis and Error Prevention.

16. Lack of understanding why highly-structured languages like Ada are good for the DOD software engineering environment where most developers are process immature.

17. Lack of understanding that good software development is event driven, not schedule driven.

18. Lack of understanding of the software acquisition life-cycle activities and their purpose.

19. Abdication of decision making to contractors because we don't want to deal with software issues.

20. Lack of trust of good contractors.

21. Too much trust in less-than-competent contractors.

22. Failure to make sure you have systems engineering capability in the program office staff.

23. Letting esoteric technology issues cloud your software decision making ability.

Twenty-nine Rules for Managing Software Acquisition

The 29 rules for managing software acquisition are as follows:

1. Learn not to be afraid of software. Put your arm around it and give it a hug.

2. Understand and manage the software development *process*.

3. Understand the greatest strength of software: flexibility.

4. Understand the greatest weakness of software: flexibility.

5. Learn and recognize the software issues that can kill you.

6. Understand that the software development process is manageable as is any engineering process.

7. Learn the importance of software configuration management.

8. Learn the different kinds of software tests, and when and how they can be used best.

9. Let the requirements definition process happen. Don't close it too soon. Keep the user involved. Understand how to do good prototyping.

10. Learn that software requirements changes after critical design review (CDR) can kill your system cost, schedule and performance. Don't let the user or contractor jerk you around after CDR. Don't jerk the contractor around after CDR.

11. Recognize that a software preliminary design review (PDR) or CDR done too early might as well not be done at all.

12. Recognize that once you are beyond the software B5 spec (software requirements spec), the user is probably useless in reviewing software documents.

13. Never underestimate what a good, process-mature, software contractor can do for you to pull you out of a cost/schedule/performance hole.

14. Never forget that a process-immature software contractor will be virtually useless in pulling you out of a cost/schedule/performance hole.

15. Never forget that if you hire a process-immature software contractor, your success or failure on the contract will happen in spite of your skill as a program manager, not because of it.

16. Never forget that software has no production cycle. The first article is "it" and if you fail, you're dead in the water.

17. Always think *risk* at every step of the software acquisition process. Remember *risk* is always a potential. When the risk event happens, it has become a problem.

18. Trying to manage performance outcome is a blueprint for disaster. Learn to manage process.

19. Always think software support (PDSS) at every step of the software acquisition process.

20. Learn that for unprecedented systems, the waterfall model of software development doesn't work. Don't let DoD-STD-2167A drive you over that waterfall in a barrel.

21. Learn that for unprecedented systems, you must prototype software as you have to prototype hardware. It takes time, but that's the way you learn what requirements are. Let this happen and move PDR/CDR if you have to.

22. Learn to get your metric information and software status from your computer resource working group (CRWG) and monthly reviews, not just in CDRLs which take too long to produce. The data are too old by the time you get it.

23. Get software expertise and systems engineering expertise in your program office, even if only on a consulting basis.

24. Understand that commercial off-the-shelf (COTS) software seldom works in custom-designed unprecedented systems, especially in embedded system software.

25. Never incorporate a Non-Development Item (NDI) or COTS software in a system without first having the contractor evaluate the feasibility *in the intended environment*. It seldom

works as well in the intended environment as you had hoped.

26. Never force a certain technology, like object-oriented design, on a contractor unless you really need it. It can be like asking a toddler to drive an Indy 500 race car.

27. Understand that if you design for reuse, it will cost more. The payback comes later, possibly on the next program.

28. Learn to use, and tailor, software standards, including industry standards. Read them. Understand what they require. If you don't need a part, tailor it out.

29. Set up and use a Computer Resources Working Group. Make sure they produce and keep up-to-date your Computer Resources Life-Cycle Management Plan (CRLCMP). Watch your interfaces. Get the Interface Control Working Group (ICWG) in place early and use them. Make the CRWG interface with the ICWG and other working groups (WGs).

Remember that risks are always a future consideration. Once a risk event happens, it is no longer a risk; it is a problem.

Recognizing Software Risks

Remember that risks are always a future consideration. Once a risk event happens, it is no longer a risk; it is a problem. How far in the future you can spot a risk is a function of how well you do measurement and strategic planning. We look at risk in terms of probability and severity. If low severity, you may not care if it occurs. If high severity, you had better care a lot. If high severity, but low probability, you should always get nervous. If the probability of a high-severity risk is not zero, you always worry about when your number is coming up. You must build fall-back positions into your strategic planning. There are questions you need to ask yourself about different types of software risk.

Feasibility Risks: Can we do the job? Is the technology there? Can this contractor do the job? How much experience does he have? Is the task unprecedented for this contractor? Do you know all the requirements? How sure are you? Are the users involved in requirements definition? Do you have to prototype? Has the contractor ever done prototyping? Should you use an acquisition strategy of evolutionary acquisition?

Engineering and Producibility: Do you have to do evolutionary acquisition (requirements not fully determinable for Block 1)? Can you produce a working system for Block 1? Does the contractor have requisite skills?

Do you have to supplement the contractor with a directed subcontractor? What is the contractor's process capability maturity? Do you need to do a Software Capability Evaluation (SCE) or equivalent? Do you have a means for doing a software pre-award audit? Have you factored these possibilities into your source-selection criteria? Who can you get to do the audit? What is the contractor's meaningful experience with the language (Ada)? In what environments? Does the schedule give the contractor enough time for requirements definition?

Are you and the contractor working together to minimize post-CDR requirements changes? What kind of communications processes have you set up with the contractor to address and control technical issues? Does the schedule give the contractor enough time for design? Do you understand the contractor's software test program? Is it adequate? Do you and the contractor understand which metrics the contractor is using and the utility of the information? Does the contractor management use the information? Are meaningful metrics being used in all life-cycle phases?

How does the contractor select and use Computer-Aided Software Engineering (CASE) tools? Which ones? Why were they chosen? Are they force multipliers? Is the contractor dependent on a subcontractor? How well do they manage subcontractors? How do you know? What evidence? Do you understand the contractor's software quality process? Do you understand the contractor's software configuration management process?

Cost Risks: Do you expect the contractor's cost proposal to be accurate? Why? What will you do if it isn't? What if, by the time you hit CDR, you are seeing a 40-60 percent overrun? What are your fall-back options? Plan for these well in advance, because this is likely to be what happens. Cost estimates on unprecedented systems are usually a lot lower than eventual reality. We simply don't know how to do it better.

One thing that helps is a well-written Statement of Work (SOW). Do your strategic planning early. Plan for cost overruns and alternate actions you must take. Make sure you

collect the necessary metrics often to spot potential cost overruns as far in advance as possible. Always know the difference between requirements and desirements. Never cut requirements; just desirements.

Rules to Follow

Follow the next 17 rules to keep software contracting from "biting" you:

1. Invoke the desired standards (DoD-STD-2167A, DoD-STD-2168, MILSTD-1521B and DoD-STD-973, or industry standards like IEEE-STD-982.1). Invoke these standards in Section 3 of the SOW, and anywhere else you need them, not just Section 2. Tailor them where necessary; never invoke "blanket" standards. Always know what you are imposing, and what you are tailoring. Read the standards.

2. Watch for pitfalls in chaining standards between specification documents. Treat each specification as a stand-alone in terms of standards invocation.

3. If you want metrics, ask for them. If you want specific metrics used, say so in the RFP/Contract.

4. If you want metric data provided, say when and how and how often.

5. Ask the contractor to describe their own:

—Software engineering environment, including CASE tools
—Software management, including subcontractor management process
—Software development processes and tools
—Error correction/analysis process
—Software test and evaluation process; all of it, at each life cycle phase.

6. If you want to see *CASE* tool output, say so. What, when, how and how often.

7. If you want specific processes used, say so.

8. If you want *software root cause analyses* instead of bug fixes, say so. Allow time for it. Why would you want this? Because the industry average is that 14.7 percent of software error (bugs) fixes are bad fixes, and fast-bug fixing can turn your software into spaghetti code overnight, and you won't have enough money left to recover. Remember software's biggest weakness — flexibility.

9. When you know what you want, make it part of the source-selection criteria. Use the criteria to drive you to the most capable, not just the cheapest, contractor.

A lowest-price software contractor may be your most-expensive choice. It may be your best choice. *It Depends.* It depends on their software engineering and quality processes, metrics, use of *CASE* tools, and their process capability maturity. If they are process capability *immature*, don't know how to use *CASE* tools, and are low bidder, run, don't walk, to the next contractor in line. You'll be sorry if you don't. When you get the proposals, read them critically. Read between the hype, between the chest-pounding, and get to the real meat. The rest they must include because we expect it. But, listen to what they say they really do. Then do a pre-award audit or software capability evaluation.

10. In proposal responses:

—Read the Software Development Plan (SDP)
—Read the Quality Plan
—Read the Configuration Management Plan.
—Read the Test Plan
—Read the Software Subcontractor Management Plan.

11. *Pay close attention* to what they say about subcontractor management. Ask for their subcontractor management plan.

12. Watch out for *data rights* issues.

13. Look for front-end quality processes, like:

—Design and code inspections
—Complexity analysis *CASE* tools
—Requirements generator *CASE* tools
—Code generator *CASE* tools.

14. *CAUTION*: If they use the Integration and System Test phases as the primary time to find and fix software bugs, your system very probably will be late, will overrun cost, and probably will have unsatisfactory performance no matter what you do. If they skip Unit Test, expect big problems later, but ones that are, perhaps, not found until a disaster during operational test or actual use — *big* costs, and possible injuries.

15. Look for whether they design for maintainability.

16. If they say they use a development process, like design and code inspections, go back and ask them to describe it and how they use the results. Many contractors give lip-service to good processes but don't understand well enough to use them properly. Don't ever forget that no matter how good a proposal looks, more than 80 percent of the DOD contractors are software process capability *immature* (Initial level on SEI scale). It is one of the most, if not the most, serious hidden risks we face in contracting for software.

17. *CAUTION*: Software cost models are everywhere: COCOMO, REVIC, PRICE-S, etc. None of the results are worth anything if you can't get a good estimate of software size. All the models are dependent and results are biased by software size. Software size is a sensitive parameter. *Therefore*, for an unprecedented system, don't expect contractors to provide accurate cost proposals. They can't. Neither can we. Nor, can anyone else. The best software cost estimate will come from a technically-capable and process-mature contractor with a good database, who collects good metrics, and who had experience a few times. That probably means a contractor at the Defined level on the SEI scale. The process capability immature contractors don't have good databases, or databases with valid data for estimating cost. Even if they have a database, their immaturity biases the cost data.

What's All This Stuff About Metrics?

The DoDI 5000.2 requires using metrics in software management. It doesn't say how, which, when or what for. That's up to you. Congratulations! Metrics are only a number. Their only utility is in understanding the infor-

mation baggage that goes along with the number; the implications; what they tell you that *helps you make a decision. They answer a question for which you need an answer.* Don't look at metrics as isolated things; think in terms of sets of data that you use together to get a total picture of issues which need decisions.

Computing metrics for its own sake, or because something is measurable, or just to check-off a DoDI 5000.2 paragraph is a waste of time and money. Good contractors know this. Process capability immature contractors do not. Good contractors compute meaningful metrics as a matter of course because it helps them control their processes and make good decisions. They use metrics as a tool to achieve continuous improvement in their processes and control their technical efforts and costs. Process-immature contractors compute metrics because the government makes them; they whine, complain about the cost, and most don't understand implications of measurements they do take. They look at metrics as an unnecessary cost-driver.

Two fundamental types of metrics exist. These are generally classified as *management metrics* and *quality metrics*. Don't be confused by the nomenclature. Both are important to management and to technical personnel.

If the slope does not level off to a very low value by and after CDR, you're at big-time risk for meeting your threshold (forget about the objective). This junkyard dog can bite hard. When it does, effects often are unrecoverable.

Most metrics are utilized best when presented as trend data as opposed to point-in-time data. There are exceptions but this is a good general rule.

Metrics You Need for Any Development Program

Requirements Volatility. How rapidly are changes being made over time to software requirements? What is the rate of change? Plot the cumulative number of changes over time and watch the slope of that line. Do it for the entire program, and for each Computer Software Configuration Item (CSCI) independently. If the slope does not level off to a very low value by and after CDR, you're at *big-time risk* for meeting your threshold (forget about the objective). This junkyard dog can bite hard. When it does, effects often are unrecoverable. It throws you almost automatically into a cost-and-schedule problem. You and your contractor *must* manage this from *early on*.

Software Size and Size Growth Over Time. Disaggregate the data. Don't just look at total size, but also at the size dynamics of each component CSCI; also, each type of code (function and language). Keep track of these values separately for New Code, Reused Code and Modified Code. Understand the cost implication of changing the original projections of new, modified, and reused code. The total lines of code may not change, but if *new* code goes up, and *reused* code goes down, the cost will go up because it is more expensive to build and test new code than to reuse old code. Take continuous size projections at all *life-cycle* phases. The initial contractor estimates are usually low, often considerably low. You must match size growth projections against hardware capacity.

Personnel. Same sort of thing as software size. Track separately the changes for total personnel, but also the mix of experienced and inexperienced. Watch for changes near major review times.

Computer Attributes. What is the *capacity*? How is the software size growth affecting the hardware capacity? Require at least 50 percent reserve memory. Does the slope of the software growth curve make you nervous? If the memory reserve capacity drops below 30 percent, you *are* in trouble. It's no longer a future tense; no longer a risk; it has become a problem. What is the *processor speed*? Can the computer speed match software requirements?

Software Volatility. Don't worry about counting how many trouble reports you have open today. It's useless information. Rather, what is the rate of change for the software after Unit Test? Plot cumulative changes (trouble reports and requirements changes) over time for the whole system and for each CSCI independently; also by trouble report severity class. For trouble reports, *at the midpoint of the computer testing period*, or earlier, the slope of this cumulative curve should change rapidly and begin to approach zero as an asymptote. If it doesn't start doing that by the midpoint, you are in trouble. Get with your contractor and understand the problem. Is it in one CSCI, or is it system wide? Probably one or two CSCI. Work out a plan of recovery.

Complexity. Understand software complexity, especially Cyclomatic Complexity (also called McCabe's Complexity). It is a *very powerful* metric. Use a CASE tool to help analyze the code. Many analyzers can analyze Ada. If you have one that does, you can analyze the design as well as the code if you are using Ada as a program design language (PDL). Understand that when you compute the complexity number, this unitless number has 11 implications.

Eleven Implications of the Complexity Number

1. It is the exact number of independent paths through a software CSU (module).

2. It is the *minimum* number of test cases you must have to test each part of the module at least once, assuming the programmer recognizes the independent paths and develops separate tests for each independent path. (Some of the CASE tools automatically highlight the independent paths, and automatically compute the test conditions you need to test that path. All the programmer must do is copy the test conditions into the tests written.)

3. If the number goes above 10 for a module, the module begins to be error-prone.

4. The higher than 10 the module complexity number is, the higher the risk to the system from that module.

5. If the complexity for several modules goes above 30, get somewhat nervous and take corrective action.

6. If the complexity for several modules goes above 40, get *really* nervous and take strong corrective action.

7. If the complexity for several modules goes above 50, panic. A majority of the module paths will not be tested in Unit Test, or any later test; therefore, a significant percentage of the software will be delivered completely untouched by any test. This is an *enormous*, often-hidden risk to your system, especially if software failure could have life-threatening consequences.

8. Most process capability immature contractors don't understand anything about software complexity.

9. If a high-complexity module is in a critical path, especially a safety critical path, it is a time bomb waiting to explode.

10. Look at the complexity values for the entire system, and for each CSCI. Compute the average complexity but also look at the complexity distribution curve (a histogram). The average complexity can be deceiving. Look at how the module complexity values are distributed from lowest to highest values.

11. During computer-based testing, have the contractor compute the complexity and get complexity graphs, before and after each module change, intended to fix a trouble report. This is to ensure the change did not damage the module structure and drive up the complexity. This lengthens the lifetime utility of the module and lowers its total life-cycle cost.

Software Reliability. Watch out for this one. Make sure you understand the implications. Hardware reliability is a well-understood, valid and useful discipline. It is important in understanding maintainability issues and spare-parts provisioning. This reliability is usually computed as a mean time between failure (MTBF) or mean time to next failure (MTTF). These values have real meaning for hardware, and it says something about the hardware itself.

The MTBF is a measure of the expected time between failures of system components. If you have a fighter aircraft MTBF of 10 hours, and your average mission duration is 20 hours, you will not be able to fly a mission. We need this information for hardware to determine when to expect to change the hardware. We change hardware to get it back to its original condition, because it breaks. After you swap out the broken part, the old value of MTBF is still good.

For software, MTBF doesn't mean anything except in a gross sense. It may not be telling you anything meaningful about the software. There are many software reliability models that have been developed and work just like the hardware models. They are statistical Bayesian or Poisson models. They compute MTBF, which is supposed to be a measure of the expected time between failures. If you have an MTBF of 10 hours, what does this mean for software? Is it telling you anything about the software itself? Software doesn't break. You never change software to get it back to its original condition. Once you change software, the prior MTBF measure is no good.

From where does the data for software MTBF come? It comes from testing and operational use. Suppose you develop a mediocre test and run the software, and you get a certain number of errors. You enter this data into the reliability model and get a number that you think represents the reliability of the software being tested. Now develop a different, more stringent test and repeat the process. You get a different number of errors and, consequently, a different reliability number—same software, same model. Are you measuring the reliability of the software or the test you wrote? Once you know what data condition

causes an error to manifest itself, you can cause the error to happen as often as you want by recreating that environment. Software is data environment responsive, and it reacts to data environments.

It will not fail until it is asked to do something it was not properly designed to do. Until you hit that condition, it works fine and will continue to for as long as the computer runs. Once you find the condition that causes the software to fail, you can turn your 10-hour MTBF into a one-second MTBF in a heartbeat. You don't want the system to hit that condition during a dogfight.

This is why *software complexity* is important. You must know the different software paths and know you have tested all of them. High-complexity modules impede your ability to do this, and unexpected operational software failure results. Properly using things like software complexity analysis is how you control software reliability; that is how you *manage* software reliability. Running an MTBF model is more of a diversion than a help.

Don't make the mistake of asking the contractor to give you a system-reliability number that is an arithmetic combination of hardware and software reliability numbers taken from statistical models. There is none. Conceptually, the two components are as different as comparing apples and oranges.

Managing Software Testing

Software testing should include human testing and computer-based testing. For process-immature contractors, it is usually confined to computer-based testing. Human testing is done before Unit Test. Computer-based testing is Unit Test, Integration Test, System Test, and all subsequent contractor and government software testing.

If you do good human testing, your computer-based testing processes will be controlled, manageable, and will give the management flexibility needed.

Human testing is desk checking (almost worthless); walk-throughs (can be good, but you never know in advance); and, software inspections (the best, with highly-consistent and predictable results). Inspections, when done properly, will remove a minimum of 70 percent of life-cycle defects from the software before Unit Test. Consequently, the traditional computer-based testing processes, instead of being the primary place to find and fix software errors, become a validation phase. This makes your management job orders-of-magnitude easier, more controlled, and gives you options you would never have otherwise.

Process capability immature contractors often have not heard of software inspections; if they have, they do not know how to do them properly, or how inspections can help. All they see is the cost, not life-cycle payback and not quality drivers.

If you do good human testing, your computer-based testing processes will be controlled, manageable, and will give the management flexibility needed. You can recognize when you have tested enough by the slope on the cumulative error detection graphs. Without human testing, all bets are off. You may be in a scrap continuously (and don't forget the 14.7 percent bad fixes); the complexity may grow continuously and exponentially; and, you will test until there is no time, money or both, and hope for the best to meet the threshold.

Put human testing in your RFP and contract for software intensive programs. *After Unit Test, the contractor* should have the software in a configuration-controlled test library, to which changes are made only with approval of the Software Change Control Board (SCCB), and made only by the Software Control group (not the programmers). Remember, Unit Test is the last test phase where the focus is inside the module. After Unit Test, the tests focus primarily on interfaces, between CSUs, between CSCs, between CSCIs, and between software and hardware. It may be an informal test, but its importance should never be underestimated.

Errors detected during tests using the configuration-controlled libraries should be categorized by severity. Have the contractor keep the error-detection rate charts for the total system, each severity type, each CSCI, and each severity type within each CSCI. Contractors need that data as much as you do, whether they realize it or not.