

# Requirements and COTS-Based Systems: A Thorny Question Indeed

David Carney



In this issue, we will focus on one of the most common words in our vocabulary, but one that has caused more than its share of trouble since people started building software systems: *requirements*. It is a powerful word. The very sound of it makes old-timers, those with 2167-A scars, roll their eyes and think of waterfalls.

## Dare we say the word?

Let it be said first off that the issue is not entirely a COTS issue; people started arguing about requirements long before COTS entered the picture. But whatever else is true, the growing use of COTS software has added fuel to the existing controversy, and it is certainly valid to speculate on how requirements play a role, given the reality of how we build systems these days. In truth, many people have already taken a very definite stand and in some quarters, use of the word “requirements” has become Politically Incorrect. (Just as in George Orwell’s novel *1984*, where troublesome words are condemned to oblivion, even saying “requirements” these days can get you into trouble, especially if you’re locked in a room with the Spiral/Evolutionary/Prototyping crowd.) Yet if carried to its logical end, this extreme position—don’t even *think* about “requirements”—is somehow troubling: surely the person who has a need for (and who pays for!) a system to come into existence should have control over what that system is required to do.

As I consider this gnarly question, and speculate on the related issues of requirements, COTS products, and software systems, I hope the reader will bear with me as I wander down some unfamiliar paths. Though it may not be immediately obvious how a short side trip through literary history may be relevant, I promise that the excursion has some value to the topic at hand.

## A visit to the dictionary

A good first step in resolving any dispute is to get agreement on what we are talking about, so it seemed logical to take this step with “requirement.” The *American Heritage Dictionary* is direct and unequivocal:

*Something that is required; necessity; something obligatory; prerequisite.*

But *Webster’s*, while generally agreeing with this, adds just a bit of ambiguity:

*Something wanted or needed.*

Whoa there. This is a little different, since this definition seems to open the door a bit from the absolute, unconditional (i.e., obligatory, requisite) into what is simply desired (i.e., wanted). So off we go to the ultimate authority, the venerable *Oxford English Dictionary*, and there some interesting lessons can be learned.

For one thing, in the *OED* the very first definition of the word “require” is “to ask,” and the first several entries for the word, stretching back several centuries, are all variants of the idea of wanting; they all imply asking for something. Reaching back to Chaucer’s *Canterbury Tales* (1386):

*The firste thyng I wolde hym requere, he wolde it do.*

*(The first thing I would [request of him], he would do it.)*

To be sure, subsequent definitions (which generally reflect later historical uses of the word) gradually make the word more imperative, and begin to imply necessity, obligation, or insistence. But the initial definition of “require,” maintained for several centuries, has a much stronger flavor of asking than it does of demanding.

The same is true of the initial definition of the word “requirement:” it is a request rather than a demand. The earliest use of “requirement” given by the *OED* (from 1530) is cited in precisely this sense:

*My Lord Cardinal, that obtained his legacy by our late Sovereign Lord’s requirements [e.g., petitions] in Rome.*

The more familiar, absolutist definition of “requirement,” as

*a condition which must be complied with*

is not seen until 1841, several centuries after the word entered the language:

*Has any individual...ever yet come up to their...requirements?*

So what does this have to do with software, and especially COTS software? Quite a bit, actually.

## **The evolving notion of software requirements**

For one thing, it is fairly obvious that the term as usually used in the software engineering world refers to the later meaning of something obligatory and necessary. This was certainly its meaning in the heyday of the “waterfall” process, and at the time that the “waterfall” process was developed and popularized, it was a reasonable principle. The firm grounding provided by a collection of system requirements is the basis upon which many complex things—buildings or airplanes, for instance—are constructed, and it was perfectly normal for software engineering to share that understanding. The requirements specification was the source of all subsequent activities; in the very real sense, the requirements were in the driver’s seat.

There were also reasons of historical circumstance. Most software systems were new, and therefore unprecedented. It made sense, when creating something new out of whole cloth, to specify it as rigorously as possible, through a collection of required capabilities, and then to adhere to that specification. The success of the end product was determined by a test of whether each required item of that specification had been successfully implemented.

However, notwithstanding this belief in the value of a good initial specification (and certainly long before COTS use became commonplace), there were many problems with the requirements-centric paradigm. For one thing, some systems had very long development times, and rapid advances in software technology often caught up with the system—and its requirements—and left the developing system behind. Thus, users

sometimes got “new” systems that were obsolete before they were fielded. Another problem lay in the very act of specifying the requirements, since it was often true that a system’s users were not the ones who wrote the requirements. Thus, users sometimes got systems that did a lot of nice things, but not the things that they wanted or needed.

And the essential flaw in the concept was that for a complex software system, it was extremely difficult (and perhaps impossible) for a full set of system requirements to be known at all during the initial stages of development. The inevitable result was that, particularly for large systems, requirements specifications contained too much detail, and often contained inconsistent or contradictory requirements, which demanded some sort of negotiation between the competing requirements. (More on this below!)

One admirable solution to this problem lay in the notion of a “spiral” development process. First described by Barry Boehm, this concept assumes that since the full set of requirements can generally not be known when a system is begun, all attempts to “nail down” the requirements will very likely fail. To resolve the dilemma, the development process should make frequent use of prototypes, have heavy involvement with the system’s end users, make ongoing refinement of the developers’ understanding of the system’s goals and mission, and so forth. Thus, the development process actually consists of a series of “spirals,” zeroing in from a very general to a very precise knowledge of the system and what it does.

This notion of system construction is a clear improvement over the straight line “waterfall;” a spiral development process, in one form or another, is widely used on many systems today. Note, however, that an implicit idea in the spiral process is that while the requirements probably cannot be determined initially, they can eventually be determined by appropriately spiraling, prototyping, and refining. Although perhaps not known initially, the requirements for the system are somehow there, invisible perhaps, but gradually becoming more and more visible and explicit. And, presumably, still absolute.

### **Enter the COTS paradigm**

It is not surprising that the growing use of commercial products (whether truly “off-the-shelf” or maybe needing a little tinkering) has further eroded the possibility of making an initial definition of system requirements: it is demonstrably foolish to set forth some abstract set of necessary features and then to hope that a group of COTS vendors have already created components that will just happen to meet those needs. So in a COTS-based paradigm, the concepts embedded in the spiral process—especially the idea of frequent prototyping—are especially valid and useful.

But in a more subtle manner, it may also be true that using COTS in our systems is really a semantic revolution: we may well be reverting to the centuries-old definition of “requirement” as something we want, perhaps want very much, but that may turn out to be something we are willing to do without.

Consider how easily we are willing to use the words “requirements tradeoff.” Strictly speaking, this phrase is an oxymoron: if something is a requirement (in the absolute sense), it is utterly necessary, and cannot be dispensed through any sort of negotiation, no matter what. Yet it is a familiar phrase. We used it back in the 2167-A days; that is how we decided between competing requirements. We use it even more today, and with justification. For there are many circumstances where we really do trade off our “requirements.”

Suppose, for instance, that some COTS product is available and ready to hand, and we need a usable system urgently. Add to this mix a strong desire to maintain technological currency in a world that is changing with blinding rapidity. In such a case it is quite possible that we are willing to reconsider features that we thought were mandatory, reexamining them in the hard, cold light of pragmatism and practicality. Given sufficient cause (which can span a range from utility to capability to sheer economics), a “requirement” might well be abandoned; in essence, we demote it, since it is no longer a “necessity,” and is by definition something that we are willing to do without.

I am not preaching anarchy here. There are still absolutes, and there must exist some set of capabilities and features that are not optional, that cannot be traded off, because without them our system will fail. But that set of items is usually smaller than the overall set of wants and desires we typically connect with our software systems. For one thing, we are now worried much more about dollars: part of the new acquisition reform initiative is the notion of “cost as an independent variable” (CAIV), premised on this very principle of requirements negotiation. According to CAIV, requirements tradeoff against cost should be explicitly considered in the design process; those “requirements” that escalate costs are to be revised, a far cry from the days when cost was a non-requirement.

Distinguishing the real absolutes from all the rest is the key. We need better self-knowledge about the bottom line of what we demand from our systems, and we must distinguish that bottom line from all of the other system goals. Like the familiar movie scenario of the gullible tourist entering the bazaar in the Casbah, we need to know clearly what we want, what we will settle for, and what we are willing to pay. Otherwise, the movie scenario has a *very* unhappy ending.

In short, our choice of using COTS products in our systems expands and cements the notion that requirements need to share the driver’s seat. Since we choose to accrue the many benefits that come from letting the vendors develop COTS products, we must

therefore be willing to permit some parts of our systems—the “requirements” that drove the creation of those COTS pieces—to be controlled by those same vendors. And the law of transitivity holds here as well: if vendors have control over their own products, then some degree of the control over our fielded system is therefore in their hands, not in ours.

## **Last thoughts**

In a rather contorted way, I guess I am claiming that the way the word “requirement,” as used in COTS system building and especially in the notion of “requirements tradeoff,” is historically justified. By viewing “requirements” as a set of wants and desires, all of which reside on a sliding scale from absolute down through desirable to simple “nice-to-haves,” we are, in reality, returning to the original meaning of the word. (And perhaps we should think that those who use the phrase “requirements tradeoff” are actually linguistic Puritans at heart, and closet Chaucer scholars to boot.)

Maybe so, maybe not. But it is certainly true that for a COTS-based system, the more flexible meaning of “requirement” is the only one that makes sense. As we specify our systems today, we create some collection of system features that is really a bunch of things we want, but that includes a lot of things that we know we won’t get. And the act of “trading off” involves negotiation, giving and taking, weighing the risks we take if we give up certain capabilities, and similar unfamiliar but critical activities.

One last thought: During this brief consideration of how requirements figure in COTS-based systems, one thing has been a subtle and constant presence, and warrants further discussion. That is, in pursuing a COTS-based acquisition strategy and a COTS-based development paradigm, almost all of the new or novel things that we now must do (requirements tradeoffs being just one of them) somehow revolve around our loss of control over certain aspects of our systems. This loss of control is manifest in functionality, in schedule, in dependencies, and in other less obvious ways. It is a very interesting and vexing issue, and will be the topic of my next column. Stay tuned.

## **About the author**

David Carney is a member of the technical staff in the Dynamic Systems Program at the SEI. Before coming to the SEI, he was on the staff of the Institute for Defense Analysis in Alexandria, Va., where he worked with the Software Technology for Adaptable, Reliable Systems program and with the NATO Special Working Group on Ada Programming Support Environment. Before that, he was employed at Intermetrics, Inc., where he worked on the Ada Integrated Environment project.

---

The views expressed in this article are the author's only and do not represent directly or imply any official position or view of the Software Engineering Institute or Carnegie Mellon University. This article is intended to stimulate further discussion about this topic.

The Software Engineering Institute (SEI) is a federally funded research and development center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon University.

<sup>SM</sup> IDEAL, Interim Profile, Personal Software Process, PSP, SCE, Team Software Process, and TSP are service marks of Carnegie Mellon University.

® Capability Maturity Model, Capability Maturity Modeling, CERT Coordination Center, CERT, and CMM are registered in the U.S. Patent and Trademark Office.