Requirements that Handle IKIWISI, COTS, and Rapid Change

Barry Boehm, University of Southern California

In the good old days, dealing with software requirements was relatively easy. Software requirements were the first order of business and took place before design, cost estimation, planning, or programming. Of course, it wasn't simple. Certain straightforward criteria required satisfaction:

- completeness (no missing elements and each element fully described);
- consistency (no mismatches among the elements);
- traceability (back to the requirements for the system); and
- testability (specific enough to serve as the basis for a pass/fail test for the end product).

And, while avoiding design decisions about how all of this should be accomplished, you had to ensure that the requirements detailed the software's capabilities and any quality levels it must satisfy.

Completing these tasks might take time, but it was the only way you could guarantee that the delivered software conformed to original specifications. And, if you were contracting for software development, it was the only airtight way to ensure that competitive bidders were bidding to produce the same product. It was also the only basis for negotiating an airtight contract to ensure that the winning bidder would deliver what he promised. As the project proceeded, a requirement would occasionally need changing, but some relatively straightforward change control procedures could usually handle that.

Requirements help you deal with "I'll know it when I see it" users, off-the-shelf components, and rapid change.

SOME COMPLICATING FACTORS: IKIWISI, COTS, AND RAPID CHANGE

The recent developments of IKIWISI (I'll know it when I see it), COTS (commercial-off-the-shelf) software, and the increasingly rapid change in information technology have combined to unsettle the foundations of the old airtight-requirements approach.

IKIWISI

Successfully specifying software requirements in advance is difficult. But when user- or group-interactive systems are involved, it proves nearly impossible. Users asked to specify requirements generally claim, "I don't know how to tell you, but I'll know it when I see it."

Furthermore, users may initially feel that they "know it" when they see an initial demo or prototype. But their needs and desires change once they begin operating the system and gain a deeper understanding of how it could support their mission. Thus, the requirements tend to emerge with continued use and mission understanding rather than be prespecifiable.

COTS

Another fundamental tenet of the airtight-requirements approach is that the prespecified requirements completely determine the system capabilities. However, with large, pervasive COTS products, the COTS capabilities effectively determine the requirements.

For example, suppose you specify a one-second response-time requirement for a large transaction-processing system, and the best available COTS database management system can only handle your workload with two-second response time. Are you going to build your own version of Oracle or Sybase and hope you can make it twice as fast? Hopefully, in this and similar situations, you'll recognize that it's not a requirement if you can't afford it. Let the best available COTS capabilities determine your response time or other requirements—the reverse of the airtight-requirements approach.

Rapid change

As I discussed, specifying airtight requirements takes time. But particularly for Internet and Web-based systems, rapid change can create an impossible-to-win game of catch-up. As you slowly grind out and validate airtight requirements, rapid changes in COTS releases, competitive threats, stakeholders, reorganizations, and price structures make these requirements increasingly obsolete. And by the time you thoroughly change-control, update, and

revalidate them, new developments make them obsolete all over again.

THE REQUIREMENTS QUANDARY

Should you then forgo requirements completely? No—that would be like a convoy trying to navigate a complex route without a road map. With so many decision branches, independent project stakeholders can easily head in wrong or incompatible directions. Customers and users still need a description of the product destination assuring them that what developers produce will do what they need. This means that developers need some guiding principles—rather than inflexible rules—to help each project determine the best combination of requirements rigor and flexibility for each technical and organizational situation and environment. This situation-dependence implies an overarching primary principle: *Avoid one-size-fits-all approaches to requirements*.

The most significant problem with the airtight-requirements approach was its attempt to be a one-size-fits-all solution. Although this tactic fails for IKIWISI, COTS, and rapid change situations, the airtight-requirements approach proves valuable in many situations. Some examples are software-intensive safety-critical autonomous control systems for relatively stable products, such as heart pacemakers, automotive steering systems, and nuclear power plants.

SURMOUNTING THE REQUIREMENTS QUANDARY: FOUR GUIDING PRINCIPLES

Here are four guiding principles that help tailor a requirements strategy to fit your situation. These ensure that your requirements are value-, shared-vision-, change-, and risk-driven.

Value-driven requirements

Determining your most important requirements requires a business-case analysis. This shows the value added from various combinations of requirements relative to the investment necessary to achieve them. Analyzing the business case also forces you to focus on determining your system's operational concept.

This concept should use scenarios to identify how your new system will operate and add value over your existing system. It should also identify the success-critical stakeholders involved in transitioning to the new system, and how the new system will add value for each stakeholder. The operational scenarios are also useful for performing the business-case analysis; they provide specific situations around which to quantify costs and benefits.

Forgoing requirements completely would be like trying to navigate a complex route without a road map.

A particularly important consideration for new product introduction is the time-dependent value of the product sequence. Arriving first to market with only a partial capability is often preferable to arriving later with a complete set of requirements. In some cases, such as demonstrations at major trade shows, the value may decrease dramatically if it's not available on time. In such cases, using a schedule-as-independent-variable (SAIV) rather than a complete-requirements-as-independent-variable process model is preferable. A SAIV approach involves prioritizing the requirements, defining a core capability that is easily deliverable on time, and architecting the system so that lower-priority requirements can be dropped if they threaten on-time delivery.

Note that a value-driven requirements approach implies concurrent development of the requirements, the architecture, and the development plans. For cost-benefit and return-on-investment analysis, the requirements determine the benefits, but the architecture and plans determine the cost and schedule.

Shared-vision-driven requirements

Rapid changes in the problem situation (market competition), solution situation (new technology), and value situation (price structures or organizational realignments) imply that the system's stakeholders frequently need to reassess and revise the system and software requirements. This means that it is more important to emerge from the initial requirements definition process with a shared vision of the system's goals and values than with a precisely defined requirements spec. The shared vision helps all the stakeholders quickly adapt to the new situation, while the precise spec takes more effort to change and some stakeholders may not understand it. Beyond a shared vision, however, stakeholders must emerge with a set of shared commitments to realizing the vision and its shared values.

Too often, software requirements operate under a field-of-dreams assumption: "Build the software to the requirements, and the benefits will come." John Thorp cites extensive evidence that there is little correlation between a company's level of information technology investments and its profitability or market value (*The Information Paradox*, McGraw-Hill, New York, 1998). His book provides convincing evidence that the field-of-dreams approach is responsible for many of the lost opportunities leading to the *information paradox*—the disconnect between IT spending and business benefit.

The Thorp book also provides the best approach I've seen for avoiding this problem: the DMR Benefits Realization Approach, represented by the results chain in Figure 1. It establishes a framework that links initiatives, which consume resources (for instance, implementing a new order entry system for sales), to contributions (not just the delivered system, but rather its effect on existing operations) and outcomes. Outcomes can lead to further contributions or to added value (like increased sales). A particularly important contribution of the results chain is the link to assumptions, which are conditions necessary to the realization of outcomes. For example, in Figure 1, if order-to-delivery time is not an important buying criterion for customers, the reduced delivery time will not result in increased sales.

This framework is also good for identifying non-software stakeholder initiatives (for instance, training, public relations, and order fulfillment speed-

ups), which are also necessary conditions to realizing benefits. It also provides a way to track progress on all the necessary initiatives and their effects on contributions and outcomes. Tracking realized benefits permits businesses to apply necessary corrective actions if benefits don't materialize (including changes to the software requirements).

Change-driven requirements

Another major problem with current software requirements practice and guidelines is that they only capture a snapshot of the requirements from any given moment. Particularly in competitive bidding, such snapshot requirements specs lead to point solution architectures: The winning bidder can satisfy the stated requirements at lowest cost, but the software will be expensive to adapt to later requirements changes.

More than 20 years ago, David Parnas's paper, "Designing Software for Ease of Extension and Contraction" (*IEEE Trans. Software Eng.*, Mar. 1979, pp. 128-137), provided an elegant solution to this problem. It involves identifying the most likely directions of change in the requirements (for instance, new workstations on which to operate), and encapsulating these sources of change in the design via Parnas's information-hiding techniques. For example, you could hide workstation details in a workstation-handler module. Then, when the changes come, they only affect a single module rather than the entire software product.

Although information hiding has become a widely adopted design technique, it is surprising how rarely developers practice its counterpart: specifying and using *evolution requirements* to achieve a change-driven design. Most of today's standards for specifying requirements still do not have a section on evolution requirements.

Another problem with current requirements and design specs is that they capture only the surviving decisions, and not the rationale by which other alternatives failed. This often leads to misguided change adaptation. For example, a programmer or subcontractor may reuse a module to save time, even though the module reuse failed to pass the prime contract's requirements definition because of portability or maintainability weaknesses.

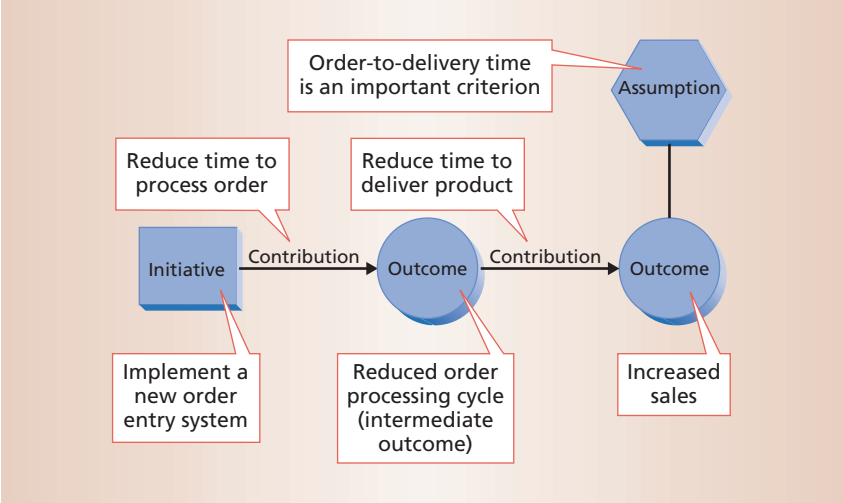

Figure 1. Benefits Realization Approach results chain.

Several techniques are emerging for capturing such rationale and facilitating adaptation to change. These include the results chain described earlier, the stakeholder win-win negotiation results captured in the MBASE (Model-Based System Architecting and Software Engineering) Feasibility Rationale, and scenario-based rationale capture.

Risk-driven requirements

Once they've done value-, shared-vision-, and change-driven requirements, developers still face the question, "How much requirements specification detail is enough?" As with similar questions regarding planning, prototyping, testing, and change control, the best answer I've found is to take a risk-driven approach. This basically says, "If it's risky to leave it out, put it in. If it's risky to put it in, leave it out."

Thus, for example, anything less than thoroughly specifying interface requirements between the software and a specialized hardware device, or between the software in two large, integrated systems, is risky. If these interfaces are ambiguous or undefined, there will be major risks of interface mismatches causing either serious operational problems or massive rework and delays during integration. If the designs on either side of the interfaces are still getting sorted out, it's best to spawn one or more risk-driven spiral cycles to suitably define the interfaces.

On the other hand, it's very risky to prespecify the exact layout of a GUI in stiff prose. There's too much risk of an IKIWISI phenomenon making the specs rapidly obsolete. And, with a GUI-builder package, there's little risk involved in changing the layout as new insights emerge. Thus, in this case, it's better to agree to a prototype as the initial GUI requirements definition, and to agree to evolve the GUI within the technical capabilities of the GUI-builder package.

Several organizations have successfully developed approaches that work well in coping with IKIWISI, COTS, and rapid change. Some particularly good examples are e-commerce solution builders such as C-bridge and the IBM and Oracle e-commerce divisions. The DMR Group has successfully applied its Benefits Realization Approach across a wide variety of application areas, as has Rational Inc. with its Rational Unified Process.

In over 100 requirements definitions of Web-based rapid-development applications for University of Southern California and Columbia University clients using the MBASE approach, my col-

leagues and I find that the value-, shared-vision-, change-, and risk-driven approaches to system and software requirements definition are mutually reinforcing. Using a stakeholder win-win requirements negotiation approach, business-case analysis, and a benefits-realization approach all help stakeholders prioritize their requirements and capture the rationale for their decisions. The lower-priority requirements become evolution requirements, providing the basis for architecting the system to easily drop them (if necessary to meet schedule) or incorporate them in later increments. And the risk analyses developed for the risk-driven spiral process help determine how much is enough for the requirements specs. Thus, there are good prospects for a mutually reinforcing set of requirements practices, providing a stronger sense of security than was previously achievable with airtight requirements.

Barry Boehm is director of the University of Southern California's Center for Software Engineering. Contact him at boehm@sunset.usc.edu.

Editor: Barry Boehm, Computer Science Department, University of Southern California, Los Angeles, CA 90089; boehm@sunset.usc.edu

Sources and Resources

For value-driven requirements, the best source is Thorp's *The Information Paradox* (McGraw-Hill, New York, 1998). Besides Parnas' paper, the best source for change-driven requirements is James Highsmith's *Adaptive Software Development* (Dorset House, 2000). For shared-vision-driven requirements, Donald Gause and Gerald Weinberg's *Exploring Requirements: Quality Before Design* (Dorset House, 1989) is excellent. Suzanne and James Robertson's *Mastering the Requirements Process* (Addison Wesley, 1999) and Michael Jackson's *Software Requirements and Specifications* (Addison Wesley, 1995) both complement the Gause and Weinberg book. For risk-driven requirements, the USC MBASE approach (http://sunset.usc.edu/MBASE) has the most specific coverage. Some good unified approaches of all four principles are MBASE, Thorp's Benefits Realization Approach, and *The Rational Unified Process* (Phillippe Kruchten, Addison Wesley, 1999).