COVER FEATURE

The Art of Requirements Triage

Triage is the process of determining which requirements a product should satisfy given the time and resources available. The author presents three product development case studies and 14 recommendations for practicing this neglected art.

Alan M. Davis
University of Colorado at Colorado Springs

In meeting rooms of development organizations worldwide, a familiar scene plays out: Marketing personnel say, "The new release must provide feature X in eight months or we might as well not even build it." System developers respond, "If we build feature X into the next release, we'll need 12 months to deliver it." This exchange has two possible outcomes: One, developers bend under the pressure and agree to an impossible schedule. The product is delivered late, and the company fails to meet its forecasts. Two, marketing personnel bend under the pressure and agree to a less-than-satisfactory product. The product is delivered on time but to an uninterested market, and the company again fails to meet its forecasts.

More and more companies are feeling this requirements-versus-resources tension. Driven by an increasingly competitive market, they attempt to add features and compress schedules for the delivery of every product. The result is often a complete mismatch of requirements and resources, resulting in products that fail to satisfy customer needs.

Medical personnel must deal with similar considerations when treating victims of a disaster—a practice dubbed triage. They systematically categorize victims into three groups: those who will die whether treated or not, those who will resume normal lives whether treated or not, and those for whom medical treatment may make a significant difference. Each group requires a different strategy. The first group receives palliative care, the second group waits for treatment, and the third requires some ranking in light of available resources. As new victims appear, personnel must repeat the categorization.

Determining what requirements a product will satisfy follows a similar triage process. In the first group are requirements that the next baseline clearly must satisfy. In the second are requirements that the next baseline clearly need *not* satisfy. In the third group are requirements that the product could incorporate, but that the development team must first carefully weigh against available resources. Again, each group requires a different strategy: Put the candidate features in the product, put them in a bin for consideration in the next release, or group them by priority on the basis of available resources. As new requirements appear, new priorities are established, and the team revisits the categories.

Triage—the process of determining which requirements a product should satisfy given the time and resources available—comprises three main activities:

- Establish relative priorities for requirements. This may include establishing priority-related interdependencies as well.
- Estimate the resources needed to satisfy each requirement. This may include establishing resource-related interdependencies as well.
- Select a subset of requirements that optimizes the probability of the product's success in its intended market, whether commercial or internal, relative to the resource constraints.

The practice of triage increases the likelihood that products will meet customers' needs, and thus contributes significantly to the economic impact of that product on the company's bottom line. Yet despite

Published by the IEEE Computer Society 0018-9162/03/$17.00 © 2003 IEEE

the potential benefits of requirements triage, not much has been written about it, and the descriptions that do appear are brief.[1-4] Doubtless, this is because triage is a difficult art, fraught with political and financial dangers—politically dangerous because both technical and marketing personnel claim the tasks as part of their responsibility; financially dangerous because a mistake could trigger a major loss of revenue.

I have studied the requirements practices of approximately 100 companies and organizations over 25 years and found that few incorporate triage in their requirements practices. In this article, I present three case studies* extracted from those experiences and offer recommendations that will aid practitioners in aligning requirements and available resources.

CASE STUDY 1: MARK VERSUS DEV

In winter 2000, a product manager at a large manufacturer of mass storage devices contacted me to serve as a consultant to resolve a problem. Mark, the marketing manager, was demanding delivery of version 3.0 of product Y to customers in nine months. Meanwhile, Dev, the product development manager, was insisting that delivery in nine months was impossible. Mark and Dev had reached an impasse. My first task was to ask each of them to describe his position.

Mark said, "Look, the window of opportunity starts in nine months. We know that the competition is planning to release similar products 10 to 12 months from now. Since their products and our 3.0 release are so similar, the only way we can be successful is to be the first to market."

Dev responded, "I understand what you are saying. But just wishing for something is not going to make it happen. My team cannot produce all the features you want in release 3.0 in nine months. It simply cannot happen."

Making an appeal to Dev's corporate allegiance, Mark asked, "Don't you realize that you'll be letting down the entire company if you don't build it when it is needed?"

Dev counterattacked, "Look, Mark, a year ago when we were planning the 2.4 release, you demanded that I deliver it in five months. I told you then that we had two choices: Build it in five months with an architecture that could not support additional features or build it in eight months with an architecture redesigned to handle many additional requirements. You chose the first option, so it's your fault that we're in this mess now! The current architecture simply cannot support the 3.0 features. I need a full year to revamp it and then add the new features."

An hour later, they were no closer to a mutually agreeable solution, so I presented a schedule probability graph to the team. The curve in Figure 1 made it clear that the team had only a 32 percent likelihood of delivering the project in nine months—a level of risk that neither product development nor marketing was willing to accept. In other words, Dev wasn't lying.

During a short break, I asked Dev to consider what he would do if he owned a majority of the company stock, and the project's success or failure in the marketplace was key to his personal financial success. Clearly, promising to deliver something in nine months when it would actually take 12 would not be productive. Delivering it in a year to a stale market also was not an option.

Dev responded by saying, "I would increase my head count so that I could staff two parallel product development teams. One team would work on incorporating as many of the 3.0 features as possible into the old architecture. We could call it version 2.5 and release it in around seven to eight months. Meanwhile, the other team would start in parallel to revamp the architecture and be ready to deliver the full 3.0 capability in about a year."

After we had reconvened and Dev made his suggestion, Mark responded with "Wow! Would you really do that for me?" We had finally reached a

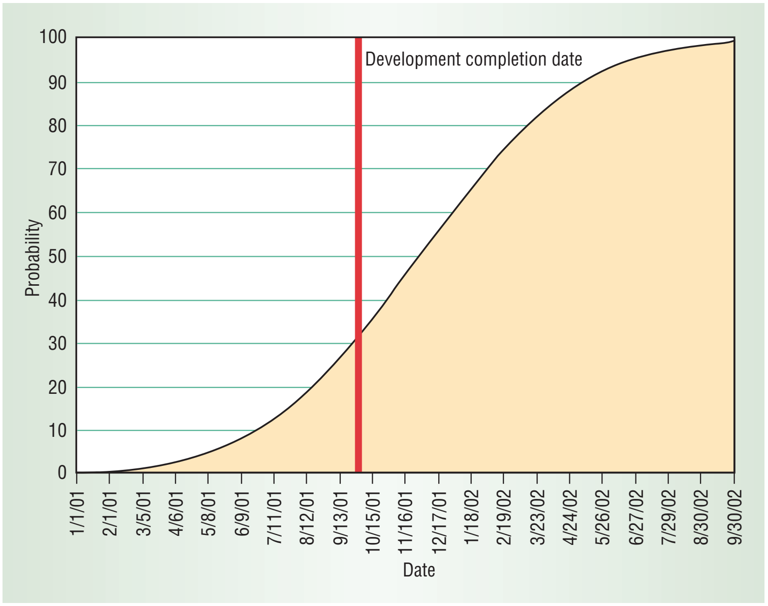

Figure 1. Probability of completing release 3.0 on schedule. The red vertical line represents the desired nine-month time to market. With this completion time, the point where the red line crosses the curve shows that the project's likelihood of completing on schedule is only 32 percent, an unacceptable level of risk.

*I have extracted parts of these case studies from my forthcoming book, *Just Enough Requirements Management* (Prentice Hall, 2003). The first and third studies are completely true as described; only the names are fictitious. The second case study is a compilation of several stories; each event described is true, but the events did not occur in the sequence given, nor did they occur at the same company. Case study 3 was described in detail in A. Davis, "Can You Survive Your Management Mistakes?" *IEEE Software,* July 1992, pp. 72, 85.

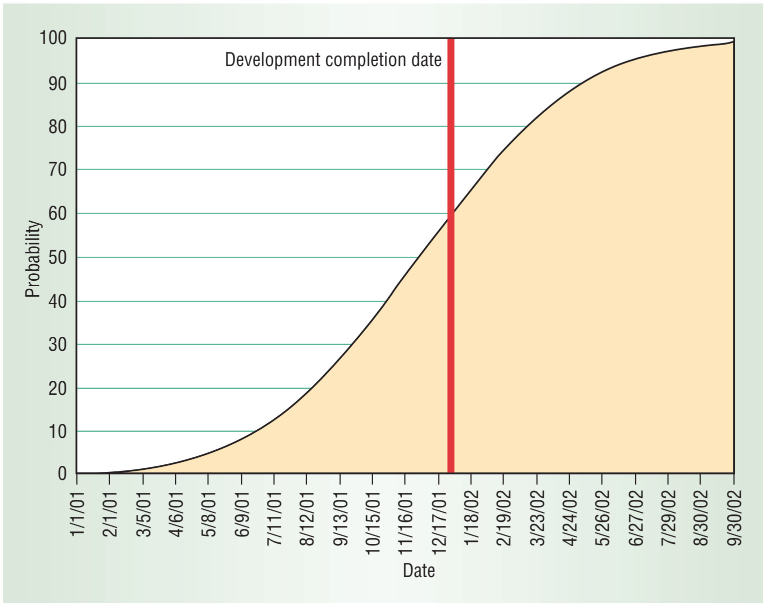

Figure 2. Probability of completing release 3.0 in 12 months. The probability of success jumps to 58 percent, an acceptable risk level.

tentative agreement in principle. We worked a few more hours to see if it was realistic.

First, we checked to verify that the 12-month schedule for the 3.0 release was reasonable. Moving the vertical line in Figure 1 to the right by three months resulted in the graph in Figure 2, showing that the likelihood of success jumped to 58 percent—an acceptable risk level. But the real test came next. To determine whether release 2.5 was possible, we let the development team select the subset of requirements they could include with the old architecture. Figure 3 shows the resulting probability graph, which also indicates an acceptable risk level.

At first, Mark was reluctant to sign up for the proposed strategy; the requirements that Dev had suggested did not make for a particularly impressive product. Then Mark had a great idea. The proposed release 2.5 was not as good as the products the competitors were to release in 10 to 12 months, but it was better than anything currently on the market. Mark's plan was to offer version 2.5 to customers at a very low price, not even enough to recover the company's R&D and manufacturing costs. In so doing, the company could seriously dampen the market demand for the competing products that would come out a few months later. When the company released version 3.0, the competitors' products would not have already captured the market.

CASE STUDY 2: SUPER-REQUIREMENTS

In spring 2000, I was consulting for an e-business solution provider that was struggling over what features to include in its next major product. This company was unusual in that the vice presidents of product development and marketing seemed to respect each other. The development VP trusted the marketing VP's assessment of the changing market dynamics. The marketing VP trusted the estimates that the development VP provided. Both VPs realized that their employees needed to work together to solve this marketing puzzle instead of attacking each other, as in most companies.

The company had already had numerous brainstorming sessions with their existing and potential customers, which resulted in around 200 candidate requirements. To satisfy all those requirements would take around 18 months, but the customers who were depending on their foundation solution needed it in just six months.

My first recommendation was to begin with fewer requirements. It would simply take too long to perform triage on all 200. I suggested that they group the requirements into small sets—features[2] or super-requirements—that they could sell as a package to aid a customer in satisfying one or more business goals. The requirements within each set also had to be closely related from an implementation perspective. That is, the developers could easily visualize how to implement them as a package. After a two-hour discussion, we arrived at a list of some 30 super-requirements.

We debated a variety of business strategies and finally converged on the concept of building a series of very small increments, while keeping customers informed of all our plans. Our goal was to develop a group of extremely loyal product users who would always know the date of the next release, the requirements the next release would satisfy, and so on. We invited select customers to a meeting at the corporate offices to explain our strategy and what we needed from them.

It took a full day of discussion with the customers to arrive at a schedule of 10-week releases and their associated super-requirements. (The first release would take a bit longer because we had to build much of the supporting infrastructure.) We analyzed the first five releases separately using graphs similar to Figures 1 through 3 to ensure that they were feasible. The customers were excited. Marketing was thrilled. Product development was confident that they could deliver each release on time.

The first release of the product was created without a hitch, within budget, and with no surprises, and customers were satisfied with its functions.

Unfortunately, soon after the first release, new customers emerged, market needs changed, and cooperation started to ebb within the company. As a result of the earlier meetings, product development was making many technical decisions that were based on the agreed-to releases and associated super-requirements. When needs changed, the esprit de corps began to erode. Rather than rethink prior decisions, they viewed changing needs as a threat to the success of the entire project.

CASE STUDY 3: CAPITULATION BASED ON FEAR

I was a director at an R&D laboratory for a large telecommunications company in the late 1970s. One of my project managers was responsible for building a software system for an operating division of the same company. The division had contacted us about doing the job because we had the technical expertise to create the unique system.

The project manager carefully analyzed the job and decided that he and his team could complete it in about 12 months. However, the customer wanted to have the system in nine months. When I discussed our alternatives with the project manager, the first idea was to dig in our heels. We knew that 12 months was already optimistic. We would simply tell the customer that it couldn't be done. The second idea was to revise our schedule and agree to the nine-month delivery. After all, it was not impossible (just extremely improbable) that we could complete the product in nine months.

We role-played the customer in each scenario. The first scenario did not bode well for us. We could visualize digging in our heels and hearing the customer respond, "Fine, I'll go somewhere else to get my product." In that case, I would have had to fire the project manager and his entire team because we had no other productive work for them to do. Consequently, fearing that we would lose our jobs, face, or opportunity, we agreed to reduce the schedule to nine months.

To no surprise, we failed to make the delivery. Unlike many similar situations, we could not even blame the customer for changing the requirements during the development process because the requirements did not change. Both the project manager and I ended up eating crow. However, a few months after the originally scheduled delivery date, the customer's priorities changed, and he determined that he would not need our product for some time to come. We eventually delivered the product in around 13 months from the start date.

HOW TO PERFORM TRIAGE

These case studies give a glimpse of what can happen with a complex combination of personalities, anxieties, changing markets, unknown requirements, poor planning, and inadequate analysis and comparison tools. Practitioners need concrete strategies to deal with these complexities and intelligently analyze candidate requirements. On the basis of these case studies and my other experiences, I offer 14 key recommendations.

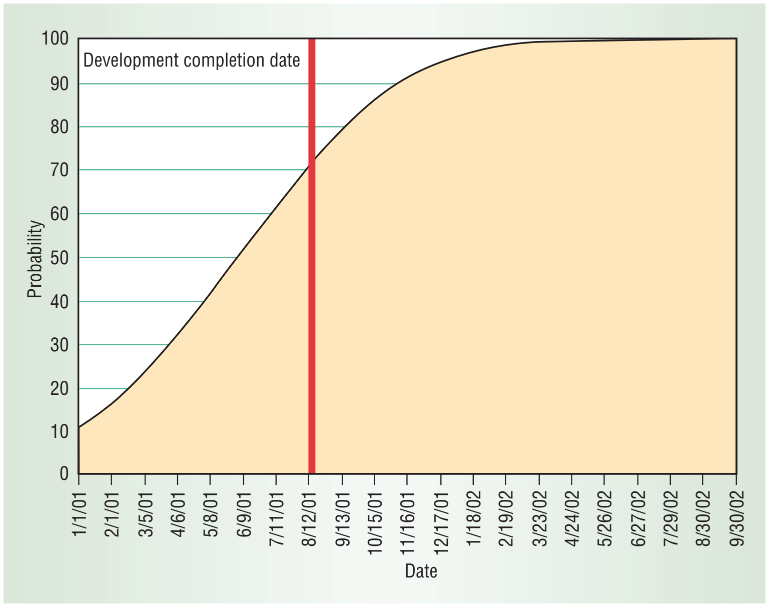

Figure 3. Probability of completing release 2.5, a compromise that includes a subset of the requirements for release 3.0. Again, this is an acceptable risk level.

Maintain a list of requirements

When you gather candidate requirements as part of elicitation, maintain them as a list—in a spreadsheet, database, requirements management tool, or even a bulleted list in a word processor. You can then more easily answer questions like, "How many requirements do we have?" and "What percentage of the requirements are we planning to implement in the next release?"

In most cases, it helps to maintain a hierarchical list. Many hierarchies are possible, but the containment relationship is most often used as the basis of a requirements hierarchy. If three requirements are each part of (or represent a refinement of) requirement A, they become A.1, A.2, and A.3 and appear as children of requirement A in the hierarchical list. This practice makes it clear which requirements subsume other requirements and helps prevent meaningless arguments about incorporating requirement A or requirement A.2.

In case study 1, the team maintained a hierarchical list. In case study 2, the team maintained super-requirements as a simple (nonhierarchical) list for triage activities. However, behind the scenes, they kept a record of which original requirements were subsumed by each of the super-requirements. In both cases, the lists served as the basis for intelligent discussions about which requirements to include and which to defer. In case study 3, the team maintained requirements in a word-processing file including requirements, other sentences, paragraphs, chapters, and so on, which made it impossible to discuss any subset of the requirements and thus to reach a compromise about what to include.

After listing the requirements, the next step is to determine the degree of effort associated with satisfying each requirement.

Record necessity interdependencies

Often a requirement makes sense only when one or more other requirements are also satisfied. For example, the requirements

- The system shall provide the Fiends and Famine capability for our customers.
- The system shall bill customers \$3 per minute when they use the Fiends and Famine feature.

have a necessity interdependency: There is no value in providing either capability without the other.

Requirements can also have a one-way necessity dependency. For example, in the requirements

- The system shall provide a stop button in the upper right corner.
- The stop button shall be red.

the first requirement makes sense without the second, but the reverse is not true.

If you maintain requirements in a list, you can simply maintain a column (field, attribute) for storing the necessity dependencies.

None of the teams in the three case studies recorded necessity dependencies. In case studies 1 and 2, the teams maintained separate lists of viable subsets of requirements that made sense from a customer perspective. This practice worked well in both cases for product development, but failure to explicitly maintain the interdependencies in the lists could have had implications in later maintenance activities.

Annotate requirements by effort

Once the requirements are in a list, developers should determine the degree of effort associated with satisfying each requirement. Annotation helps them do triage intelligently.

To decide whether to include or exclude a requirement, you need to know approximately how much effort it takes.The units of such calibration are unimportant. Some companies use function points or feature points; others use person-hours or lines of code. For triage, any measure works as long as you're consistent.

You must take care to define the effort (*e*) associated with requirements when they are organized hierarchically. If requirements A.1, A.2, and A.3 are parts of requirement A, then you must maintain the property

$$e(\text{A}.1) + e(\text{A}.2) + e(\text{A}.3) \leq e(\text{A}),$$

which becomes an equality if A.1, A.2, and A.3 cover all the functionality that A implies.

Effort dependencies also can exist between requirements that do not share a hierarchical relationship. For example, requirement B may require three person-weeks of effort to satisfy by itself, and requirement C may require five person-weeks, but once requirement C is satisfied, satisfying requirement B takes only two additional person-weeks. Here you would record the effort dependency in yet another column (or field or attribute) of the list or database. In this case, the value for that field for requirement B would contain an estimate of the corresponding effort savings—namely, one person-week.

In case studies 1 and 2, the teams annotated requirements according to the estimated effort required to satisfy them, using person-weeks as the measure. This practice aided intelligent discussions of which requirements to include and which to defer. In case study 3, the team only estimated the effort to complete the project. Consequently, they had no flexibility for negotiation; the choices were either to build or not build the entire product.

Neither of the case study 1 and 2 teams actually recorded effort dependencies. Instead they kept this knowledge in their heads and expressed it as needed during triage. Although this practice worked fine in these cases, it might not work as well with longer lists of requirements, constantly shifting requirements, or in an environment with high employee turnover.

Annotate requirements by relative importance

While developers are determining the degree of effort needed to satisfy the requirements, other stakeholders—customers, marketing representatives, or consultants—should rank each requirement's importance. Stakeholders' input is critical, but if you ask them individually to categorize requirements, you're likely to hear, "We need them all. That's why we listed them in the first place." Instead, gather the stakeholders in one location and follow a group voting mechanism, such as the hundred-dollar test.[2] Using this test, stakeholders show their preferences by distributing an imaginary one hundred dollars among the requirements; the more dollars allotted to a requirement, the stronger the interest.

Another alternative is to point to each requirement and ask for a show of fingers to indicate stakeholders' enthusiasm for including it in the next release:

- One finger up says, "You should include that requirement."

- Two fingers up says, "If you omit that requirement, the product won't be at all useful."
- One finger down says, "You should exclude that requirement."
- Two fingers down says, "If you include that requirement, the product won't be at all useful."
- No fingers up or down says, "I am neutral about including or excluding that requirement."

Rarely will stakeholders vote the same way for every requirement because their votes would then have no effect on the overall decision.

Be careful when defining priorities (p) associated with requirements if they are organized hierarchically. For requirement A and its subrequirements A.1, A.2, and A.3, for example, you must maintain the property

$$\max(p(\text{A.1}), p(\text{A.2}), p(\text{A.3})) \leq p(\text{A}),$$

which becomes an equality if A.1, A.2, and A.3 cover all the functionality that A implies.

In case study 1, the team did not maintain relative priorities of candidate requirements. During triage, stakeholders simply voiced their opinions about the importance of requirements. Consequently, in the initial triage session, the customer representative (marketing) consistently demanded the inclusion of almost every requirement. In case study 2, the team collected and maintained relative priorities, which made the initial triage session much easier. In case study 3, no one had even enumerated requirements, much less annotated them.

Do triage overtly

Individuals can perform many product development activities in isolation. But to be successful, triage requires input from at least three constituencies: customers, developers, and financial representatives. Triage becomes falsely trivial if any one of these constituents is missing. Without customers, financial representatives and developers could decide to build all the easy requirements. Without developers, customers and financial representatives could decide that effort estimations are exaggerated and should be halved. Without financial representatives, customers and developers could decide to add more resources.

In case studies 1 and 2, the teams performed triage in open meetings. However, in case study 1, the absence of the financial representatives led to the subsequent disintegration of the compromise reached in the meetings. When they learned about the decision, the financial representatives refused to support the increase in staffing for the development organization, and the entire project had to return to the drawing board. In case study 2, because the financial representatives were present, the developers proceeded through the first release with few complications. In case study 3, the customer was also the financial representative. This position of power, although often unavoidable, makes negotiation more difficult.

To be successful, triage requires input from at least three constituencies: customers, developers, and financial representatives.

Base decisions on more than mechanics

Dietrich Dörner writes that when managers use purely mechanical means to arrive at a decision, they have little stake in the outcome.[5] If the decision proves wrong, no ego is involved, and they can simply declare that the mechanism failed. When managers use tools to aid in making a decision rather than allowing the tool to make the decision, they are more apt to work toward a successful outcome.

Thus, when using tools such as the cumulative probability graphs in Figures 1 through 3, be sure that the solution makes intuitive sense. You can then make a triage decision because all stakeholders agree that it makes sense, not solely because the graph says a situation has a 75 percent likelihood of success.

Establish a teamwork mentality

In the more than 75 companies I have consulted for, I have inevitably found enormous energy channeled into adversarial relationships. Although this type of tension can be positive, more often it leads to months of infighting that stymies progress in building the product. The schedule becomes moot when the team argues for six months about whether the product needs to be delivered in six or nine months.

Case study 2 illustrates the positive benefits of teamwork. Early in the process, everyone adopted the philosophy that their company had a group challenge of getting the best product to market as soon as possible. They adopted a "we versus them" rather than a "we versus we" attitude. A positive attitude focuses the team on finding a solution that assists all parties in winning;[6] an adversarial attitude merely blurs the goals.

Manage by probabilities of completion, not by absolutes

A typical exchange between marketing personnel and product developers exemplifies thinking in absolutes. When attempting to understand customer needs and determine whether a product can be built within a given time, *nothing* is absolute. The outcome can be anything from highly unlikely to highly likely. Once everyone recognizes this, negotiation becomes much easier.

When attempting to understand customer needs and determine whether a product can be built within a given time, *nothing* is absolute.

Instead of speaking in absolutes, make observations based on mechanisms like cumulative probability graphs. These graphs are based on data that is easy to collect, and they help prevent adversarial absolute attitudes about the project. Stakeholders can then debate, think through the problem, and arrive at a compromise based on constructive observations such as: "If we include feature X in the next release, the likelihood of delivering the release in June drops from 40 percent to 3 percent."

Collecting the data to support such graphs is extremely easy. I generated Figure 1, for example, from a table of past projects, with two data elements per project: the original estimate of effort, using the same metric used for estimating requirements effort, and the actual project duration. Once you have this data, creating the graphs is trivial. To answer the question, "What is the probability of completing X work units (by any measure) in six months," just look at the data to see what percentage of projects that projected X or fewer work units managed to complete within six months.

Some may argue that such data is invalid: The requirements change on a project, making the original estimate incomparable to the actual completion schedule. My counterargument is that if requirements changed, say, by 50 percent on all past projects, they are likely to change about 50 percent on the current project. Thus, we really are comparing apples and apples.

The teams in case studies 1 and 2 followed this recommendation and were able to reach decisions. In case study 3, no compromises were possible because all positions were stated in absolutes.

Understand the optimistic, pessimistic, and realistic approaches

The optimistic approach to triage is to assume you can address all the requirements. If the likelihood of success is too low, you remove requirements one at a time until the risk is acceptable. In all three case studies, the team followed this approach. It is popular because, in most cases, every member of the team, regardless of affiliation, truly wants to see all the customer needs satisfied.

The pessimistic approach is to assume you can address none of the requirements. Your risk level is likely to be very low, so you add requirements one at a time until the risk is barely tolerable. This approach works well when teams are composed of curmudgeons who believe that little can be accomplished.

In the realistic approach, you assume that you start off with some "reasonable" subset of requirements, then you add or remove requirements until you reach an acceptable compromise.

All three strategies work, but it helps the team if they understand what they are doing and why.

Plan more than one release at a time

Instead of planning the requirements for just one product release, plan them for at least two. During triage, the team typically analyzes each requirement to determine if they should or should not include it in the next baseline. This becomes a binary decision, with no middle ground. Instead, when a deadlock arises about including a requirement, establish a consecutive release as an alternative. Suppose you are planning which requirements to address in release 2.5. Rather than forcing a binary decision—put them in release 2.5, or don't—plan releases 2.5 and 2.6 simultaneously. Now you have an intermediate position. If the stakeholders cannot agree about whether or not a requirement belongs in release 2.5, just allocate it to release 2.6 if doing so makes business sense.

This strategy was fundamental to the success of case study 1, in which the team planned two releases simultaneously and compromises were therefore possible. In case study 2, a long series of releases were planned, making compromises easy. Unfortunately, the company saw this series as carved in stone and did not allow for future flexibility.

Replan before every new release

Although I recommend planning at least two releases at a time, you must replan every release as soon as you complete the previous release. When the development efforts for two consecutive releases overlap, replanning for the second release must of course take place before you finish the previous release.

Replanning prevents using decisions made on the basis of yesterday's information to drive tomorrow's actions. Instead, you are always updating the plan for each new release on the basis of your knowledge about the current market.

Case study 2 shows what can happen if the team doesn't replan. As market conditions changed, the developers stubbornly clung to old agreements, which created turmoil when marketing wanted to change the strategy.

All team members need to understand that the world will change regularly, and that the agreements they make today are based on what they know now—made in the interest of moving ahead with the project. Tomorrow they could make new decisions based on what they know then.

Don't be intimidated into a solution

When product development is pressured into accepting a schedule that has an unacceptably low probability of success, every member of the team loses. Disaster often occurs when an organization staffs for a product release and plans its revenue forecast based on a false expectation of product release. In many situations, as in case study 3, the development organization is intimidated into compressing a realistic schedule, resulting in either poor quality or late delivery.

Intimidation can also force marketing into accepting a date that is technically feasible but presents a marketing nightmare, which has an equally unacceptable result—a product delivered late in the market window.

The antidote to intimidation is to seek an innovative combination of releases, pricing strategies, and marketing ideas, as in case study 1. The right tools are also important. In case study 3, we might not have been intimidated into a six-month schedule if we could have shown the customer some cumulative probability graphs. In this case, the graphs would have clearly shown that the six-month schedule would result in a lose-lose outcome.

Find a solution before you proceed

Too often, companies that can't find a compromise decide to just march ahead toward an impossible delivery date or a stale market, hoping that something will change. They are correct that something will change, but the changes are almost always for the worse.

Remember that perfection is impossible

Many companies spend all their energy pursuing the perfect product or the perfect product strategy. Ironically, the time spent trying to achieve perfection is the very thing that sabotages perfection. As Robert Meltzer noted, "It is better to be 90 percent right today than 95 percent right six months from now."[7]

Requirements triage is a multidisciplinary art that is critical to the success of any product development, yet companies rarely practice it. Not following this approach can result in years of infighting among project stakeholders, products that fail to satisfy customer needs, and considerable loss of revenue. On the flip side, practicing triage enables companies to foster cooperative, low-stress development environments, build products that meet their customers' needs, complete projects on schedule, and optimize their return on investment. ■

References

1. A. Davis and A. Zweig, "Editor's Corner: The Missing Piece of Software Development," *J. Systems and Software*, Sept. 2000, pp. 205-206.
2. D. Leffingwell and D. Widrig, *Managing Software Requirements*, Addison-Wesley, 2000.
3. K. Wiegers, *Software Requirements*, Microsoft Press, 1999.
4. E. Yourdon, *Death March*, Prentice Hall, 1997.
5. D. Dörner, *The Logic of Failure*, Addison-Wesley, 1996.
6. H. In, "Applying Win-Win to Quality Requirements: A Case Study," *Proc. Int'l Conf. Software Eng.*, IEEE CS Press, 2001, pp. 555-564.
7. R. Meltzer, "Accelerating New Product Development," *The PDMA Handbook of New Product Development*, M. Rosenau Jr., ed., John Wiley & Sons, 1996, pp. 345-359.

***Alan M. Davis** has more than 20 years' experience in industry. He is a professor of information systems at the University of Colorado at Colorado Springs and author of more than 100 papers and two books:* Software Requirements: Objects, Functions and States *(Prentice Hall, 1993) and* 201 Principles of Software Development *(McGraw-Hill, 1995). Davis received a PhD in computer science from the University of Illinois and has been a Fellow of the IEEE since 1994. Contact him at adavis@uccs.edu.*