The Art of Expectations Management

(Dedicated to the memory of Tom Bauer)

Barry Boehm, University of Southern California

One of the most valuable skills a software professional can develop, expectations management is something surprisingly few people know or practice. I've witnessed more than 100 stakeholder software requirements negotiations in which inflated expectations about the simplicity of the problem or ease of providing a solution have caused the most difficulty. Expectations management holds the key to providing win-win solutions to these situations.

When I ran TRW's office of Software Cost Estimation, my greatest asset was Tom Bauer. Tom programmed and operated TRW's version of the Constructive Cost Model (Cocomo), which the company used on all its projects and proposals. He had been a project and department manager, and had gone back to his first love, programming, after a heart attack. He had a lot of experience and wisdom, and a smile for everyone.

One day I went to Tom and said, "We need to add a new computer security cost driver to Cocomo for this new proposal. It looks like a straightforward addition to our algorithm, and they need it in a week. Can you fit it in?"

Tom said, "Well, let's see. Besides the change in the algorithm, we'll need to change our data structures, our file formats, our input forms and user interfaces, our output formats, and our help and error messages. Besides that, there's regression testing, the changes to the users' manual and model definition manual, and the Basis of Estimate feature I'm working on for Pricing. That looks more like six weeks to me.

"In the short term, though, if there's one of the other cost drivers they're not using, I can give them a special version of the program with that slot replaced by computer security in a couple of days. How's that for an interim solution?"

This proposal wasn't what I expected, but it looked reasonable, so I went along with it. Tom delivered the quick fix on schedule, and three weeks later Tom pleasantly surprised me by delivering the full new version in half the promised time.

But suppose Tom had initially said, "Sure, no problem," to my request for a one-week completion, and proceeded to deliver the solution in three weeks. This would have made me distinctly unhappy with Tom, and the new-proposal people would have been distinctly unhappy with me.

Instead, Tom had successfully managed my expectations, and had provided an acceptable short-term fix while doing so.

Clear communication, careful estimation, and precise planning can help you shape and meet realistic expectations.

WHAT'S THE PROBLEM?

In my experience, three main culprits cause expectations problems in software project development. First, developers often show a desire to please that's taken to the point of avoiding any confrontation. This behavior leads to many situations in which people will promise more than they can confidently deliver.

Second, in their desire to sell, developers sometimes become overenthusiastic about how well their methods, tools, or packaged products will solve clients' problems. Rosy sales literature about a product or unscalable demos of how well it works on a sample problem can be just as misleading as optimistic assumptions.

Third, developers and clients have such divergent viewpoints they can legitimately be called "The Two Cultures." In his classic book with that title (C.P. Snow, *The Two Cultures and the Scientific Revolution*, Cambridge University Press, New York, 1959), C.P. Snow explains that science and technology policymaking is particularly difficult because it requires the combined expertise of scientists and politicians, two cultures with little understanding of each other. Software developers and their clients in various industries have similar problems. For example, neither community has a good feel for the capabilities, hot-buttons, or difficulties of the other.

HOW DO WE SOLVE IT?

The key strategies for uniting the two cultures and managing client expectations involve clear communication and solid planning.

Speaking clearly

To defuse potentially problematic situations, you must first clarify what the customer really wants and why. In the

example I gave, we didn't really want a fully packaged new model in one week; we wanted quick access to a model, including the computer security cost driver, followed by a fully packaged new model sometime later. Tom Bauer clarified this distinction when he counterproposed a more achievable approach to meeting our requirements.

Next, you must work out the details and explain them. Tom Bauer's six-week estimate was clearly more realistic than the one-week estimate, once he articulated a more detailed set of necessary tasks. Customers usually focus on the visible-applications tip of the software iceberg, and are generally unaware of how much additional, essential software lies underneath.

Clear customer communication depends on two factors. First, you must choose the most suitable communications media: If a picture can be worth a thousand words, a prototype can be worth a thousand pictures.

Second, you must carefully define communications content. Does "interoperable" mean just CORBA-compatible or full plug-and-play guarantees? When you say "advanced information hiding techniques," will your customer hear "module descriptions confined to essential capabilities and not implementation details?" Or will they get the wrong idea and think that you're not going to let them know what their system is doing?

Simplifier and complicator lists can help developers gauge how much effort the client must expend to complete a given task—and vice versa. For example, we put out a series of 15 to 20 digital library applications annually, built by teams of computer scientists for groups of librarians. The two-cultures problem reared its ugly head when we saw that one group didn't know what was easy or hard for the other group to do. By having the groups become familiar with domain-specific lists of things to facilitate or complicate the application, we were able to reduce the first-review failure rate for these projects from roughly 25 percent to roughly 5 percent. In a multimedia archive application, for example, using standard query languages, search engines, and media formats made applications much easier for the development teams; while trying to achieve natural-language processing, automated metadata determination, or simultaneously achieving rapid access and high-fidelity media presentation made applications much more difficult. (Barry Boehm, Marwan Abi-Andoun, Dan Port, Julie Kwan, Anne Lynch, "Requirements Engineering, Expectations Management, and the Two Cultures," *Proc. 1999 Int'l Conf. Requirements Engineering*, IEEE CS Press, Los Alamitos, Calif., June 1999.)

Additional Reading

Several books have helped me find customer-information-gathering techniques that can be combined with expectations-management prototypes, scenarios, consensus-building questions, and brainstorming.

Naomi Karten, *Managing Expectations*, Dorset House, New York, 1994.
This book has numerous suggestions and examples on establishing clear communication, such as avoiding conflicting messages (including body-language messages), avoiding or carefully explaining jargon, establishing communication media preferences, and addressing perceptions as well as facts.

Donald C. Gause and Gerald M. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House, New York, 1989.
The chapters on "Preferences" and "Expectations" in this book provide further insights and techniques for prioritizing requirements and limiting expectations.

Roger Fisher and William Ury, *Getting to Yes*, Houghton Mifflin, Boston, 1981.
Part of the Harvard Negotiation Project, this book presents principled negotiation techniques.

Planning for success

Several planning tools can help you manage expectations and negotiate realistic requirements and deadlines. Software risk assessments and high-risk item identification techniques such as prototyping, benchmarking, and reference checking can highlight high-risk expectations, and can provide techniques for reducing those expectations, thereby yielding an acceptable but lower risk solution. One large TRW project that I worked on had a requirement for a one-second maximum response time, based on expectations that a subsecond response time would vastly improve productivity. The initial architecture required to achieve this performance would have cost $100 million to develop and had several high-risk elements. Faced with this situation, TRW and the customer developed and implemented a prototype, which indicated that users found a response time of four seconds was about as good as one-second 90 percent of the time. We revised the project to develop a much lower risk system with a four-second response time, at a cost of $30 million. Had we and they implemented the prototype two years earlier, it would have resulted in revised expectations and avoided two years' work in specifying an overly ambitious solution.

Use calibrated models to calibrate expectations. Another two-cultures problem is that customers and users have little feel for the difficulty of developing a given quantity of software within budget and on schedule. This leads to unrealistic expectations by customers that can be reinforced by developers' desire-to-please or desire-to-sell tendencies, leading to frequent budget and schedule overruns. Well-calibrated software cost and schedule estimation models have helped many customers and users recalibrate their expectations and produce realistic goals

for their projects. Calibrated performance and reliability models can perform similar functions.

Accept only one independent variable. One clear message from software cost and schedule models is that larger amounts of developed software require larger amounts of budget and schedule.

Express your needs as negotiable 'win conditions' rather than non-negotiable 'requirements.'

Yet many software developers accept both "fixed amount of software" and "fixed cost or schedule" as conditions to be simultaneously satisfied, without well-validated analyses to demonstrate that this approach is indeed feasible. If it's not, it's better to tell your customers that they can expect either a "fixed amount of software" or a "fixed cost or schedule," but not both. Called "cost (or schedule) as independent variable" (CAIV or SAIV), this last option has customers prioritize their requirements. Developers then implement a core set of highest-priority requirements, and continue adding lower-priority features so long as the budget or schedule lasts. Similar strategies can be developed for other incompatible combinations of independent variables.

COMING AWAY A WINNER

When applied to software requirements determination, the win-win approach involves all of a project's key stakeholders in the negotiation of a mutually satisfactory set of requirements (B.W. Boehm et al., "Using the WinWin Spiral Model: A Case Study," *Computer*, July 1998, pp. 33-44). It often involves concurrent engineering or joint application development by an integrated product team of stakeholders, as well as principled negotiation techniques. Reflexively, good expectations management facilitates win-win solutions. Tom Bauer's three-week delivery was a win if you were expecting six weeks; it would have been a loser if you were expecting delivery in one week.

Techniques for win-win requirements negotiation can produce advantages for expectations management and project outcomes. These techniques can help you

- express your needs as negotiable "win conditions" rather than non-negotiable "requirements," which creates a more flexible climate for adjusting expectations;
- assimilate other stakeholders' win conditions to provide a greater understanding of how their constraints may affect your level of achievable results and thus your expectations;
- sense that the other stakeholders are looking out for your win conditions, which often increases your willingness to help accommodate their win conditions by adjusting your expectations; and
- collaborate to understand each other's win conditions.

Using these techniques, you can help bridge the two-cultures gap and increase stakeholders' ability to invent new win-win options for mutual gain.

With the new millennium upon us, I hope all of us learn how better to manage expectations and seek win-win solutions. The alternative—win-lose situations—generally devolve into lose-lose situations, ensuring worldwide frustration and chaos. ✷

Barry Boehm is director of the USC Center for Software Engineering. He developed the Constructive Cost Model (Cocomo), the software process Spiral Model, and the Theory W (win-win) approach to software management and requirements determination. 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