Editor: Suzanne Robertson ■ The Atlantic Systems Guild ■ suzanne@systemsguild.com

Never Go to a Client Meeting without a Prototype

Michael Schrage

We hear a lot about how collaboration is the key to implementing the right requirements. But how can clients and developers achieve this collaboration? Michael Schrage gives us the benefit of his experience on how prototypes and professionalism make this happen. Let us know about your experiences with collaborative prototypes. —*Suzanne Robertson*

Every programmer with an ounce of brains and more than two ounces of experience knows a bitter truth: The road to software development hell is paved with "good" requirements.

Requirements are the "big lie" that organizations systematically tell themselves to prove that they really do analyze, prioritize, and optimize their software and network needs according to rigorously defined criteria—honest!

Requirements for miscommunication?

But every developer I know—no exceptions—ruefully tells the same pathetic software story. They talk of how they genuinely listen to their client's needs, how much they care, and how they're professionals. So, they gather up requirements and specs, circulating and refining them, until there's a genuine consensus and mutual understanding. The development team then goes off for 30, 60, or 90 days and builds a pretty decent prototype.

With pride and a flourish, the team then demonstrates this prototype to the client. You know what happens next. "Well," says the client with a disconcerting air of disappointment, "that's pretty much what we asked for, but now that we've seen it, we realize it's not what we really want. What we really need is …. Can we have another prototype by Thursday?"

Welcome to the worst of both worlds. The client now thinks the developers are a bunch of propeller-headed prima donnas who don't grok the imperatives of the business. The developers think the client is a fickle moron who doesn't know what he wants but doesn't hesitate to waste everyone's time trying to find out. Perhaps they're both right. Nevertheless, the punch line to this unhappy joke—now that credibility has been destroyed and mutual contempt established—is that the two groups have to work together to get the job done. Good luck.

The supply-demand mismatch

The banal pervasiveness of this programming cliché fascinates me. How can so many smart people make the same self-destructive mistake time and time again? Why do seemingly rational managers let this kind of dysfunctional development persist? What does this perennial pathology say about how organizations really pursue what they say they want to accomplish?

To my surprise, the answers to these questions are shockingly simple. Even more shocking, however, is that these simple answers make perfect sense. The trick is to think back to Economics 101.

Clients are responding all too logically to

 0740-7459/04/$20.00 © 2004 IEEE

the peculiarly perverse economics of requirements-driven software development. Requirements create markets where clients have unambiguous incentives to avoid rigorous thinking, shun effective risk management, and delegate the more difficult design trade-offs to IT. Perverse incentives yield perverse outcomes.

So what are these perverse incentives? What's wrong with the requirements marketplace in software development? Economics 101: A fundamental mismatch exists between supply and demand. Costs are misunderstood, benefits are misaligned, risks are mispriced, and rewards are mismanaged. The result? Software development succeeds in spite of "good" requirements, not because of them.

Costs without benefits

Think coldly and dispassionately about costs. How much does it really cost a client to come up with yet another "good" requirement? The answer, of course, is almost nothing. It's relatively fast, cheap, and easy for even quasi-intelligent clients to develop long lists of eminently reasonable features, functionalities, and requirements to build into the software.

So why should anyone be surprised by the surfeit of requirements, "enhancements," and "improvements" that inevitably materialize as development begins? It means we always have an oversupply.

But the problem gets even worse. Remember, clients are rewarded for generating "good" requirements, not for generating good software. Software is IT's job. Clients are rewarded for coming up with something that's cheap for them to produce, and for the client, requirements are an end in and of themselves: the requirement is the deliverable. For IT, requirements are a means to an end. This is a market mismatch.

In other words, the client's job is to give IT good requirements in exchange for good software. Indeed, many clients believe that their requirements don't impose a cost on IT as much as they confer a benefit—that is, a valuable insight into business process and priorities. Along with the development budget and schedule, clients think of requirements as a form of currency they give developers to help them better prioritize their own efforts. Clients collaborate with IT by sharing "good" requirements with them.

The economic asymmetry is obvious: it's orders of magnitude faster, cheaper, and easier to come up with "good" requirements than to generate good software. Yet top management in most organizations typically behave as if the requirements are just as valuable as the software.

Superficially, of course, this behavior makes sense. After all, wouldn't we waste a lot of time, money, and energy on software development if we didn't have such requirements? The economic reality underlying this argument, however, utterly annihilates its credibility.

Significant change is inevitable

As we know, requirements change. Always. Requirements change because perceived needs change, the business environment changes, budgets and schedules change, and top-management expectations change. Just as importantly, requirements change because individuals and institutions learn about processes and applications as they try to translate them into software. By definition, we can't know these changes in advance—if we did, they'd be part of the original requirements.

For the client, requirements are an end in and of themselves. For IT, requirements are a means to an end. This is a market mismatch.

Clients—let alone software developers—seldom know in advance which requirements will be most important or require the most modification or prove the most difficult to test or the riskiest to implement. The result is a "perfect storm" for development turbulence. Clients who are recognized and rewarded for generating a glut of requirements that might or might not change substantially as development proceeds collide with IT developers who don't know—who can't know—which requirements should be refined, revised, or removed.

Indeed, most IT shops aren't rewarded for refining, revising, and removing requirements—they're compensated for building to requirements. In other words, the IT default is to try to give clients what they say they want, whether the requirements make sense or not. (Which, of course, is why IT always has such powerful incentive to get clients to "freeze" requirements as early as possible or try to make it difficult or expensive to change requirements later in the development process.)

The analogy here is a client having a massive banquet on a tight schedule and an even tighter budget, with IT as the world-class caterer. The client, however, is responsible—and rewarded for—the menu not the meal. Negotiations go back and forth about the mix of chicken, beef, fish, and vegetarian meals. How many courses? Light or heavy dessert? Which appetizers? Who coordinates the wines? What is the likeliest number of guests? And so on.

The caterer, of course, can "outsource" appetizers and desserts. Similarly, the caterer can argue with the client about appropriate portion sizes to stretch the budget or push for cheaper but more filling appetizers to reduce the entrée cost. The caterer can surreptitiously substitute tap water for bottled water. The creative caterer has innumerable options for producing both a good meal and a healthy profit.

The problems, of course, come when the client calls the caterer after

the budget, schedule, and menu have ostensibly been decided. What does the caterer do when the client says 25 percent more people are coming and there's a 40 percent chance that half will be vegetarians? Or when the banquet must be rescheduled for a week earlier? Or when the client says the CEO now forbids alcoholic beverages, has cut the budget 40 percent, and declares that all the attendees are on the Atkins diet? Or when the client insists that his or her own proprietary recipes and wholesalers be used?

This example isn't meant to show that we must plan for contingencies but that experience teaches that we take requirements far too seriously. In portfolio management terms, most software development organizations are heavily overinvested in requirements at the expense of other development areas. This overinvestment represents a considerable waste of time, money, and opportunity.

Schrage's Iron Law of Requirements modestly codifies this overinvestment's magnitude: "The first client demo renders 40 percent of the listed requirements either irrelevant or obsolete—the problem is, we don't know which 40 percent." We accelerate past the point of diminishing returns on requirements far faster than we care to acknowledge.

Some critics assert that this "anti-requirements" stance is simply much ado about the so-called waterfall software development method. Not so. The perverse economics of requirements transcends waterfalls.

The requirements surplus

In fact, if we look at the rise of so-called agile programming methodologies and the Extreme Programming movement, what do we invariably find? We discover a de-emphasis on requirements and a focus on use cases that look at how people actually behave instead of what they say they want. In essence, at the XP development philosophy's core is a belief that actions speak louder than words. There's no requirements surplus here.

The rise of Enterprise Resource Planning represents an even more provocative approach to the requirements conundrum. What an SAP or Oracle effectively declares is, "To heck with your requirements; you should make your business processes conform to our requirements." In other words, they've created a de facto software monopoly that marginalizes the firm's internal requirements marketplace. Of course, when you try to customize this software, all those requirements pathologies resurface because the firm's requirements are subordinated to the software.

I've made a better than comfortable living advising software development groups to stop gathering requirements after the first 20 to 25 and then do a quick and dirty prototype to lure the client into codevelopment. Why? For two excellent market-tested reasons: First, you tend to get better quality requirements when they're generated by ongoing client interaction with a constantly improving prototype. Prototype-driven requirements ultimately lead to better apps than spec-driven prototypes.

The second reason relies more on Psychology 101 than Economics 101: clients are happy to cavalierly reject your work. They tend to think twice, however, before throwing out their own. In other words, when clients are vested in software development with more than just money, you get both a better development process and a better software product. The economics of software prototype-driven requirements are inherently less dysfunctional than the economics of requirements-driven software development.

You tend to get better quality requirements when they're generated by ongoing client interaction with a constantly improving prototype.

So what are the Productive Programmer's Prototyping Principles that should emerge from this shift from economics-oriented to cost-effective requirements? Here are three principles that I commend to my own clients. We discovered them the hard way, wasting considerable time, effort, and thought trying to get "better" requirements. Don't repeat my mistakes.

Design software as if actions speak louder than words

Listening to clients is more polite than productive. The purpose of every design conversation with a client should be to create a behavioral representation of the desired requirement. Whether a paper prototype, a storyboard, or a simulated screen sequence detailing a use case, the goal should always be to get the client to realize, react, and respond to an implemented requirement's possible implications. Programmers should design according to what people do as opposed to what they say.

Certainly, clients are always best observed in their "natural habitats" before any serious conversations about requirements begin. Indeed, at least two members of a programming team should spend on-the-job time at a client site before a serious, all-hands kick-off design meeting. Yes, this is obvious, yet fewer than half of the organizations I know do this with any degree of rigor or consistency.

Never go into a client meeting without a prototype

This is the opposite side of the Principle 1 coin. A model, prototype, or simulation—not a "risk-adjusted" set of requirements—should be the design-interaction medium between the development team and the client. Indeed, the development team, not the client, should be driving the conversation because clients need the

context of usable models and participatory prototypes to better articulate what they really wish to accomplish through the software. The word for programmers who would rather discuss client needs than observe their behavior is "unprofessional."

Clients are prototyping partners, not customers

We model, prototype, and simulate software with clients, not for them. The development team should educate clients who are unready or unable to be effective partners. Clients unwilling to be partners should be fired.

Professionalism isn't just about competence; it's about integrity. To be blunt, developers who won't collaborate on prototypes with clients are as unprofessional as clients who won't collaborate with developers. It's imperative that clients realize that they have a professional responsibility and accountability for cost-effective software development and deployment and that coming up with lists of pretty requirements for the propeller-headed geeks isn't acceptable.

The Prototyping Partnership Principle is the indispensable way of getting developers and clients to productively learn from each other from a position of parity rather than the typical master-slave asymmetry.

If the benefits of partnership don't outweigh its costs, partnership will fail. However, my work with companies worldwide has convinced me that the most robust software, and the most cost-effective software deployments, have been the products of prototyping partnerships between professionals.

Michael Schrage is a codirector of the MIT Media Lab's eMarkets Initiative and an instructor on innovation economics for several MIT executive education programs. He is author of *Serious Play: How the World's Best Companies Simulate to Innovate* (Harvard Business School Press, 2000). Contact him at schrage@media.mit.edu.