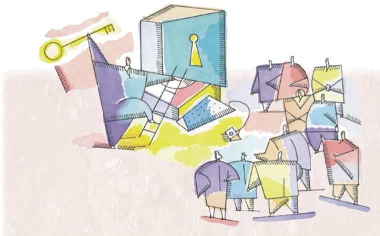
Reframing Requirements Analysis

James Bach, Reliable Software Technologies

Apple Computer, 1988. I was having a bad week. Apple's Development Systems Group was producing a tool called ASMCVT, the purpose of which was to convert assembly source files into a format suitable for a new assembler that was still under development. I'd been informed, as the leader of the corresponding assembler test team, that I would receive this tool in a few days. But no one had sent down—literally, since the developers worked on a floor above the lowly test department—any details about the tool. When I visited the developer and explained to him that we needed a spec in order to plan a decent set of tests, he told me that the tool was very simple and there would be no problem writing one.

The next day he left a single sheet of paper on my chair. It was a handsome little document, with "ASMCVT" emblazoned in bold 30-point Helvetica across the top of the page. There were other markings, too. As I recall, it had a couple of horizontal rules, the developer's full name, the date, a number of category titles with "N/A" written beside them, and ample white space. At a distance it might pass for a genuine technical document—a tribute to the nameless scribe who designed the document template. But it contained only one sentence of genuine technical content: "This tool converts files from the old assembly format into the new format."



Software development is an exploratory and self-correcting dialogue, complete with stuttering, hemming and hawing, and Freudian slips.

The developer seemed surprised when I stomped into his cube and scowled at him.

TALKING ABOUT RISK

He offered a feeble defense. "There are almost fifteen hundred different transformations needed to turn the old files into the new files. Surely you don't expect me to write them all down." Yes sir, I do. If you can code it, you can document it. My next stop was the cube of my boss, Chris Brown, Development Systems Quality manager.

"Fifteen hundred, Chris. That's what he said. But look at this," I held the spec overhead and dropped it. It fluttered through the air like a feather. "He calls this a specification. I refuse to test this product until he tells me what it does."

Chris seemed unimpressed by my demonstration. "Jim," he said, "We need to talk about risk."

To my surprise, Chris didn't see a problem with testing an unspecified tool. He explained it like this: "ASMCVT is a tool that takes a text file as input and produces a new text file as output. It's a one-shot process. It doesn't change the original file in any way. Both the original file and the new one are human readable. Errors in the new file will be easy to detect simply by running it through the new assembler, which will either report the error or produce an object file that is not bit-for-bit equivalent to the object file produced by the old file and the old assembler. If we don't provide this tool, our customers will have to convert by hand, whereas even a buggy tool will potentially save them a lot of work."

"This is not a high-risk situation," he continued, "and James, testing is nothing more than risk management. Even a simple testing process, based on our assumptions about what the tool does, could be enough to manage the risk of an embarrassingly bad tool. Besides, you already know a lot about the differences between the old assembly syntax and the new syntax. If any specific questions arise about the functionality or risks of the tool, sit down with the developer and ask him nicely. But don't expect an open-ended brain dump.

"Just run a heap of source files through ASMCVT," he concluded. "Examine the results. Report anything that seems strange. If there's an important problem in there, you'll probably find it."

DISCIPLINE OR DELINQUENCY?

I was annoyed that my manager would contradict a well-established doctrine of software engineering: Software should be implemented in accordance with a clear specification. Although I couldn't fault his analysis as it applied to the situation at hand, I worried that such situational reasoning would provide an excuse for resisting good software development processes. What I didn't realize at that time was that my view of the "good processes"—and of the relationship of people to process—was naive. I thought of process only as a defined sequence of tasks, whereas Chris viewed it as a way for a team to achieve a goal.

The idea that software processes should be rigorously defined, repeatable by anyone, and followed carefully is popular among senior software managers and SQA enthusiasts. They want control and predictability, and they think they can get it by promoting that kind of discipline. The problem with this otherwise reasonable point of view is that, in practice, what passes for rigor and discipline is so often just dogmatic and thoughtless behavior. The problem of poor process discipline is thus replaced by process fixation—the obsessive pursuit of preconceived tasks and goals regardless of their value in the present context.

What distinguishes a fixation is not that it is the wrong thing to do, but that we do it for its own sake. We serve what we perceive to be the needs of the process, rather than knowing what we need to solve a genuine problem. In the ASMCVT case, I became fixated on the goal of getting a specification that itemized each data transformation performed by the tool, even though I didn't actually need that document in order to fulfill my true mission as a test manager. I also fell into fixating on the goal that the developer use his time to follow the official process, regardless of what other matters might be more important.

For his part, the ASMCVT developer was task fixated. He did indeed write a specification. He followed the protocol laid down in our defined process by using the official specification template. But what he produced completely ignored what I needed. We did not negotiate effectively to resolve our conflict. This is a case where the defined process by which we operated did not serve us.

Unfortunately, many of the processes we choose to standardize in our projects and organizations are based on fundamentally wrong ideas about how successful teams actually create software. Our desire to control software projects has seduced us into embracing unhelpful manufacturing and engineering metaphors. Software development is, in fact, dominated by human cognition, not machinery, brute labor, or the laws of physics. Software is a collaborative invention. Software development is an exploratory and self-correcting dialogue, complete with stuttering, hemming and hawing, and Freudian slips. Out of that dialogue emerges a product. When software process is forced into a manufacturing mold, we cripple the dialogue, or drive it underground.

Unfortunately, many of the processes we choose to standardize in our projects and organizations are based on fundamentally wrong ideas about how successful teams actually create software.

How can software development be reframed as a goal-seeking dialogue? As an example, let's look at requirements analysis and documentation.

REQUIREMENTS REVISITED

In generally accepted software development practice, requirements analysis is supposed to happen before design, and design is supposed to happen before coding and testing. Some kind of document is supposed to come out of requirements analysis. The IEEE 830-1993 Recommended Practice for Software Requirements Specifications includes a list of quality attributes for requirements: correct, complete, unambiguous, consistent, ranked for importance, verifiable, modifiable, and traceable. The specification is supposed to guide the engineering activities that follow.

The so-called generally accepted practice, however, isn't generally practiced. As I typically encounter them, requirements specifications are tossed salads of ambiguous problem statements and design decisions: They are expressed too vaguely or too precisely, usually require substantial background knowledge in order to interpret, and become obsolete or forgotten halfway through the design phase. Does that sound familiar?

There is at least one straightforward solution to poor requirements specifications within the framework of accepted practice: Do more of it. Instead of compressing the process into a scant few weeks of interviews, meetings, and late-night writing, spend months on it. Keep at it until there is no more ambiguity or design risk. Use testers, simulators, full-scale prototypes, and focus groups. Explore the full range of technological solutions and evaluate buy-versus-build strategies. Send the developers, incognito, to the client site and have them live among the potential users like field anthropologists.

There's abundant literature about the techniques of excellent requirements exploration and definition. Knowing what can be done is not the main problem, though. The main problem is that almost no one believes that any of that is necessary. Most either think that a quick-and-dirty requirements analysis is good enough, or else don't realize that their process is quick and dirty. Another problem is that few have the skills to execute all the analysis needed to get it all done in one round.

A GOAL-SEEKING DIALOGUE

On the level of brass tacks, all that's really needed to produce a good product is a set of ideas about what the product should be that is understood by the developers, practical to implement, and (if implemented) would lead to a sufficiently desirable product. We should use whatever process that satisfies these needs well enough and in good enough time. It may even be possible to satisfy these needs without writing any kind of requirements document or having any defined requirements analysis phase. Whatever the value of documents and phases, they are only a means to an end, and staying clear about means and ends is a powerful defense against fixation.

In the words of requirements expert Brian Lawrence, "Many people focus on requirements as existing in a requirements specification. Requirements are the drivers of design choices—the rea-

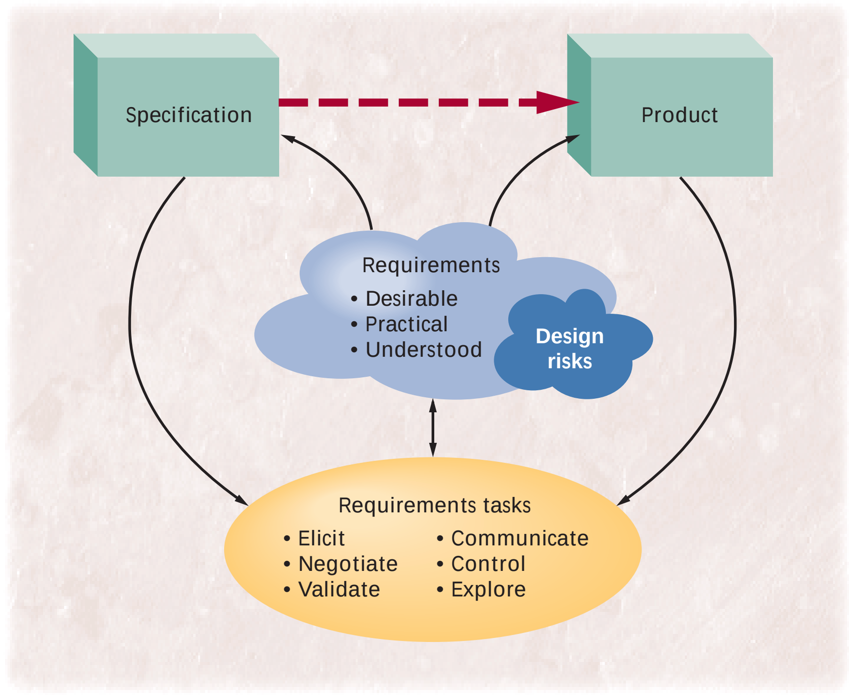


Figure 1. Requirements evolution. The cloud labeled "requirements" represents the requirements as understood. The block labeled "specification" represents the requirements as documented. The cloud labeled "design risks" represents the problems that may occur due to poor requirements. There are two dialogues that proceed simultaneously: defining and manifesting. The defining dialogue, on the left, is the cycle that refines understanding of what is desired. The manifesting dialogue, on the right, refines understanding of what can be reasonably achieved.

sons why we decide to build what we build. Those reasons exist inside our minds. So the full set of requirements exists in the shared mind space among all the product stakeholders." A requirements document is only a model of the information in that mind space, but it helps us manage the risk of misunderstood, impractical, and undesirable requirements.

Figure 1 is a depiction of requirements evolution as a dialogue, rather than as a monolithic phase. Any requirements process, from rigorous clean-room development to pure hacking, could be described in terms of this model, which describes two dialogues that proceed simultaneously: defining and manifesting. The defining dialogue refines understanding of what is desired, while the manifesting dialogue refines understanding of what can reasonably be achieved. The entire dialogue is a goal-directed activity. The goals are, in order of precedence:

1. Produce a good product.
2. Ensure that requirements are understood well enough—and are sufficiently practical and desirable—so that design risks that threaten the first goal are acceptably small.
3. Create a requirements documentation, as necessary, to support the first two goals.

Immediate practical implications flow from this model:

- A requirements specification is a tool to facilitate requirements management. Even if it's incomplete or ambiguous, it can still provide useful clues that promote a deep, shared understanding about requirements.
- Inasmuch as the requirements document is incomplete, it should not be used as the sole basis for future work. It should be augmented by other channels of information, such as oral communication, domain expertise, and prototypes.
- Requirements specifications are no substitute for requirements discussion and negotiation.
- The requirements phase of development is never where all the requirements definition happens, but rather where enough happens to gain commitment and proceed without incurring an unacceptable risk of a buggy product and massive rework.
- Requirements negotiation continues as technological limitations are encountered and explored. Requirements priorities may change in the face of the difficulties and costs of realizing them in a product.

When we fixate on an elaborate requirements process, we try to persuade the powers that be to let us perform that process, and bemoan our lot when they don't support us. When we fixate on a quick process, we try to get it over with, doing the minimum needed to satisfy the process fanatics. There is a third way, however. We can reject process fixation and solve problems instead. We can reframe the requirements process as a goal-seeking dialogue whose purpose is to manage the risk of building the wrong product.

In the ASMCVT situation, the dialogue was very simple. Chris reasoned about requirements and came to the conclusion that we did not need an elaborate specification detailing the external design of this particular tool in order to satisfy our testing mission. History proved him correct, as far as I can tell. We shipped the tool without incident. But his demonstration had effects far beyond saving labor on a minor test project. He got me started on a ten-year odyssey, investigating methods of risk analysis and management, and learning to be reasonable. ❖

Acknowledgments

Thanks to Brian Lawrence (http://www.coyotevalley.com) for helping me understand the requirements for this article.