July/August 2008 $9.95 www.StickyMinds.com

BETTER SOFTWARE

The Print Companion to StickyMinds.com

HIP-HIP HOORAY!
Praise for ambiguity

DOUBLE YOUR FUN
8 Benefits of pair programming

Two Cheers for Ambiguity

by Michael Bolton

DREAMSTIME

I was sitting at the back of the room, munching on a donut, sipping a coffee, and listening to the presenter talking about the importance of unambiguous requirements. "What does he mean by 'unambiguous'?" I wondered.

He also seemed to be opposed to jargon—yet, jargon, to those who use it, is extremely precise and unambiguous. "Jargon" and "ambiguity" are like "quality" or "purpose"—subjective and context-dependent, not properties of something but rather a relationship between some person and the thing. Ambiguity may be a problem or it may not, depending on its meaning and significance to some person.

I was reminded of all this recently when a colleague observed that he avoided using words like "skill," "diversity," "problems," and "mission," because he found them inherently ambiguous. I replied that I use these words constantly for exactly the same reason. I find them to be not only necessary but also useful as gauges for assessing consensus on a project and how we get to that consensus.

Some people are comfortable with ambiguity; others are not. We can spot a member of one group or the other by the answer we get when we ask, "Can we talk about what you mean by 'skill' (or 'diversity' or 'test' or 'bug')?" When someone responds, "Well, for example …" or "In this context …" or "There are many possible meanings, but around here we mean …" then we know we're in good company. We can have an evolving, ongoing conversation that helps us work together and understand one another, and if we discover that we don't have a consensual understanding of something, we can work it out. On the other hand, when someone responds, "Isn't that obvious?" or "Why do you keep going meta?" or "Can't we talk about practical stuff?" then we know that there's work to be done, because someone is suffering from that terrible disease, Single Model Syndrome.

Single Model Syndrome is the silver bullet for ambiguity problems; something is unambiguous when there's no possibility of a second interpretation of it. On the other hand, Single Model Syndrome can lead to frightful misunderstanding, especially when two people suffering from it—and using different models—show up at the same meeting.

The battle against ambiguity has to do with the problem of closure. Psychologically, some people are comforted by closure and require it; others don't require it and, in fact, may be leery of it. Developers and project managers tend to value closure because it gives them a finish line, a clear goal that can be met. Good testers are aware of the risk of closure, especially when it's premature. Suspending conclusions helps us to see more alternatives and to adapt to change, both in problems and in solutions. Seizing certainty at the requirements stage cuts us off from alternative approaches and new information. One common manifestation of this problem is an excessively detailed test plan—one that doesn't match the product that is eventually delivered.

For testers in particular, *recognizing* ambiguity is useful. Recognizing ambiguity is naturally important because ambiguity is a way in which misunderstanding may provide homes for bugs in the product. Yet ambiguity, which implies more than one possibility, might also be a blessing in disguise. An ambiguous sentence might trigger a discussion about what we perceive, what we agree upon, and what we don't yet understand. An ambiguous word might have several open-ended meanings that help reduce tunnel vision. An ambiguous problem statement might remind us that there are often several alternative approaches to solving a problem. The precise expression of a requirement *might* make testers' lives easier, but perhaps the meaning and the significance of the requirement are more important, even though they may be imprecise.

James Bach tells a wonderful story that illustrates the distinction. On a project several years ago, a junior tester asked James to help interpret a line in the requirements document that said, "When the user presses the touchscreen, the system shall respond within 300 milliseconds." Holding a stopwatch in one hand and using the system with the other seemed impractical, and automation seemed to have a high development cost, so James decided to train the testers to recognize 300 milliseconds. He bought an inexpensive stopwatch for each of the testers. They went to lunch and practiced turning their stopwatches on and off until they could estimate 250 milliseconds, plus or minus fifty, with rea-

sonably good precision. Then someone brought up the question, "What if we were off by ten milliseconds, and the system took 310 milliseconds to respond? That would be a failure, but would that be a *problem*?"

James realized that he had considered the precise wording of the requirement—a shallow sort of meaning—but neither its deeper meaning nor its significance. He went to the project manager for clarification. It turned out that the previous version of the program had taken upward of seven seconds to acknowledge that it had received input, and customers had found this annoyingly slow. The designers had put the precise timing—300 milliseconds—into the specification because they had thought that you couldn't use words like "annoyingly slow" with testers. James suggested instead that the developers show the testers the old system so the testers could understand the problem from a visceral perspective. In this case, ambiguity was disguised as precision—and this was no place for stopwatches.

So how should we deal with ambiguity in requirements and elsewhere in the project? How do we seek it, and how do we resolve it? Here are some heuristics.

The first thing is to recognize that the *requirements* are not the *requirements document*; at best, the document is a stand-in for the ideas of one or more real people. Recognize that all statements, whether written or spoken, are potentially ambiguous, but the ambiguity might not represent a problem for the project, so look for ambiguity that *matters*. A testable requirement is not necessarily one that is painstakingly precise, mathematically falsifiable, or unfailingly unambiguous. A testable requirement is one that helps us ask and answer the question "Is there a problem here?"

Conversation is a fast and powerful approach to discovering and resolving ambiguity. Ask plenty of questions and watch for disagreements on the answers from various people; then seek consensus on meaning and significance. There can be several levels to the conversation—one in which we're talking about something, another in which we ensure that we agree on what we're talking about, and yet another in which we work out a protocol for resolving our differences. This may seem like extra work but, in fact, people are doing it all the time. The trick is to do it *consciously*.

Make sure that conversations are supplemented by a wide variety of media—whiteboards, tables, scenarios, mind maps, wikis, knowledge-crunching sessions, diagrams—in addition to the more traditional forms of documentation. Be skeptical that any one document will identify all the things you need to know about your project.

When you spot ambiguity problems, make the problem clear by pointing out alternative interpretations: "There could be a bunch of testing missions in play here—finding important problems quickly, investigating the problems we've found, assessing backward compatibility, identifying new risks. What can we agree on as the primary goal?"

Don't feel obliged to document minute details of every discussion. Documentation may have high cost and low value when consensus is the goal. Some things on a development project are so important that we *don't* write them down; instead, they become part of the project culture. (Everyone remember to breathe!)

Above all, remember Jerry Weinberg's definition of a tester—a definition that highlights the significance of ambiguity in our work: "A tester is someone who knows that things can be different." {end}

How do you know what you know? How do you know it? And how do you know that others understand things the same way?

Follow the link on the StickyMinds.com homepage to join the conversation.

Sticky Notes

For more on the following topic go to www.StickyMinds.com/bettersoftware.

- Further Reading