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

Seeing More of the World

Michael Jackson

Requirements engineering experience shows that failure to look at all aspects of the problem space results in missing or incorrect requirements. Michael Jackson provides a systematic approach for identifying which parts of the world require your attention, along with some easily applied heuristics for discovering whether you've looked far enough.

—Suzanne Robertson

Requirements engineers must look hard at the world. Peter Neumann's risks forum (ftp.sri.com/risks) has a long list of failures in software-based systems—some catastrophic, some serious, and some just inconvenient. Many are due to faults in the system's functional requirements.

Here's one catastrophic example. A US soldier in Afghanistan used a Precision Lightweight GPS Receiver—a "plugger"—to set coordinates for an air strike. He then saw that the "battery low" warning light was on. He changed the battery, then pressed "Fire." The device was designed, on starting or resuming operation after a battery change, to initialize the coordinate variables to its own location. The resulting strike killed the user and three comrades.[1]

It's hard to avoid defects in functional requirements. After a major failure, it's usually easy to see where you should have paid more attention and exercised more care, but it's not so obvious while you're developing the system. There are so many places to look, especially if you don't know what you're looking for. There's no easy answer for this problem.

Keeping it simple

One classic answer is, Keep It Simple. In his Turing Award lecture, Tony Hoare said, "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies."[2]

This is a good reason to keep your requirements descriptions as simple as possible. You should be reluctant, for example, to use UML 2.0's new, improved sequence diagram structures, which give you—in effect—an extraordinarily complex programming language. Use it and your requirements will certainly be complicated enough to have no obvious deficiencies. But that's hardly what you want.

Achieving this kind of simplicity isn't enough in itself to reduce the faults in your functional requirements. The difficulty is that requirements aren't about the software but about the *problem world*, which is always potentially complex, even for a system whose software is itself quite simple.

The world and the machine

Let's be specific. For an e-commerce system, the problem world is the business organization, its customers, suppliers, warehouses, product and fulfillment subcontractors, the credit-card companies, and so on. For an elevator control system, the problem world is the lift shafts, the cars, the winding gear, the buttons, the floors, the passengers, the doors, and more. For a welfare agency system, the problem world is the welfare recipients, their families, activities and entitlements, any other agencies they deal with, the government department that provides the funds, the postal service, the agency's local offices and staff, and so on.

The problem world interacts with the machine at the *machine-world interface*, where they meet. For the e-commerce system, they meet at the customers' browsers and at the ports where EDI (electronic data interchange) messages are exchanged with suppliers and credit-card companies. For the elevator system, this is where the computer's ports are connected to the elevator equipment's sensors and actuators. For the welfare agency, it's at the ports where EDI messages are exchanged with banks, at the PCs in local offices where staff interact with the central computers, and at the printers where letters are printed that will be posted to welfare recipients.

Requirements engineering must focus on the problem world from two related perspectives. *Outer requirements* describe the effects the stakeholders would like to experience in the problem world. Waiting elevator passengers want the lift to come when they press a call button. The e-commerce customers want to be able to order goods and have the right ones delivered, and the company wants to manage its inventory efficiently and receive payment. The welfare agency wants to pay out the correct benefits, and the recipients want to receive their welfare checks and clear statements of their entitlements.

Inner requirements specify how the machine should behave at the machine-world interface to ensure that the whole system satisfies the outer requirements. Software design and programming make the machine behave this way, but choosing, designing, and specifying the behavior is the vital second part of requirements engineering.

We distinguish inner from outer requirements because the stakeholders' requirements are scarcely ever located at the machine-world interface. Almost always, some inherent *problem world properties*—some causal chains, some intermediate behaviors—separate inner from outer requirements. The elevator controller can send the car to a floor only because the motor and the winding gear respond a certain way to machine outputs, and the sensors react a certain way to a car traveling in the lift shaft. The e-commerce customer can receive the ordered goods only because the fulfillment subcontractor responds in a certain way to EDI messages coming from the e-commerce system. The welfare recipients can receive their checks only because the banking and postal systems operate in a certain way.

Adopting or rejecting a systematic method

A systematic way to address inner and outer requirements and their relationship is to separate the output of your requirements engineering work into three distinct but related parts. In the first part, you capture the outer requirements; in the second, you capture the problem world properties that let the machine guarantee the requirements; in the third, you specify the inner requirements—that is, the machine behavior that's needed at the interface. Then you can show—perhaps even by formal reasoning—that if the machine has the specified behavior and the problem world has the properties you describe, the whole system will satisfy the requirements. You don't necessarily have to do this for the whole system; you can do it for just one critical part.

However, you're probably not going to do it at all, for reasons good or bad. Perhaps you're doing what Walter Vincenti[3] calls *normal* rather than *radical* engineering. The system might be critical, but the inner and outer requirements and the problem world properties are all familiar, as is the structure and content of the software you must build. Perhaps your project is small and has low value and low risk; if you can't build it quickly and cheaply it's not worth building at all. Perhaps you've been convinced by an advocate of agile methods that careful documentation and reasoning are merely "ceremonial"—what Walter Bagehot would have called the "dignified" rather than the "efficient" parts of system development.[4] Or perhaps you're just ready for a small improvement in your approach to development, not a fundamental change.

Some heuristics

Without being fully systematic, you can still improve matters significantly by reviewing your requirements in an informal way. The idea is to try to find some of the functional requirements' faults before they emerge as failures in system operation. Here are a few heuristics—rules of thumb to help you find what you're looking for—that can help you see possibilities in the problem world that you might have missed.

Abandoned interaction

Start from a use case, or even from a piece of code, describing the machine's behavior in a compact episode of interaction with another party—a user, an operator, or another system. What happens if the other party abandons the interaction? In one e-commerce system, customers register by entering an email address on the first page, repeated for confirmation; they then enter additional personal information on the second page. A would-be customer who takes a coffee break while completing the second page gets an unpleasant surprise: the system times out, leaving an incomplete record keyed to the email address. Any further attempt to register is rejected because an incomplete registration already exists at the given email address. The abandoned registration can be neither bypassed nor completed.

Use case sequence

Start from any set of use cases with an actor or some other entity in common. What sequences of those use cases might be executed? What's the effect in the problem world of each particular sequence? Here, you're looking at the requirements with a larger *granularity*—not just single use cases but some individual participant's history of use cases. The plugger failure is an egregious example. Looking at each use case individually, it's easy to miss the lethal effect of the sequence `SetCoordinates-ChangeBattery-Fire`.

Attribute-change event

Start from any object attribute intended to model some problem world state or property—for example, a name, address, or birthday. What might happen in the problem world to change the attribute's value? The requirements for one insurance system assumed that birthdays, becoming known when a birth certificate is presented to the insurance company, could not then be changed. But some immigrants from countries with lax or chaotic administrations arrive with dubious birth certificates and subsequently want and need to provide correct information. The system provided no means for the necessary changes. (It didn't provide for gender change, either.)

Events without a use case

Start from any human—or other—actor participating in one or more use cases. What other significant events can the actor participate in for which the requirements provide no interaction with the machine, either through a use case or otherwise? Human actors, in particular, could die, emigrate, become bankrupt, go to prison, or even just lose things. In one insurance system, policyholders could obtain replacement documents if necessary—for example, because the existing document was old and had worn out. The `ReplacePolicy` use case required the policyholder to enter the document identification from the document they wanted to replace. Unfortunately, the most common reason for wanting a replacement document was that the original had been lost. The policyholder would be unable to supply the required information. There was no need for a `LosePolicyDocument` use case. But the event of losing the policy shouldn't have been ignored.

Following the chain

Start from any system output—for example, a printed document or an EDI message sent to another system. What chain of events can flow from this in the problem world? What events are expected but might fail to happen? What if there's a delay? One sales system, in a country where Internet usage is still low, relied heavily on postal communication for some of its interactions with customers. If a customer fell behind with installment payments, the system sent a demand letter asking for immediate payment and enclosing a bar-coded payment slip. The system requirements didn't account for the effects of postal delay. Some customers who had fallen behind spontaneously sent additional payments while the demand letter was in the post. Because the payment made wasn't accompanied by the bar-coded slip, the system couldn't apply it to the outstanding demand. The result was often a cycle of crossing communications that was difficult to break.

None of these faults is subtle or complex. They're all obvious and simple once your attention has been directed to them. That means, above all, once your attention has been directed to the relevant behavior and properties of the problem world. These faults aren't visible in the machine-world interface. The heuristics won't find all the faults in your requirements, but they should find some. You can extend the set by adding your own. It's a game, really. Have fun!

References

1. V. Loeb, "'Friendly Fire' Deaths Traced to Dead Battery: Taliban Targeted, but US Forces Killed," *Washington Post*, 24 Mar. 2002, p. 21.
2. C.A.R Hoare, "The Emperor's Old Clothes," *Essays in Computing Science*, C.B. Jones, ed., Prentice-Hall, 1989.
3. W.G. Vincenti, *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*, Johns Hopkins Univ. Press, 1993.
4. W. Bagehot, *The English Constitution*, P. Smith, ed., Cambridge Univ. Press, 2001.

Michael Jackson has worked in software development and development methods since 1961. His most recent work focuses on analyzing and structuring software development problems and solutions. He has described this work in his books *Software Requirements & Specifications* (Addison-Wesley, 1995) and *Problem Frames* (Addison-Wesley, 2001). Contact him at jacksonma@acm.org.