Requirements When the Field Isn't Green[1]

Karl E. Wiegers
Process Impact
www.processimpact.com

Most books and articles on software requirements are written as though you're gathering requirements for a brand-new product—what's sometimes called a *green-field* project. In reality, few people have that opportunity on every project. Many developers work on maintenance projects. In such a project you're usually adding new features to existing systems, modifying functionality to comply with updated business rules, or building extensions onto commercial packages. Too often, you can't find anything resembling a requirements specification for the system you're enhancing. It's not unusual to have to maintain a system without adequate documentation, with the original developers who held all the critical information in their heads being long gone.

If you're in such a situation, you might be tempted to reject the requirements literature as irrelevant. Not so! You can apply several useful requirements engineering methods even in these situations—even when you're squeezing new functionality into an unruly product in a documentation vacuum. The same ideas apply to iterative development projects doing a series of successive releases. Here I'll describe seven principles to help you deal with requirements issues in such maintenance situations. We'll also look at ways in which thoughtfully applied requirements development and management practices can help you improve both your product quality *and* your ability to perform future maintenance.

Principle #1: Don't dig your hole any deeper.

Let's do a little survey. When people on your project team have needed to add a new feature or change the current functionality of a product, how often have they had to reverse-engineer from code just to get enough knowledge of an ill-documented system to let them? I thought so. Nearly every maintainer has had to do this at some time.

Now a follow-up question (and I'm not trying to embarrass anyone here): How often did your team *capture* what they learned from such reverse engineering in a shareable way?

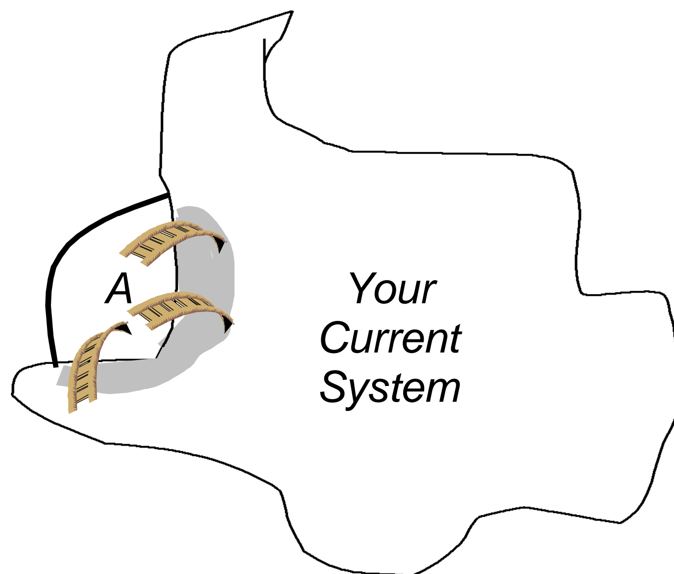
Just as I suspected. When we gain some insights from painfully exploring an existing black box system, we rarely take the time to record them for posterity. And in this case "posterity" is no farther away than you or your colleagues having to work on the same system in a few months. So it's worth taking some time to document your newly discovered understanding in the form of requirements or design descriptions. This will put your team in a position to perform future enhancements more efficiently. If you're performing maintenance in what's already an information void, vow to not dig the black hole you're in any deeper as you add enhancements.

---

[1] This paper was originally published in *STQE*, May/June 2001. It is reprinted (with modifications) with permission from Software Quality Engineering.

Perhaps your current system is a shapeless mass of history and mystery, like the one in Figure 1, and you've been asked to contribute some new functionality around region *A* in Figure 1. Begin by recording the new requirements in a structured, if partial, software requirements specification (SRS) or a requirements management tool. When you add the extra functionality, you'll have to figure out how new screens or features will interface to the existing system, as illustrated by the little bridges between region *A* and your current system. This analysis in turn provides a bit of insight into the shaded portion of the current system. This is the new knowledge you want to capture!

Figure 1. Enhancing an obscure system provides some visibility into it.



One technique you might find useful for doing that is to draw a dialog map—a user interface architecture—for the new screens you have to add, showing the navigation connections to and from existing display elements. (See my book, *Software Requirements*, for more information about dialog maps and many other requirements engineering good practices.)

By building a requirements representation that includes portions of the current system, you've accomplished two things. First, you've halted the knowledge drain, so future maintainers better understand the changes you just made. Second, you've collected some information about the current system that previously was undocumented. As you perform additional maintenance over time, you can extend these fractional knowledge representations, thereby steadily improving your system documentation at low cost. Think of it as a kind of "requirements refactoring," a way to leave the modified system in better shape than you found it.

Other modeling techniques (such as class and interaction diagrams, data flow diagrams, and entity-relationship diagrams) are also useful for representing system requirements and structural and behavioral information. Even the venerable context diagram has its place, showing the external entities or actors that connect to the system.

Another way to begin filling the information void is to create data dictionary entries as you add new data elements to the system and extend or alter existing definitions. Perhaps you

Copyright © 2001 by Karl E. Wiegers

need to support a new user class, or maybe you've subdivided a current user class into two groups who now have different needs. (User class descriptions also are an important part of a product's requirements domain.)

Many problems show up at interfaces—including software-to-software, software-to-hardware, and software-to-people junctures. When you implement a change, take the time to document what you discover about the external interfaces of the current system or the commercial package you're extending. These interface descriptions could include application programming interfaces (APIs), communication protocols, files and their formats, function argument lists, event lists, and so on. Interface definitions are part of your system architecture. Architectural information about how the system components fit together can help you add future extensions safely and efficiently. For suggestions on how to represent architectures, see *Software Architecture in Practice* by Len Bass, Paul Clements, and Rick Kazman (Addison-Wesley, 1998).

If you have to perform software surgery—because of a new business rule, a revised government regulation, or company policy—begin building a list of the pertinent business rules. If you don't, vital information will remain distributed among many different team members' brains. That kind of information fragmentation contributes to wasted time and effort as team members work from different interpretations of the important business influences on your project.

The key point here is that if you learn something useful about the current system during reverse engineering, save future workers the pain of having to rediscover that same insight. The incremental cost of recording your newly found wisdom is small compared to the costs of starting over researching each enhancement or having critical system knowledge walk out the door to another employer.

Principle #2: Practice new requirements engineering techniques.

Even if you're motivated to try some unfamiliar requirements engineering techniques, it's hard to find the opportunity. You'll never get a nice, quiet spell between projects to invest in learning something new.

But if you don't carve out time to climb the learning curve, you'll never be prepared to use better practices on the *next* big project. Minor enhancements provide an opportunity to learn how to make new requirements methods work for you. This is a way to tackle the learning curve in bite-sized chunks, so when the next new project comes along you'll have some experience and confidence in an improved toolkit.

As an example, suppose that Marketing (or a user) requests that you add a new feature to a mature product. This is your opportunity to explore the new feature from the *use-case* perspective, discussing with the requester the various tasks that the user will be able to perform using that feature. Use cases are a powerful method for understanding requirements from the user's business point of view. They also help you avoid overlooking specific functional requirements that developers must implement to let the users perform the intended tasks. If you're not already proficient with use cases, a new feature request gives you an opportunity to wrestle with use case concepts and application. This can reduce the risk of implementing use cases for the first time on a green-field project, when your ability to use them effectively might make the difference between high-profile success and failure.

Copyright © 2001 by Karl E. Wiegers

Note that you probably won't have an opportunity to apply *every* available requirements engineering method to your maintenance work. For example, JAD (Joint Application Development) is a powerful elicitation method that brings analysts, developers, and customer representatives together in a facilitated workshop. JAD would likely be overkill for exploring anything other than a major feature enhancement, but its principle of close collaboration between development and customers *is* something you could apply to smaller projects.

Some of the other techniques that you should look for opportunities to try on a small scale during maintenance include:

- verifying that the stated requirements will truly satisfy customer needs
- inspecting requirements specifications
- writing test cases from requirements
- defining customer acceptance criteria
- building user interface or technical prototypes
- modeling requirements
- specifying quality attributes and performance goals
- creating a data dictionary
- using a commercial requirements management tool to store requirements information

For details on these and many other practices, see my own *Software Requirements* (Microsoft Press, 1999), *Requirements Engineering: A Good Practice Guide* by Ian Sommerville and Pete Sawyer (Wiley, 1997), *Mastering the Requirements Process* by Suzanne and James Robertson (Addison-Wesley, 1999), or *Managing Software Requirements* by Dean Leffingwell and Don Widrig (Addison-Wesley, 2000).

Principle #3: Reconstruct requirements selectively.

It's rarely worth taking the time to regenerate a complete requirements specification for an entire production system (even though I have seen some companies do just that, for different and sensible reasons). Of course, you have many options between the two extremes of (A) continuing in perpetuity with no requirements documentation, and (B) launching a massive effort to reconstruct a perfect SRS. To help make the right decision, know precisely why you'd like to have written requirements available, so you can judge whether the cost of rebuilding all or part of the specification is a sound investment.

I know of one company that needed a comprehensive set of test cases for their current flagship product, a complex and mission-critical financial industry application. The analysts began by using a commercial tool, Rational Rose, to reverse-engineer class diagrams from their production code base.

Next, they wrote use case descriptions for the most common user tasks, some of which were highly complex. These use case descriptions incorporated knowledge of all the scenario variations performed by actual users, as well as an understanding of many exception conditions

Copyright © 2001 by Karl E. Wiegers

the system handled. The analysts drew sequence diagrams to further illustrate the actions involved in the use cases and tie them back to the system classes. The final step was to manually generate test cases from the use cases. When you create such requirements and testing artifacts by reverse engineering, as this company did, you can be confident they reflect the system as actually built and its known usage modes.

Your product's current stage of maturity is another factor to consider when contemplating after-the-fact requirements specification. The earlier you are in the product lifecycle (not the project lifecycle), the more worthwhile it will be to tune up the requirements specification. I visited one organization that was just beginning to develop the requirements for Version 2.0 of a major new product. They hadn't done a good job on the requirements for Version 1, which was currently being implemented. Was it now worth going back to improve the SRS for the first version of the product?

To answer that question from the lead requirements analyst, I asked him how long his company expected this product and its successors and relatives to be around. He said his company anticipated that this product line would be a major revenue generator for at least ten years. They also planned to reuse some of the core requirements in several spin-off products.

In this case, I recommended making the investment in improving the requirements documentation for Version 1, because that provided the foundation for all subsequent product development work. Had they been working on Version 5.3 of a legacy system that they expected to retire within a year or two, it wouldn't have been worth the effort to rebuild an accurate SRS. (But remember our first principle: Don't dig the hole you're in any deeper!)

Principle #4: Couple requirements development and testing.

We've all heard that a software project should begin developing test plans and test cases in parallel with requirements development, but I suspect this is still a rarity in actual practice.

It really is a good idea. Even the contemporary eXtreme Programming philosophy advocates "test before code." I've found that writing even high-level, conceptual test cases during requirements analysis reveals missing, ambiguous, and misunderstood requirements. When you work on the requirements for an enhancement, I heartily recommend you develop the corresponding test cases in parallel.

I came to appreciate the value of early requirements testing when I once asked our team's Unix scripting guru, Charlie, to code a simple email interface extension for our newly-installed commercial defect tracking system. I wrote about a dozen functional requirements for the email extension, and then I wrote twenty-odd test cases that described my vision of how the email function would operate. In the process of writing the test cases, I discovered an error in one of my requirements. A bit chagrined, I corrected the defective requirement before Charlie had completed his implementation, and when he delivered the script it was defect-free. The lesson here for me was to test my requirements *before* I give them to Charlie.

Use cases are an effective technique for both requirements and test case development. Ross Collard pointed this out in his article "Test Design" in *STQE*, July/August 1999:

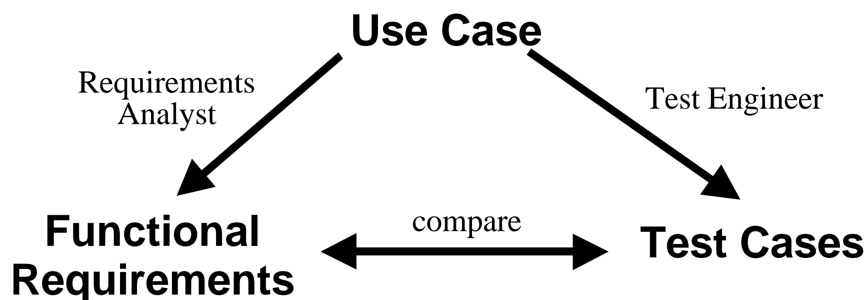
> "If the use cases for a system are complete, accurate, and clear, the process of deriving the test cases is straightforward. And if the use cases are not in good shape, the attempt to derive test cases will help to debug the use cases."

Copyright © 2001 by Karl E. Wiegers

I'm a strong proponent of use cases, and the first time I used them on a project I was struck by how easy it was to develop preliminary test cases from them. So if you're thinking from the *use case* perspective as you ponder enhancements in the next release, test cases are an obvious byproduct.

You can use these *test* cases to verify the functional requirements you develop from the *use* cases. This is a powerful verification method because you use different thought processes—and maybe even different team members—to generate functional requirements and test cases from a use case (see Figure 2).

Figure 2. Verifying functional requirements with conceptual test cases often reveals errors.



Check to see whether all of your functional requirements are covered by at least one test case—and whether all test cases could be executed with the current set of functional requirements. Any discrepancies between your test cases and your requirements reveal misunderstandings, ambiguities, or outright errors. Every time I do such systematic test thinking, I find errors in my requirements—and usually in my test cases, too. As you continue with implementation, you can expand the high-level test cases into detailed test procedures and add them to your test library.

Principle #5: Follow your change process.

Change control is the centerpiece of requirements management, so an effective change control process is an essential software maintenance tool. While organizations often implement such a process to handle bug reports, it will also help you manage other elements—enhancement requests, functionality modifications, and changes to requirements for new systems under development.

If you don't currently have a defined change control process, or if the actual practice doesn't agree with the "official" process, take the time to document a realistic and meaningful change control process. This will give you an effective process to use on your next maintenance activity or your next new development project. Other teams can then borrow your process to improve their own effectiveness and efficiency.

Copyright © 2001 by Karl E. Wiegers

A robust change process includes techniques for prioritizing approved changes against the backlog of other requests. It also includes a procedure for evaluating the likely impact of each change or enhancement on the rest of the system. This helps the decision makers make informed choices about how best to invest their limited resources. (If you don't presently have a change control process in place, you can download an example from www.processimpact.com/goodies.shtml.)

Part of change control is performing an *impact analysis*—a cause-effect examination that can help you avoid the quagmire in which a seemingly simple enhancement or change leads to an endless succession of system breakages and patches. Inadequate impact analysis is a significant underlying cause of misjudged scope. *Traceability* information can help greatly with impact analysis by revealing system components you might have to touch if you modify existing functionality. Impact analysis for a proposed enhancement should include several considerations:

- Identify the other system components you'll likely have to change. These might include other requirements, design descriptions, code, tests, user publications, help screens, system documentation, project plans, shared libraries, hardware, and even other subsystems or applications.
- Judge whether the change is technically feasible and can be accomplished at acceptable cost and risk. Will it conflict with other functions or overtax system resources such as processor capacity, memory, or communications bandwidth?
- Evaluate the possible impact on the system's performance, response time, efficiency, reliability, usability, integrity, and other critical quality attributes.
- Estimate the amount of work effort involved.

You might want to create some checklists of questions to ask yourself—questions to help do an adequate impact analysis of non-trivial enhancement requests. Start with the impact analysis checklists at http://www.processimpact.com/goodies.shtml. Then modify them, as you gain experience, to best meet the needs of your own projects. This is another area in which spending a small amount of time to get effective processes and tools in place during maintenance will pay big dividends when you *really* need them for the next big project to succeed.

It's important to remember that if you ask your project stakeholders to follow unfamiliar processes such as formal change control, the processes have to work. Beta test new processes before you commit the organization to follow them. As with software itself, beta testing or piloting a new process almost always reveals valuable adjustments to make before the process is ready for prime time. If your process doesn't work—if stakeholder change requests disappear into a vacuum with no apparent action or feedback—the participants will find ways to bypass the process. And perhaps they should.

Principle #6: Inspect down the traceability chain.

Requirements traceability involves defining logical links between individual functional requirements and other system elements—such as parent use cases or business rules and bits of design, source code, and other connected artifacts. Traceability links help you determine which components you'll have to change if you implement a change in a specific requirement.

Inspection is a powerful tool for ensuring that requirement changes are implemented fully and correctly. If you don't conduct some peer review of the new requirements and the

Copyright © 2001 by Karl E. Wiegers

corresponding changes made in affected work products, you're betting that you and other maintainers won't ever make a mistake. Based on my personal experience, that's not a very safe bet.

Requirements inspection should begin by ensuring that the new requirement is in scope, and that it's consistent with any pertinent business rules. Look for all the qualities we desire in an excellent requirement: *complete, consistent, correct, feasible, necessary, prioritized, unambiguous,* and *verifiable*. Requirements presented for making changes in an existing system can appropriately contain more implementation detail than is desirable when eliciting requirements for a green-field project. Those new requirements must fit into an existing reality of implementation and design constraints.

Next, make sure the requirement change has been correctly propagated into changes in the designs, code, test cases, user documentation, help screens, and any other affected work products. Requirements traceability ties in nicely here, because in order to inspect the appropriate items, you'll need to know which other system components each changed requirement affects. Odds are you don't have complete (or perhaps any) traceability information for your legacy system—but this is something else you can begin assembling whenever you make changes. It takes little effort to collect traceability links as you perform the work, but it's nearly impossible to regenerate from a completed system.

The inspection participants should represent three perspectives:

1. **The author of the work product being inspected** (that is, the new requirements for the modification or enhancement). Including the author's peers is helpful too; having another analyst or maintainer peruse the new requirements often reveals ambiguities, missing information, or poorly written requirements.

2. **The author of any predecessor work product or specification for the item being inspected**. For enhancements, consider having the requesting customer or an appropriate representative (perhaps from Marketing) participate in the inspection to ensure that the new requirements accurately describe the needed change.

3. **People who are responsible for downstream work products that are affected** by the modified requirements. You might include a developer, a tester, and a user documentation writer—all of whom will spot different kinds of problems. If you're making all the changes yourself, invite to the inspection some fresh participants who understand the work products well enough to find possible problems.

A possible fourth perspective includes individuals whose work products might be affected adversely by the change, or who will have to make corresponding changes in the components for which they're responsible. A knowledge of the interconnections between the components you're changing and the rest of the system is helpful here. Successful peer reviews also depend on having trained and knowledgeable reviewers available. A one-day training seminar on software technical reviews is usually enough to let the team members begin to effectively examine each other's work products for errors and improvement opportunities.

Principle #7: Start now.

Enhancement requests typically come with a sense of urgency attached. Customers need new functionality and they need it *now*. In that rushed atmosphere, it's hard to convince yourself

Copyright © 2001 by Karl E. Wiegers

to take the time to try a new technique, to write down the modified or new requirements, or to follow a seemingly bureaucratic change process.

So why go to the trouble? Because these practices let us work smarter instead of harder, ultimately leading to better products, more satisfied customers, and higher profits. But the perfect and easy time for doing the things you know you *should* do will never come.

Early in my software career, I set a personal goal of learning one or two new technical or managerial practices on each project. I forced myself to spend some time enriching my bag of software tricks, practicing these methods in a low-risk environment, so I was more confident in my abilities on the next project. Look for opportunities to apply requirements engineering techniques on every maintenance assignment, and climb a lot of small learning curves instead of a few big ones. Strive to improve what your team knows about its application portfolio each time you touch an existing system. The results are likely to include more robust and useful products and a more productive team.

Copyright © 2001 by Karl E. Wiegers