When Telepathy Won't Do:
Requirements Engineering Key Practices

Karl E. Wiegers

Process Impact

www.processimpact.com

The software industry is exhibiting an increasing interest in requirements engineering — that is, understanding what you intend to build before you're done building it. Despite the hype of "Internet time," companies across many business domains realize that time spent understanding the business problem is an excellent investment. Clients have told me they're getting serious about requirements because the pain of having built poor products has simply become too great.

You can best achieve requirements success by applying established good practices on your projects.[1,2] Figure 1 suggests a requirements development process framework with steps that incorporate the key practices described here. Thoughtfully tailor the practices to suit your project type, constraints, and organizational culture. Some highly exploratory or innovative projects can tolerate the excessive rework that results from informal requirements engineering. Most development efforts will benefit from a more deliberate and structured approach, though. Telepathy and clairvoyance rarely suffice.

A REQUIREMENTS ENGINEERING FRAMEWORK

Requirements engineering is primarily a communication, not technical, activity. Communication problems can begin early on if project participants have different ideas of exactly what "requirements" are. My favorite definition comes from Ian Sommerville and Pete Sawyer[1]:

> Requirements are…a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property or attribute. They may be a constraint on the development process of the system.

This definition points out that many different kinds of information fall in the domain of software requirements. A project needs to address three levels of requirements, which come from different sources at different project stages:

- *Business requirements* describe why the product is being built and identify the benefits both customers and the business will reap.
- *User requirements*, captured in the form of use cases, describe the tasks or business processes a user will be able to perform with the product.
- *Functional requirements* describe the specific system behaviors that must be implemented. The functional requirements are the traditional "shall" statements found in a software requirements specification (SRS).

1. Define the project's vision and scope.
2. Identify user classes.
3. Identify appropriate representatives from the user classes.
4. Identify the requirements decision-makers and their decision-making process.
5. Select the elicitation techniques that you will use.
6. Apply the elicitation techniques to develop and prioritize the use cases for a portion of the system.
7. Gather information about quality characteristics and nonfunctional requirements from users.
8. Elaborate the use cases into the necessary functional requirements and document them.
9. Review the use case descriptions and the functional requirements.
10. Develop analysis models if needed to clarify the elicitation participants' understanding of portions of the requirements.
11. Develop and evaluate user interface prototypes for portions of the requirements that are not clearly understood.

12. Develop conceptual test cases from the use cases.
13. Use the test cases to verify the quality of the use cases, functional requirements, analysis models, and prototypes.
14. Repeat steps 6 through 14 for the other portions of the system until the team concludes that requirements elicitation is as complete as it needs to be.
15. Baseline the documented requirements.

Figure 1. A suggested requirements development process.

A lack of agreement over what to call the entire software requirements field can lead to further confusion. I split the domain of *requirements engineering* into *requirements development* and *requirements management* (Figure 2). Requirements development is further subdivided into *elicitation*, *analysis, specification,* and *verification*.[3] Each of the requirements engineering key practices described below fits into one of these subdisciplines. The deliverable from requirements development is a baseline that constitutes an agreement among key project stakeholders as to the new product's capabilities. During requirements management, the project controls changes in the requirements baseline and monitors requirements implementation.

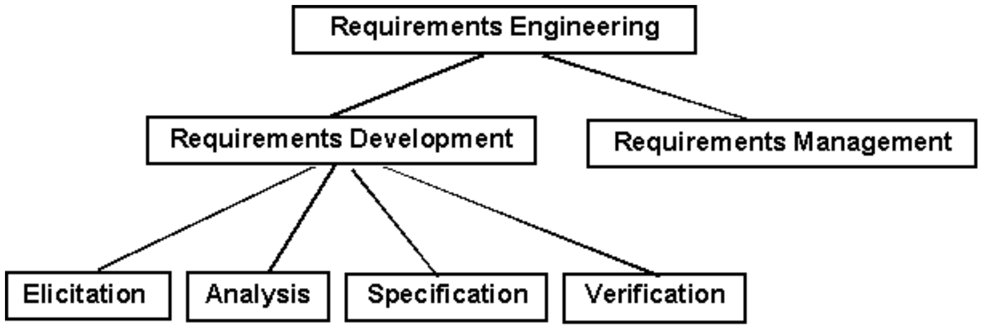

Figure 2. Subdisciplines of requirements engineering.

ELICITATION KEY PRACTICES

Software product development must begin by gathering the different kinds of requirements from suitable stakeholders. The critical first step is to...

Define the Product's Business Requirements

Surveys at my seminars suggest that nearly everyone has worked on projects that suffered from scope creep, yet few of these projects had a documented scope or vision. How do you even know that the scope is creeping if no one ever clearly defined it? Documented business requirements will help you answer the first question to ask whenever someone proposes new functionality: "Is this in scope?"

The business requirements should articulate how the product's developers and their customers will benefit from this product. A short vision statement describes what the product could ultimately become. However, you probably won't implement the grand product vision in a single release. Therefore, you should include the project's scope and limitations with the business requirements. The scope description should summarize the major features included in the initial release and describe how you will more fully realize the vision through subsequent releases. The limitations identify specific capabilities the product will *not* include. A vision and scope document template is available at www.processimpact.com/goodies.shtml. Once you've defined your project scope, be sure to...

Get Extensive User Involvement

Multiple studies indict insufficient user involvement as a common reason why software projects struggle and fail (for example, see http://www.scs.carleton.ca/~beau/PM/Standish-Report.html). Every project should identify its distinct user classes and their characteristics. Users might differ in their frequency of product use, features used, privilege levels, or skill levels. Determine which of your user classes will carry the most weight in priority discussions, when resolving conflicting feature requests, and in driving design choices.[4]

Next, find suitable representatives who can serve as the voice of the customer for each important user class. I have used a "product champion" model to engage user representatives in the development process.[2] Product champions should be actual users of the new product, not surrogates such as user managers, funding sponsors, marketing staff, or developers. Document your general expectations of the product champions to serve as a starting point for negotiating each champion's responsibilities. Typical responsibilities include:

- Developing usage scenarios and use cases
- Resolving conflicts between proposed requirements
- Defining implementation priorities
- Specifying quality attributes
- Inspecting requirements documents
- Evaluating and prioritizing enhancement and change requests.

The product champion approach works well for internal systems development where users are readily accessible. Commercial product development often demands more creative approaches for optimizing customer involvement. One company that developed a large point-of-sale and back-office system hired several store managers to serve as full-time product champions. Focus groups can serve as a sounding board for feature ideas but they generally don't have the decision-making authority that product champions have. Nearly everyone I know who has tried the product champion approach found it to be highly effective. Make sure you select the right champions, that they take their responsibilities seriously, and that they're empowered to make requirements decisions.

Once you've engaged suitable customer representatives, your elicitation discussions should...

Focus on User Tasks

The use case approach is all the rage in software requirements circles these days and is one fad I endorse. A use case describes a task the user must be able to perform with a software product. Use cases shift requirements discussions from the traditional focus on features and functionality to the perspective of what the user will do with the product. This change in perspective is more important than whether you draw elaborate use case models.

The use case approach helps you avoid missing essential functional requirements or implementing cool functionality that no one uses because it isn't directly related to specific tasks. They can also help you identify exception conditions the system must handle, which sometimes are overlooked in the focus on expected system behaviors. I have found use cases valuable for deriving conceptual system test cases very early in the development process.[5] The test cases almost immediately revealed errors in my functional requirements and analysis models.

User representatives can generally describe their business processes and itemize the tasks a new application must let them perform. Skillful facilitation keeps elicitation workshops focused on user tasks, rather than drifting into discussions of functional requirements or detailed user interface design. The analyst should first learn how a user imagines interacting with the system to accomplish each task. That understanding lets the analyst derive the necessary functional requirements. But in addition to specifying the desired system functionality, make sure you...

Define Quality Attributes

I once worked with a project that had excellent user involvement and created a high-quality, detailed SRS. However, the delivered system failed to satisfy the users' unstated performance and efficiency expectations. Consequently, the system met with poor user acceptance, despite fully meeting their functional needs. Elicitation generally focuses on the functional requirements, which describe what the system will do or let the user do. It's equally important to understand *how well* the system will perform its functions.

Quality attributes (sometimes collectively known as the "ilities") include characteristics that users can observe as well as those that are primarily important to developers.[2,6] Users might care immensely about system availability, efficiency, integrity, reliability, robustness, and usability. The technical staff should also be concerned about such characteristics as flexibility, maintainability, portability, and reusability. You'll likely have to make trade offs to achieve the optimum balance of quality attributes; you can't create the best of all possible worlds.

Users rarely present such nonfunctional requirements spontaneously, other than to demand that the system be "user-friendly" (no one has ever requested a user-surly application). "User-friendly" is too subjective and multi-faceted to let stand, so you must explore the characteristics that would make an application appear user-friendly to its various users. Consider asking about desirable attributes before diving into functionality discussions. This can reveal critical design objectives that a functionality-first approach might miss. Unless you ask questions to understand the users' implicit quality expectations, only luck will let you build a product that satisfies those expectations.

ANALYSIS KEY PRACTICES

Requirements analysis includes decomposing high-level requirements into detailed functional requirements, constructing graphical requirements models, and building prototypes. Analysis models and prototypes provide alternative views of the requirements, which often reveal errors and conflicts that are hard to spot in a textual SRS. Another important analysis activity is to...

Prioritize Requirements

Any project with resource limitations must establish the relative priorities of the requested features, use cases, or functional requirements. Prioritization helps the project manager plan for staged releases, make trade-off decisions, and respond to requests for adding more functionality. It can also help you avoid the traumatic "rapid descoping phase" late in the project, when you start throwing features overboard to get a product out the door on time.

Prioritization often becomes politically and emotionally charged, with all stakeholders insisting that their needs are most important. A better approach is to base priorities on some objective analysis of how to deliver the maximum product value within the schedule, budget, and staff constraints. Many organizations classify requirements into "musts" and "wants," or they use three priority classifications (they're high, medium, and low, no matter what labels are used). But priorities really span a spectrum, rather than fitting tidily into just a few buckets.

A more analytical approach is to rate the relative customer value and the relative cost and technical risk of each feature. Customer value considers both the benefit if a feature is present and the penalty if it is not. The most desirable features should be those that provide the greatest value at the lowest risk-adjusted cost. A spreadsheet that implements this prioritization scheme is available at www.processimpact.com/goodies.shtml.

SPECIFICATION KEY PRACTICES

The most essential (yet often neglected) specification key practice is to write down the requirements in some accepted, structured format as you gather and analyze them. The objective of requirements development is to communicate a shared understanding of the new product among all project stakeholders. Historically, this understanding is captured in the form of a textual SRS written in natural language, augmented by appropriate analysis models. I don't expect to see formal specification languages in widespread use for decades to come, so natural language will continue to be the medium of choice, despite its ambiguities and shortcomings. However, you can overcome the limitations of storing requirements in a textual document if you...

Store Requirements in a Requirements Management Tool

Commercial requirements management tools let you store requirements and related information in a multi-user database. These products let you manipulate the database contents, import and export requirements, and connect requirements to objects stored in testing, design, and project management tools.[7] You can define attributes for each requirement, such as its version number, author, status, origin or rationale, allocated release, and priority. Traceability links between individual requirements and other system elements help you evaluate the impact of changing or deleting a requirement. Web access permits real-time sharing of database updates with members of geographically distributed teams.

Table 1 lists several widely used requirements management tools. These are specifically requirements management—not requirements development—tools. They won't help you scope your project, gather the correct requirements from the appropriate user classes, or write good requirements. However, they'll give you much more control over your requirements collection than you can achieve with traditional SRS documents.

Table 1. Major Requirements Management Tools

Tool	Vendor
Caliber-RM	Technology Builders, Inc.; www.tbi.com
DOORS	Quality Systems and Software, Inc.; www.qssinc.com
RequisitePro	Rational Software Corporation; www.rational.com

RTM Workshop	Integrated Chipware, Inc.; www.chipware.com
Vital Link	Compliance Automation, Inc.; www.complianceautomation.com

VERIFICATION KEY PRACTICES

Verification involves evaluating the correctness and completeness of the requirements, to ensure that a system built to those requirements will satisfy the users' needs and expectations. The goal of verification is to ensure that the requirements provide an adequate basis to proceed with design, construction, and testing. A powerful way to achieve this goal is to...

Inspect Requirements Specifications

Because it costs so much more to fix defects later in the development process, formal inspection of requirements is perhaps the highest leverage software quality practice available. I know of one company that has measured a return on investment from SRS inspections of ten to one. My colleagues who have successfully implemented requirements inspections find them to be tedious, slow, painful, and worth every minute because so many defects are found so cheaply. Combining formal inspection with incremental informal requirements reviews provides a powerful approach to building quality into your product.

REQUIREMENTS MANAGEMENT KEY PRACTICES

Requirements management activities include evaluating the impact of proposed changes, tracing individual requirements to downstream work products, and tracking requirements status during development. You can monitor project status by knowing what percentage of the allocated requirements have been implemented and verified, just implemented, or not yet fully implemented. But the heart of requirements management is to...

Manage Requirements Changes

Every project should document a process that describes how a proposed change will be submitted, evaluated, decided upon, and incorporated into the requirements baseline. You can support the change control process with a problem- or issue-tracking tool, but a tool is not a substitute for a documented process. Every project should also establish a change control board of the decision-makers who approve or reject each proposed change. This board could be just one person on a small project. More typically, a body representing diverse perspectives (such as development, management, customer, and testing) is appropriate. A requirements management tool can help you manage the changes made to many individual requirements, maintain revision histories, and communicate changes to those affected by them.

PUTTING THE PRACTICES INTO PRACTICE

It's easy to rave about how wonderful life will be if you apply all of these great practices. The hard part is incorporating new techniques into the way your organization routinely operates. The grass-will-be-greener argument motivates some people to change the way they work, but observing that the grass is on fire right behind you is even more persuasive. Process changes should be motivated by the pain of schedule slippages, overtime, rework, high repair costs, and customer dissatisfaction that you've experienced before. The improvement-driven organization will examine the sources of such pain and avoid repeating the same problems.

Organizations that try to implement improved requirements practices encounter numerous obstacles. Emphasizing requirements might not sit well with customers or managers who view code as the only tangible sign of progress. It takes time to learn about, try out, and incorporate new ways of working, yet few people have patience for the inescapable learning curve. Intuitively, putting more time into the front end of a project will delay delivery by that same duration, but this doesn't consider the potentially high return on investment from better requirements engineering.[8] It's impossible to predict the exact ROI a specific project can expect. However, correcting a requirement defect reported by a customer can easily cost 100 times more than addressing that error during requirements development.

To help build your case for improved requirements practices, identify problems from previous projects that you can attribute to requirements weaknesses. Estimate the costs of those problems as a way to justify the investment in new techniques or tools. Begin by implementing those new practices that are not especially difficult or time-consuming to adopt, but which have a high potential benefit—the low-hanging fruit.[2] Obtain management and project leader commitment to trying the new practices. Train the business analysts, developers, managers, and key customers in requirements principles and practices. Such training can provide a common vocabulary and understanding that reduces the barrier to trying new techniques. Avoid the temptation to swallow any new methodology whole. Instead, selectively incorporate individual practices into your elicitation, analysis, specification, verification, and management activities.

Pilot new practices in low-risk situations to learn how best to make them work for you before you incorporate them into your official requirements process. Remember that no process is a substitute for common sense and thoughtful problem solving. If you are sure that some of these practices don't apply to your project, don't force-fit them.

Despite your best attempts to persuade your colleagues that improved requirements practices are valuable, you might encounter people I call flat-earthers. Just as there are people who still believe our planet is flat, there are skeptics who refuse to be convinced that better requirements processes are worth the price. None of these approaches will succeed unless your organizational culture includes a shared commitment to building high-quality software products in a disciplined way. I've seen all the practices described here used to good effect on real projects. I'm convinced they're worth a try on your projects, too.

References

1. Ian Sommerville and Pete Sawyer, *Requirements Engineering: A Good Practice Guide* (Wiley, 1997).
2. Karl E. Wiegers, *Software Requirements* (Microsoft Press, 1999).
3. Richard H. Thayer and Merlin Dorfman, *Software Requirements Engineering,* 2d ed. (IEEE Computer Society Press, 1997).
4. Donald C. Gause and Brian Lawrence, "User-Driven Design," *Software Testing & Quality Engineering*, Vol. 1, No. 1 (January/February 1999), pp. 22-28.
5. Ross Collard, "Test Design," *Software Testing and Quality Engineering*, Vol. 1, No. 4 (July/August 1999), pp. 30-37.
6. Peter DeGrace and Leslie Hulet Stahl, *The Olduvai Imperative: CASE and the State of Software Engineering Practice* (Yourdon Press/Prentice-Hall, 1993).
7. Karl Wiegers, "Automating Requirements Management," *Software Development*, Vol. 7, No. 7 (July 1999), pp. S1-S5.
8. Dean Leffingwell, "Calculating the Return on Investment from More Effective Requirements Management," *American Programmer*, Vol. 10, No. 4 (April 1997), pp. 13–16.

[This paper was originally published in *Cutter IT Journal*, May 2000. It is reprinted (with modifications) with permission from Cutter Information Corp.]