Requirements Engineering for Software Engineers

Aaron Sher, Vanteon Corporation

7/12/12

Introduction

Often, software requirements are produced by marketing departments or product managers. These documents are handed down from on high in their final form, with no input from the engineers who will have to satisfy the requirements. Worse, as every engineer is aware, every non-trivial requirements document is incomplete and sometimes inaccurate, no matter how much effort and review time has been put into it.

While I can’t offer a complete solution for this problem, this white paper will present an overview of the discipline of *requirements engineering* in order to help engineers to produce workable, complete, and consistent requirements. Requirements engineering is often viewed as a euphemism along the lines of “sanitation engineering” – something to be done by non-engineers who want to feel like they’re doing real engineering work. While this view may be accurate for the vast majority of requirements, it is possible to apply engineering rigor to the process of creating requirements, and in so doing, to greatly improve their quality (and the lives of the engineers who have to implement them!)

High Level Description

The overall process of requirements development consists of four phases: elicitation, analysis, specification, and validation.

- *Elicitation* is the process of getting the stakeholders (marketing, product managers, customers, etc.) to tell you what they want.
- *Analysis* is the process of boiling this unstructured list of requests down into a coherent and consistent structure.
- *Specification* is the process of writing the fully-analyzed requirements down in formal language so that they are clear and unambiguous.
- *Validation* is the process of getting approval for the analyzed and specified requirements from the stakeholders.

This white paper is primarily concerned with analysis and specification. Elicitation is a science unto itself, involving interview techniques, observational studies, questionnaire design, and so forth, but this is outside the scope of this paper.

One thing I should point out here is that this is the heavyweight big-process version of requirements. Many, probably most, projects won't benefit at all from having this much work put into requirements; most of the time, you're better off writing code and refining the details after you have something to demonstrate. However, sometimes you have no choice, because the process is imposed by regulations or higher management, and if you're going to do it, you might as well do it right.

Analysis

You have a hundred pages of requests from a dozen different stakeholders. They are described at all levels of detail, refer to every possible piece of the system, are frequently contradictory, and occasionally impossible. How do you approach the task of organizing this information?

First, understand that the approach described here is by no means the One True Way. There are an infinite number of ways to tackle this problem – this approach is one that has worked in the past for the author, and one that appeals to engineers because it is logical and results in something that lends itself well to implementation.

Determine the Overall Structure

You can think of requirements analysis as being like the design phase of a large software project. You're not trying to nail down every detail, you're mostly trying to shake your ideas down into a consistent and logical structure. Once you've done that, you can fill in just enough to be sure that it's going to work, and then get to the coding. In the context of requirements, the idea is similar – you don't need to get every requirement all the way down to the formal language (that's for specification), you just need to make sure that your requirements cover the whole system and don't conflict with one another.

The approach is as follows:

1. Create a structural decomposition of the proposed system. That is, break it into major subsystems, and break those into minor subsystems. Continue this process until the structure starts to get fuzzy in your mind – this will depend on how well-defined it is already. It may be that all you can do is to roughly divide it into two or three components, or you might be able to go three or four levels deep. Note that this is the same thing that you'd need to do if you were really designing a software system.

2. Go through each of the requirements and assign it to the appropriate subsystem in your decomposition. For requirements that span multiple subsystems, they should be assigned to the parent of those subsystems. This is called the *scope* of the requirement.

In addition to classifying your requirements by their scope, it is often helpful to classify each one by some other attributes as well. The most common is by type:

- *Business requirements* describe the business case for the product; that is, why is it being built, and what benefits will accrue to each of the stakeholders?
- *Functional requirements* are the ones most people think of first. They describe what the product does, or how it does it. Functional requirements are normally visible to a user of the product.
- *Nonfunctional requirements* describe cross-cutting restrictions on the product's functions, such as performance requirements ("all operations must complete or provide feedback to the user within 500ms"), regulatory requirements ("the system design must comply with cryptographic export restrictions"), or safety requirements ("any operation that affects cabin pressurization must be verified by a suite of automated unit tests").

The advantage of classifying by type is that you can treat the different types differently. Business requirements don't have to be complete or rigorous, and often won't apply below the level of the entire system. Nonfunctional requirements will frequently also apply only to the entire system, and are normally driven by external factors. Functional requirements are the ones that require the most attention; they must describe each section of the system independently and completely, or else you have behavior that isn't covered.

In order to resolve conflicts between requirements, it is often helpful to classify them along other axes as well, such as priority, risk, level of effort, and so forth. These categories are not as helpful in actually describing the system, but they can be useful in determining which of several conflicting requirements to keep, or where to look when you want to reduce the scope of the system for a first release.

The output of this effort will be a structural tree where each subsystem is associated with several lists of requirements (one for each requirements type, some of which may be empty). Often, all of the business requirements will apply to the entire system; this is fine, as these requirements are largely not your problem in any case. Give them their own section in the final document and don't worry about them otherwise.

Look for Big Problems

Now that you have your overall structure in place, it's time to clean up your "design" and make sure that it actually describes the system. As a first pass, look for obvious problems:

- If you have any subsystems with few or no functional requirements whatsoever, you need to go back to elicitation and find out what those subsystems ought to be doing. Maybe they shouldn't exist at all, or maybe they should be folded into other subsystems?

- If you have requirements that are at a very high level of the tree not because they apply to the entire system, but instead because they apply to widely-separated subsystems, consider splitting those requirements into separate statements that can be applied specifically to the subsystems in question. Alternatively, should these requirements actually apply to a wider scope than they currently do? This happens often for nonfunctional requirements.
- If you have requirements on the same subsystem that directly contradict each other, or requirements that contradict those at a higher level, you need to resolve this discrepancy. Usually, that will mean going back to the stakeholders to get an answer for how the system truly ought to behave; sometimes, this will involve negotiations between different stakeholders who have conflicting needs.
- Requirements that are clearly unfeasible, due to underlying technical issues, budgetary or calendar restrictions, or for any other reason, should be immediately pushed back to the stakeholders.

Analyze Each Subsystem Independently

Once the big issues have been dealt with, you can systematically go through each subsystem and analyze its requirements. The main concept here is **externally-visible behavior**. The requirements on a subsystem can **only** describe those aspects of its behavior that are visible to other subsystems. If a requirement seems to describe something inside the subsystem, then it needs to be moved down in the structural tree, to a sub-subsystem. If it doesn't make sense to split up the subsystem you're describing, then the requirement is invalid.

This is an absolutely critical idea, and it's worth repeating. Requirements must describe a subsystem in terms of how it can be observed to behave from outside. Think of this like a library interface – you want the interface to encapsulate the implementation details of the library, so it needs to be described as abstractly as possible. Similarly, the requirements of a subsystem can't legitimately constrain its implementation details, and if they're trying to do that, something's wrong. Ask yourself why the requirement is trying to limit the implementation of the subsystem, and you'll probably discover that what you really need is a (possibly nonfunctional) requirement stating that the system must have whatever characteristics the original requirement was trying to enforce.

There is an example of this in the case study below, but it's worth providing a smaller example here in order to clarify the concept. Imagine that you are analyzing the requirements for an HTTP support library to be linked into a Windows application. This library will need some background threads to handle asynchronous operations, and the requirement is that the library will use pthreads rather than Windows threading APIs. This is an implementation detail that is not visible to clients of the library, and thus is not a valid requirement.

You, the requirements engineer, go back to the stakeholder who created this requirement, and discover that the reasoning behind it is that a future version of the system is expected to run on Linux and they were hoping that the library would be more portable if it used pthreads. You remove the original requirement, and replace it with a nonfunctional requirement that states “The HTTP support library shall be implemented in such a way as to allow future portability to Linux unless this approach significantly increases the implementation effort of the Windows version.” This statement gets to the actual intent of the original request, without putting arbitrary restrictions on the implementers.

By limiting requirements to describing the externally-visible interface of a subsystem, they become a sort of “contract” in the same way as a library interface layer. This makes it a great deal easier to determine whether the requirements have fully described the behavior of the subsystem needed by its “clients”.

Specification

If analysis is like design, then specification is like coding. Just like code, it’s important that requirements be phrased precisely and unambiguously. Also just like code, there is a specific language that you can use to make sure that the reader understands what you’re saying. This is called *formal requirements language*. The exact language used varies by organization, but is normally some variation of the terminology specified in RFC 2119 [1].

The purpose of formal language is to avoid ambiguity. “For” and “while” don’t mean the same thing in code, and if you try to replace one with the other you won’t get the result you wanted. Similarly, “shall”, “may”, “should”, and “will” mean different things in requirements, and it’s important to understand the distinctions if you want to be able to communicate clearly.

Terminology

RFC 2119 [1] provides definitions for the following terms:

Terms	**Meaning**
must (not), required, shall (not)	These terms mean that the statement is an absolute requirement. Most requirements are written this way, e.g., “The system *shall* detect deviations of more than 3% and notify the user within 1 second.” Note: sometimes “must” is used to express an assumption external to the requirements, e.g. “The system *shall* transmit data at a minimum rate of 1 Mb/sec. The network infrastructure *must* be capable of supporting this rate.” This usage should be clear from context, but you should be consistent about using “must” only one way or the other; don’t try to mix them in a

	single document or you might confuse your readers.
should (not), recommended	The statement is a strong recommendation, but there might be specific circumstances under which it is valid for the system to ignore it. For example, “The user interface *should* display properly on a screen with a resolution of 1024 x 768 pixels.” Possibly there are certain dialogs whose design would have to be greatly compromised to fit into this small resolution, and which most users will never need to access; in such a case, it might make sense for those specific dialogs to exceed the given screen size.
may, optional	The statement describes an optional behavior or characteristic, or explicitly permits some behavior. For example, “The application *shall* not occupy more than 50MB of working memory, but *may* use hard drive space to expand its working set of data if necessary.” Note that given this definition, the term “may not” is likely to be confusing. Strictly speaking, saying that the system “may not” do something is equivalent to saying “it is permissible for the system to not do this”, but most English-speakers are likely to read this as “must not”. Avoid this phrasing if you can.

In addition to the terms from [1], there is one additional term that is commonly used:

Terms	**Meaning**
will	This expresses something to happen in the future. It is *not* a requirement. For example, “The system *may* discard cached data if memory becomes full; however, the performance of subsequent operations *will* suffer in this case.”

Controlling Ambiguity

The terminology is the easy part. That’s equivalent to learning the syntax of your programming language, something that you’ve presumably done long before you started working on your project. The hard part is making each requirement clear, correct and unambiguous.

For example, here’s a typical user request that you might get from elicitation: “The old system was too slow in a lot of places, the new system needs to be faster.” That’s not going to be a useful direction for the engineers who are trying to develop the system, so how do you go about turning that into useful requirements?

First, you need to figure out what “a lot of places” is referring to. You’ve presumably already done that in analysis, since you assigned the requirement to some section of your decomposition tree. You can now rewrite the requirement as “Features X, Y, and Z need to be faster than they were in the previous version of the system.” Better, but still not particularly useful. How much faster is faster?

You need to figure out how fast the features need to be in order to make the users happy. This might require a trip back to elicitation, or you might be able to figure it out from existing standards or just from gut feel. Keep in mind that the requirement might be different for each feature, so now you’ve got a collection of requirements of the form “Feature X needs to complete in ### milliseconds.”

Now, you need to make sure your terms are defined. What, exactly, does “complete” mean? A good reformulation might be “After invoking feature X, the user needs to regain control within ### milliseconds.”

Finally, you’ve got an unambiguous requirement (actually, a set of them, potentially one for each feature). Now you can rephrase it in formal language: “After invoking feature X, the system *shall* return control to the user within ### milliseconds.”

One additional complication – can this be true in all cases? Are there any exceptions? What if the system is heavily loaded, or out of disk space, or somebody kicked the network cable? You may need to give the engineers an escape clause: “After invoking feature X, the system *shall* return control to the user within ### milliseconds. If Y occurs and this is not possible, the system *shall* present the user with a visual progress indication for no more than ### seconds. By the end of this period, the system *shall* either complete the operation successfully or display an error message to the user.”

Other Notes

Some other miscellaneous notes about specifying requirements:

- Try to limit yourself to no more than one requirement per sentence. This will make your life easier later if you need to remove requirements, or you need to tag them for a requirements management system.

- Be consistent with your sentence structure. I prefer to phrase requirements in the form “The system *shall* do X”, but it works as well to say “The user *shall* be able to do X”, as long as you don’t switch back and forth. Formal requirements documentation should *not* read like casual speech – your requirements document isn’t intended to be read cover-to-cover, so it doesn’t have to be a page-turner.

- If you are recording your requirements in a document as opposed to a requirements management system, make sure that the structure of your document mirrors the structural decomposition tree you put together during analysis. This will make it easy to organize your requirements in the document, and ensure that you don't have to put a single requirement in more than one place.

- In practice, analysis and specification usually happen in parallel; as you analyze each group of requirements, you capture the result in formal language, so that by the time you're done with analysis you have the overall set of requirements fairly well specified.

Example

Here's a more fully-fleshed-out example of the entire process. Let's assume that you're working on requirements for a minimal text editor. For the sake of simplicity, we'll assume that the system can import and export various file types, perform basic text editing functions, and has a spell-checker. Here's a few of the requests you might get from elicitation:

- Needs to import common text and document formats, and export DOC, RTF, TXT, and PDF
- Must handle big documents smoothly
- May not repaginate the document unless the user has actually changed something
- Must be able to display international character sets
- Error messages must be written for a non-technical user
- Spell-checker must be comparable to the one in Microsoft Word

Overall Structure

Note that these requests are at multiple levels of detail and are frequently ambiguous or incomplete. The first step of analysis is to build your functional decomposition tree so that you can start classifying your requirements by scope. For this very simple system, the tree might look like this:

1. Entire system
    1.1. Text viewing
    1.2. Text editing
    1.3. Document importing
    1.4. Document exporting
    1.5. Spell checking

Now you can go through the requests and tie each one to a section:

Needs to import common text and document formats, and export DOC, RTF, TXT, and PDF

The scope tree makes it clear that this is at least two requirements, one for import and one for export. On the export side you can easily split it into four different requirements, one for each format; on the import side it's more difficult, since they didn't provide a list of formats. More research is needed here.

Must handle big documents smoothly

The first question is, *which piece* of the system needs to handle big documents? It's probably not the import/export sections that are the first priority, since the user isn't going to be too surprised if a 2GB document takes a bit to import. This can reasonably be assigned to 1.1, 1.2, and 1.5, but that requires that we split it up. Is that a problem? Well, consider what "smoothly" might mean to each section. For viewing, it probably means that we need to display and scroll through a large document without noticeable lag (that term to be defined later). For editing, it's less clear what this might mean – maybe we should leave this out entirely, since if we can't define it, the engineers won't be able to implement it. For spell checking, this is a straightforward performance requirement along the lines of "the system shall be able to check spelling in a document in no more than XX ms per page", though we'll need to figure out the value of that constant, and maybe "per page" isn't the best metric.

May not repaginate the document unless the user has actually changed something

Think somebody's annoyed about their current system? This directly applies only to 1.1, though there are implications on 1.2 as well.

Must be able to display international character sets

As written this clearly applies to 1.1 only, but there are implications for other subsystems. For example, if we're displaying international characters (which we're going to read as "the full Unicode character set"), we're going to need to be able to enter them on the keyboard, select them with the mouse, import them from other document formats, preserve them on export, etc. This one had probably better apply to the overall system until we've broken it down some.

Error messages must be written for a non-technical user

This is a straightforward nonfunctional requirement that applies to the entire system.

Spell-checker must be comparable to the one in Microsoft Word

This should raise big red flags for any engineer. "Comparable" in what way? Performance, accuracy, compatibility, user interface? This one has to go back to elicitation before we can do anything real with it, but at least for now we can just assign it to section 1.5 and stick a warning label on it for later.

High-Level Analysis

Now we can go through each of the functional sections and look for big problems. Since we're working only with a subset of the elicitation requests here, we won't consider subsystems that are lacking complete requirements.

- This small list doesn't contain anything that directly contradicts itself, as far as we can tell, though there are potential issues with the interactions between international character sets and document export (some formats might not support international characters well or at all), and there are certainly issues with spell-check (Are we going to have dictionaries for every possible language? How do we know what language they're using just from the character set?) These need to be pushed back to the stakeholders for clarification as they are discovered.

- We've already encountered one requirement that needed to be split up (the "big documents" request) across different subsystems, and the "international character sets" requirement likely needs to be split as well.

- It's hard to tell if any of these rather vague requests are technically infeasible, but we'll have to keep in mind the possibility that the requesters don't understand how difficult it might be to, for example, import a PDF file, or spell-check Asian languages.

Detailed Analysis

We're not going to look at the entire system here, since even with this very small set of requirements that would be too big for a white paper, so let's consider only section 1.1 (text viewing) as an example. According to our breakdown, we have three requests that apply to this section:

- Must handle big documents smoothly
- May not repaginate the document unless the user has actually changed something
- Must be able to display international character sets

Let's consider each of these, and try to break them down to individual requirement statements, though we're not worrying about formal language yet.

Must handle big documents smoothly

In order to break this down, we first have to define "big". Imagine that we've spoken to the stakeholders, and they said that "big" means on the order of 1000 pages. "Pages" isn't a very good metric for software most of the time, so we'll make some assumptions and do a bit of back-of-the-napkin math and decide that this means on the order of 4 million characters. We'll leave a margin for safety and write the requirements to 5 million characters, like so:

- Must be able to display documents of at least 5 million characters, assuming that enough memory exists to store the data.

- Must be able to scroll from one end to the other without significant lag (needs definition).

- On systems with insufficient free memory, the system will swap the document from disk; this will cause lag, but will allow the user to continue working.

May not repaginate the document unless the user has actually changed something

This is problematic. The issue here is that repagination is not, inherently, an externally-visible behavior. What the stakeholder is really saying here is something like, "I don't want the system to lock up for a long time in order to repaginate my huge document unless it really needs to." But of course, this is just an example - if the system were locking up for some other reason, they would be equally irritated. We'll have to state the requirement as something like the following:

- Must respond to all user inputs within XX ms unless a long-running operation is in progress, in which case it must inform the user of the fact.
- Long-running operations must be kept to an absolute minimum.

Must be able to display international character sets

The basic requirement here is very straightforward. This one starts to get complex in the other subsystems, but from a text viewing perspective we can restate this as simply:

- Must be able to display the full Unicode character set.

Note that we didn't say "the system will store its documents in memory using Unicode" or anything like that; we simply said that, given a document containing Unicode characters, they should get onto the screen somehow. This is an externally-visible behavior, while the others are not.

Specification

Restating the requirements in formal language and eliminating some leftover ambiguity, we get:

- The system shall be able to display documents of at least 5 million characters, assuming that enough memory exists to store the data.
- The system shall able to scroll from the beginning to the end of a 5 million character document using the page-down command without more than 0.5 seconds of lag at each step.
- If insufficient free memory exists to store the document, the system shall swap portions of the document to disk as needed. This operation shall take no more than 3 seconds. The system shall display a visual indication to the user as this operation occurs.
- The system shall respond to all user inputs within 200 ms unless a long-running operation (more than 200 ms) is in progress. When any long-running operation is in progress, the system shall display a visual indication to the user. The system shall not perform any long-running operation except as a direct response to a user action.
- The system shall have the capability to display any characters from the full Unicode character set.

Note that we started with three stakeholder requests, and ended up with nine requirements statements. This is common - in fact, this example is a bit simplistic, as a single request might easily end up as dozens or occasionally hundreds of separate requirements statements in a real system.

Validation

Once we have our requirements fully specified, we need to give them back to the stakeholders to make sure that we've correctly captured their intent. Look again at what we did above, and count the decisions we had to make in order to translate the original request to the final formal specification. For the single "big documents" request, we had to determine what "big" was (yes, we got stakeholders' input on that one, but different stakeholders might disagree on the page count, and we still had to make assumptions about characters per page), we added some caveats about what the system would do if there wasn't enough memory available, and how long everything could take before it wasn't perceived as "smooth". All of those decisions need to be approved before we can say that we've truly captured the requirements.

Conclusion

Requirements engineering is a real engineering discipline. In fact, it bears a striking similarity to actual software development, and the skills of software development transfer readily to those of requirements engineering. Non-engineers have great difficulty creating good software requirements, as a certain amount of knowledge is required in order to determine whether requirements truly describe externally-visible behaviors of a subsystem and whether they are feasible to implement.

Hopefully, this paper will give software engineers who have resisted doing requirements work, and lived with the consequences, an idea of how to approach the problem. For projects that need full, high-process requirements documents, improving the quality of those requirements will help everybody get better products with less pain.

References

[1] http://www.ietf.org/rfc/rfc2119.txt

[2] http://facweb.cs.depaul.edu/jhuang/se681/CSDP_RE.pdf

[3] http://kaiya.cs.shinshu-u.ac.jp/re/lab/p35-nuseibeh.pdf