Requirements Development

Don Moreaux and Troy Pearse
Hewlett-Packard Co.

The Requirements Development process supplies requirements to the Technical Solution, where the requirements are converted into the product architecture, product component design, and the product component itself (for example, coding, fabrication). This information is fed to Product Integration, where product components are combined and interfaces are assured to meet the interface requirements supplied by Requirements Development.

Steps of Requirements Development are simply:

1. Capture the requirements, using the **FURPS+**. The concept is briefly described in the appendix. The letters in **FURPS+** stands for: **F**unctionality, **U**sability, **R**eliability, **P**erformance, and **S**upportability. The **+** indicates that there may be product or organization-specific factors that are also important. Many organizations are particularly concerned about product Localization.

   Analyze, distill and record the requirements, applying the **SMART** concepts described in the appendix. **SMART** stands for **S**pecific, **M**easurable, **A**ttainable, **R**ealizable, and **T**ime bounded.
2. Review the resulting Requirements Specification(s), both with the Project Team for completeness and understanding, and with the client to ensure accuracy and validity.

Outputs are the Functional Requirements Specification, potentially the Non-functional Requirements Specification, and the Requirements Traceability Matrix.

When the Requirements Development process is completed, client requirements are represented in a Requirements Specification, listing and describing all of the requirements in detail. Sometimes it is divided into two separate documents, one covering Functional Requirements and one covering Non-functional Requirements, as the method of specifying one vs. the other can be different. Use the templates described earlier in this Guide. The Requirements Traceability Matrix will be used to trace relationships between requirements (and underlying assumptions) and forward to design and test specifications.

Requirements documents are not intended to be literary masterpieces. In fact, the requirements document can be rather “boring.” The document is describing business requirements and will often be read (and signed off) by business people, and therefore needs to be understood by business people, using the commonly accepted phraseology of the client organization.

Here are some style guidelines:

- Keep sentences short.
- Never express more than one requirement per sentence.

- Avoid the use of jargon, acronyms, and abbreviations unless everyone who will read the document understands them.
- Keep paragraphs short. (Seven sentences or less).
- Use lists and tables wherever possible.
- Use terminology consistently. (The use of a data dictionary can help).
- Use “shall,” “should,” “will,” and “must” consistently. Shall means that the requirement is mandatory. “Should” means the requirement is desirable, but not mandatory. “Will” indicates that something will be externally provided. “Must” is best avoided.
- Do not express requirements using nested conditional clauses (if A then if B then R1a else if C then R1b else R1c). It might be easy for a programmer to understand this, but not most of the readers of your requirements document.
- Use the active rather than the passive voice. (Give an Example)
- Don’t use anonymous references.
- Pay attention to spelling and grammar.
- Have both the R&D team as well as the cross-functional team review or inspect your requirements document. That’s one way to validate and verify it.
- Keep a history of changes to the document so that you can see how the product evolved over time.

Also:

- Be specific, clear and unambiguous (SMART)
- Describe “what” not “how”
- Separate functionality and quality (FURPS+)
- Get some scale for measuring - fine tune later
- Use qualifiers to be specific (everything can be qualified)
- Document things only once (reference when possible)
- Name sources/references in detail (pages, sections)
- Use graphics where possible
- Where appropriate, define implications if requirements are not met
- Use pictures to clarify and give the big picture
- Number the requirements and assign a unique identifier to each one, e..g. Lnnnn, where L stands for the requirement type (FURPS+), and nnnn for the number of the requirement in its respective category.

A key challenge when responding to client needs is how to translate those needs, as expressed by the client, into solutions that HP can deliver in a product or set of products within a release. A major part of this difficulty is that customer needs are often passed along in imprecise, unclear and/or incomplete forms. This can lead to statements of work that do not represent what the customer truly wants or needs, or statements of work that do not provide the sufficient depth of information needed. This is sometimes not discovered until during the delivery process. One way to reduce the risk of this occurring is to pay more attention to the requirements elicitation process, and perform it in a more structured manner.

Prototyping is the time-honored way to get user inputs, especially for graphical interfaces. Keep in mind that there are different levels of prototypes you can build from pictures of the screens that you hand around in a meeting to partially functional programs. Although prototyping is most often used during the requirements phase, you can also use it in the external design or implementation phase. It’s important to ensure that you can actually build what is being prototyped. There's nothing more frustrating, especially to end users, to see a really cool prototype and then have the development team tell them that they can't have it for 5 years. Also, be careful that people realize that you're showing them a prototype (especially if it's on a computer and they're upper level managers).

Prototyping considerations:

- Paper-based
- Computer mock-up
- Partially functional program
- Most often used during requirements phase
- Can be used during design or implementation phase for exploring alternatives
- Ensure you're able to develop what you prototype

Requirements are captured and recorded in a Requirements Specification (or two, if non-functional requirements are documented separately) and tracked via a Requirements Traceability Matrix. It is useful to identify and record requirements in separate categories (beyond functional and non-functional) for ease of verifying completeness and to facilitate tracking.

Writing Requirements & Completeness Check

Now let us consider how a requirement should be stated. There are a number of characteristics that need to be met before one has a well-stated requirement, as English, as a natural language (along with all other natural languages) tends to be too ambiguous without careful use. Here is one list of attributes that well formed requirements statements should meet. A requirements statement should be:

Representation Techniques

There are a number of ways to represent requirements in a specification. The most common is to simply use text, supplemented by pictures and diagrams. Two additional techniques are provided here as well.

Use Cases

Two good references on this methodology are from Ivar Jacobson's Object-Oriented Software Engineering and Karl Wiegers' Creating a Software Engineering Culture. Briefly, the idea is to describe the user's expected interactions with the system. Each usage of the system is a "use case". For example, in a warehouse management system, "Assigning docking bays to trucks," "Locating available shelf space," and "Routing pallets of deliveries," could all be use cases. Don't develop use cases for every conceivable usage of the system, but rather for the uses that will occur most frequently.

Once you have the use case described, you can think about how you could test to verify whether or not the system implemented the use case. Doing this may uncover some problems with your use case. You may also be able to go through use cases with clients as part of your review process, although some client personnel have difficulty with this way of representing requirements, and may require some training as part of the review session.

For each type of user

- For each major usage of the system
  - Describe the goal the user is trying to accomplish
  - Describe the expected frequency of this use case
  - Describe the expected sequence of actions the user would expect to take with the system and the responses the system would generate
- Test cases can be easily derived from use cases

Decision Tables/Trees

Decision tables or trees are used when you have several conditions that can affect the behavior of a system (for instance, "system calibrated", "test plan loaded", "fixture enabled", etc…). The conditions should have a few discrete states each (either Boolean or 3 or 4 values). Describe each possible combination of conditions. Some combinations will be error states, but that's OK. In fact, error processing is one of those things that is often left unspecified, undesigned, and unimplemented until late in a "code-and-fix" project, and then causes a lot of slippage.

Sometimes a tree is a more efficient representation and sometime a table is better. Pick whichever one seems best.

- Use when there are a combination of conditions that can influence the system behavior
- Decision table:
  - Draw a row for each condition that will be used
  - Draw a column for every possible combination of outcomes of these conditions
  - Add rows at the bottom of the table for each possible action or response
  - Fill in the action rows for each combination of conditions
- Decision tree:
  - Like a flow chart without loops and without fan-in

Appendix A: SMART Method

- Complete: Nothing is missing; it conforms to the standard template.
- Consistent: It does not conflict with any other requirements.
- Correct: It accurately states a user need that must be satisfied.
- Feasible: It can be implemented within existing constraints. (Including project cost and schedule).
- Modifiable: The structure and style of the requirement is such that changes can be made when necessary. (If it is too hard to change the requirements document, you won't. Even if it is easy to change, that won't guarantee you'll keep it up-to-date.)
- Necessary: It documents something the users need, not something the developers included because they thought the users would like it. (There's no problem with delighting users, but sometimes we don't have an accurate understanding of our users and put in features that delight us but are confusing, irrelevant, or actually harmful to our users).
- Prioritized: Requirements are ranked as to how essential it is to incorporate each one into the delivered solution. This includes dividing them into *Musts* and *Wants*, as well as consideration of possible multiple releases or versions, or a staged implementation.
- Testable: Tests can be devised to demonstrate whether the requirement is properly implemented. (Often, a requirements traceability matrix is developed to ensure that each requirement is actually implemented in the system.)
- Traceable: The requirement is uniquely identified (either via a numbering scheme or via an unique name) so that it can be traced onto corresponding design, code, and testing components of the system. (Even if you don't develop the traceability matrix initially, it is handy to get in the habit of ensuring requirements are traceable.)
- Unambiguous: It has only one possible interpretation.
- Design and implementation independent: Try to state the user's needs rather than describing features. This is hard to do, but the more you can determine the underlying needs, the more flexibility you have in designing a solution to meet those needs.

Now this is a fine and quite complete list, but is rather difficult to remember when analyzing client statements and attempting to turn them into well formed requirements. Because of this difficulty, an adaptation has been made of another acronym (**SMART**) that was developed at Leeds University in Great Britain in 1992 in support of developing good objectives. In its original context SMART stands for:

- **S**pecific
- **M**easurable
- **A**ttainable
- **R**ealizable
- **T**ime bounded

Time boundedness, while a good and applicable characteristic with respect to objectives, does not fit very well in the context of requirements. However, many requirements in a Requirements Specification are dependent on other requirements or are part of a higher-level requirement. Also, common criticism of some requirements is that the original justification is lost. It would be better for a Requirements Engineer to think of **T** as standing for Traceable. If it is not possible to envisage how a particular requirement is related to other requirements and to know where it came from, then it is not a SMART requirement.

Accordingly, some researchers (Mike Mannion and Barry Keepence) in the Department of Mechanical, Manufacturing and Software Engineering, Napier University, in Edinburgh, Scotland have modified the acronym so it reads:

- **S**pecific
- **M**easurable
- **A**ttainable
- **R**ealizable
- **T**raceable

Their paper on this subject has served as the basis for this section of the Guide. This easy to remember acronym is short enough to be usable, and allows reviewers of requirements to more readily check to see if they have been carefully stated in an way that will facilitate their use as the source of a solution design and test cases.

Specific

All requirements techniques have a criterion in this area. A requirement must say exactly what is required. Specificity actually comprises several areas as follows:

- Clear i.e. that there is no ambiguity;
- Consistent i.e. that the same terminology has been used throughout the specification to describe the same system element or concept;
- Simple i.e. avoid double requirements e.g. X and Y;
- Of an appropriate level of detail.

A requirement can usually be tested for specificity by simply reading it. There are a number of words and phrases that are first rate indicators of an unspecific requirement. Consider the following requirement:

> “The Mission Planning System shall support several planning environments for generating the mission plan.”

In this example, it is not clear what is meant by "several." In addition the terms "planning environment" and “mission plan" may not have been defined.

In general terms the following guidelines are recommended:

- Avoid terms such as: "obviously," clearly," and "certainly."
- Avoid ambiguities such as: "some", “several", and "many."
- Avoid list terminators such as: "etc," "and so on," and "such as...."
- Ensure pronouns are clearly referenced e.g. "When module A calls B its message history file is updated".
- When numbers are specified, identify the units.
- Ensure all possible elements in a list are described.
- Use pictures to clarify understanding.
- Ensure all system or project terms are defined in a glossary.
- Consider placing individual requirements in separate individually numbered paragraphs.
- Ensure verbs such as "transmitted," "sent," "downloaded," and "processed" are qualified by precise explanations.

- Only use the word "details," "information," and “data" in a requirement when you can describe or refer to precisely what they will be.
- If the requirement is described by a prototype program, ensure that specific program is documented.
- When a term is defined in a glossary, substitute the definition in the text and then review the requirement.
- No "To Be Defineds".

Measurable

In the context of Requirements Engineering, by measurable we mean is it possible, once the system has been constructed, to verify that this requirement has been met. In some software engineering methodologies, the Requirements Engineer is instructed to determine the tests that must be performed in order to satisfy the requirement. This is a good discipline. The level of detail required to describe and set up the corresponding test is itself a strong indicator of whether the requirement should be broken down into sub-requirements.

Assuming that a requirement is specific, non-measurable requirements fall into two categories:

- Those which cannot be instrumented (or instrumentation interferes);
- Those which are specific but for which there is no yardstick available.

In general terms the following guidelines are recommended.

- What other requirements need to be verified before this requirement?
- Can this requirement be verified as part of the verification for another requirement? If so, which one?
- How much data or what test cases are required?
- How much processing power is required?
- Can the test be conducted on one site?
- Can this requirement be tested in isolation?

Attainable

By an attainable requirement we mean it is possible physically for the system to exhibit that requirement under the given conditions. Some requirements may be beyond the bounds of human knowledge. Others may have theoretical solutions but be beyond what is currently achievable. The consequence of attempting to meet such requirements is that the system will never be accepted or prohibitively expensive, or both.

In general terms the following guidelines are recommended:

- Is there a theoretical solution to the problem?
- Has it been done before? If not, why not?
- Has a feasibility study been done?
- Is there an overriding constraint that prohibits this requirement?
- Are there physical constraints on the size of the memory, processor or peripherals?
- Are there environmental constraints such as temperature, compressed air?

It is often the case that the attainable and realizable criteria are often considered in parallel. This does not however make them synonymous.

Realizable

In the context of software requirements, by realizable we mean is it possible to achieve this requirement given what is known about the constraints under which the system and the project must be developed. Determining whether a requirement is realizable or not is the most difficult part of creating a SMART requirement. The difficulty is twofold in nature:

- Can we satisfy this requirement given the other system and physical constraints that we have?
- Can we satisfy this requirement given the project resource constraints that we must work to?

For example, if there is a requirement to have 99% reliability but the project budget does not permit the inclusion of the extensive defensive programming needed to satisfy that requirement, then that requirement is not realistic.

In general terms the following guidelines are recommended:

- Determine who has responsibility for satisfying the requirement.
    - Can they deliver?
    - Can we afford to manage them?
- How badly is it needed?
- Are there sufficient resources?
    - staff with the right skill set;
    - space and desks;
    - hardware and software for development;
    - hardware and software for testing.
- Is there sufficient time?
- Is there sufficient budget?
- Are we constrained to a particular package that does not support this requirement?
- Will we have to develop it ourselves?
- Can we reuse from other projects?

During the first iteration of the Requirements Specification, requirements are often placed into one of two categories:

- Essential (musts)
- Desirable (wants)

If an analyst is not sure about a requirement then it is often marked as desirable. This does not change the requirement. Desirable requirements should only be left in requirements documents when there is a clear choice in the development stage. For example:

“The system must have ten operator chairs” - essential
“The operator chairs should be red” – desirable

If they were any other color, the system would still be acceptable.

Traceable

Requirements traceability is the ability to trace (forwards and backwards) a requirement from its conception through its specification to its subsequent design, implementation and test. It is important for the following reasons:

- So that we can know and understand the reason for each requirement's inclusion within the system
- So that we can verify that each requirement has been implemented
- So that modifications are made easily, consistently and completely

Most systems and software development projects that can demonstrate evidence of traceability have been driven to do so by the second of these three reasons. This applies also to CASE tools that support traceability. While such a view of traceability is essential, it does not help us understand why individual or combinations of requirements have been included, nor explain hidden requirements inter-relationships such as dependency or implication. Hence, in the specification of a requirement the provision of the following supplementary information, where appropriate, should be made:

- Originators of requirements (institutions or people)
- Underlying assumptions (These are particularly important. Often an underlying assumption applies to many requirements but the assumption is stated once, often several pages away from a requirement which is dependent on it. It is also vital to ask what will happen when (not if) the underlying assumption is not true[1])
- Business justifications
- Inter-relationships such as subsumption, dependency or implication. These sorts of relationships are vital in determining the impact of any changes brought about to the requirements specification.
- Their criticality

Glossary Terms:

Software Requirements

- **SMART** (**S**pecific, **M**easurable, **A**ttainable, **R**ealizable, **T**raceable) that was developed at Leeds University in Great Britain in 1992 and was later refined at Napier University in Edinburgh.
- **FURPS** (**F**unctionality, **U**sability, **R**eliability, **P**erformance, **S**upportability) – HP R&D Labs, 1984

Reliability

dependability; absence of failure
Reliability includes the product and/or system's ability to keep running under stress and adverse conditions. In the case of an application, reliability relates to amount of time available and running versus time unavailable. Specify reliability acceptance levels, and

[1] Typically an assumption is removed from a system (i.e. becomes invalid) but all the requirements that depended upon it are not removed. This often leads to features which are not required still being implemented.

how they will be measured and evaluated. Describe release criteria in measurable terms. If your application/product will be delivered to an integration function, i.e. will become part of a larger system, your application's or product's reliability criteria may be set by the integration function. If you have any criteria in addition to (over and beyond) the integration criteria, include those criteria in this section. Consider design robustness, design stability, learning products, or other criteria in addition to hardware and software reliability. Some of the sub-characteristics you should consider are:

- Frequency/Severity of Failures
- Recoverability
- Predictability
- Accuracy
- Mean Time to Failure