10 Best Practices in Writing Requirements

Introduction

Requirements are the conditions or capabilities needed by a user to solve a business problem or achieve an objective. An agency should prepare high level requirements to articulate the "need" which can then be developed into business requirements. The need and business requirements should be able to be traced back to the business case and project vision.

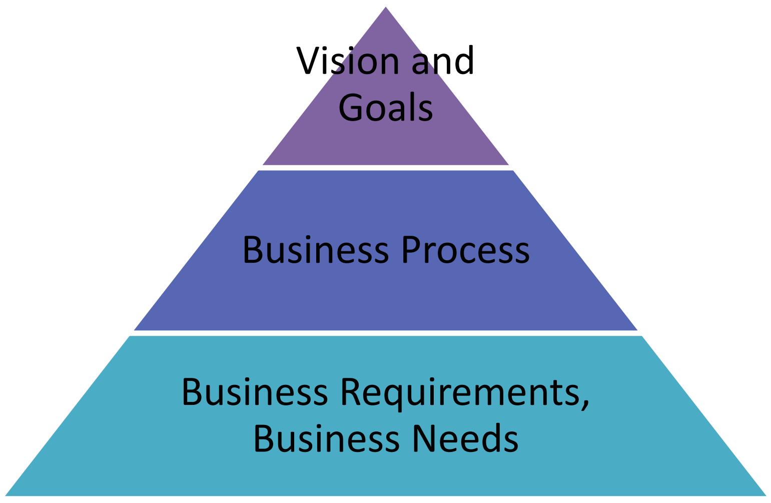
Good requirements do the following:

- Establish a common understanding between the sponsor, project manager stakeholders, and technical team.
- Provide a roadmap for development
- Should be simple, verifiable, necessary, achievable and traceable.

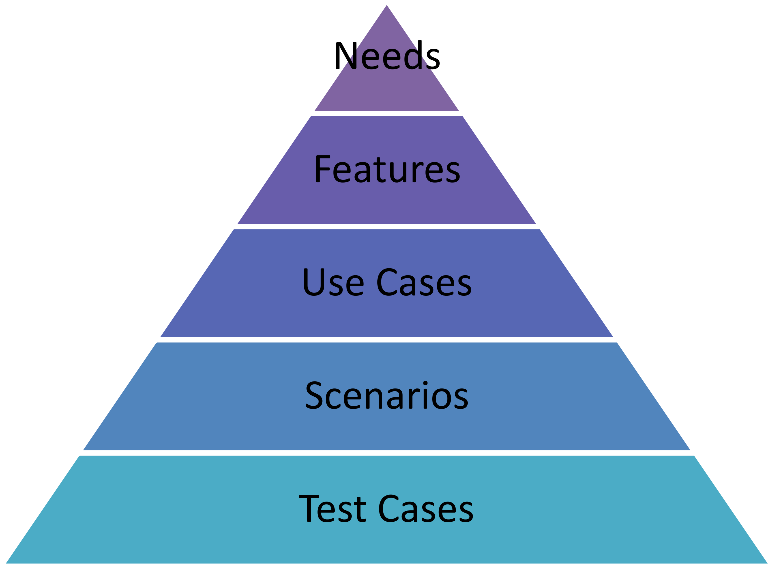
Types of Requirements

Requirements can be divided into **functional** and **non-functional** requirements. **Functional requirements** provide a high-level description of how a system or product should function from the end user perspective. Functional requirements try to address both business and technical requirement and include for whom the product is built, how it might be used, interactions and guidelines to be followed. **Non-functional** requirements represent the qualities of the system and constraints in which the system operates.

The **requirements pyramid** below shows the progression from the organization's vision and goals at the top to the business requirements that then become business "needs".



Those **needs** define the requirements. They are then further refined into features, use cases, scenarios and test cases.



Recommended practices in developing requirements are explained below.

1. Use a Basic, Best-practice Format

A basic requirement structure is Unique ID: Object + Provision/Imperative (shall) + Action + Condition + [optional] Declaration of Purpose /Expected Occurrence (will). Use accepted requirement sentence formats wherever possible. Consider using the EARS: The Easy Approach to Requirements Syntax[1] method which provides a number of proven patterns for writing specific types of requirements, as shown in the table below.

Requirement Type	Syntax Pattern
General[2] or Ubiquitous	The <system name> shall <system response>.
Event Driven	WHEN <trigger> <optional precondition> the <system name>shall <system response>.
Unwanted	IF <unwanted condition or event>, THEN the <system name> shall <system response>.
State Driven	WHILE <system state>, the <system name> shall <system response>.
Optional Feature	WHERE <feature is included>, the <system name> shall <system response>.
Complex	(Combinations of the above patterns)

[1] Mavin et al.

[2] A word about general requirements Many requirements that may seem general are really driven by some trigger or condition. Rewriting the requirement in the unwanted behavior format makes the trigger-response nature of the requirement more clear. Be sure to check all "general" requirements – especially if they're functional requirements – for hidden triggers. Most true general (or ubiquitous) requirements are non-functional.

Ideally, every requirement statement (written from the user's perspective) should contain a user role that benefits from the requirement, the desirable state the user role wants to achieve and a metric that allows the requirement to be tested. Avoid speculation and drawing up wish lists of features that are impossible to achieve.

Last, make sure you define only one requirement at a time. Don't use conjunctions (and, or, also, with) because these can cause developers to miss out on requirements. Split complex requirements until each one can be considered a discreet test case.

2. Define Terms and Use them in a Standard Manner

Create a dedicated section toward the beginning of your requirements document to define exactly how certain terms will be used within the document itself, and how they should be interpreted when found in non-requirements documents referenced by the document. For example:

- SHALL is used for binding requirements that must be verified and have an accompanying method of verification.
- MUST denotes certain quality and performance requirements that must be verified and have an accompanying method of verification. MUST is typically applied to non-functional requirements.
- WILL is used as a statement of fact (informational), declaration of purpose, or expected occurrence and is not binding
- SHOULD denotes an attribute, goal, or best practice which must be addressed by the system design (informational) and is not binding. [3]

The use of SHALL for functional requirements and MUST for non-functional requirements helps easily distinguish between the two. Use exactly one provision or declaration of purpose (such as shall) for each requirement, and use it consistently across all requirements. Strictly defining your terms and adhering to your definitions will reduce conflict and confusion in interpreting your requirements, and, with practice, will save you time in writing requirements.

3. Keep Functional Requirements Free of Design Details and Descriptions of Operations

Functional requirements should specify the required external output behavior of the system for a stated set or sequence of inputs applied to its external interfaces. In other words, state WHAT the system must do, not HOW it must do it. Constraints on manner of implementation should not appear in functional requirements. They should be spelled out in very specific non-functional requirements at the outset of the program. Keeping functional requirements free of design details allows engineers to design the system in the most efficient manner available, implementation to be modified without rewriting the requirement and reduces the possibility of conflict between requirements due to incompatible implementation details. Ask yourself WHY do

[3] These definitions are considered standard in the industry and can be applied to state projects. These terms are collectively called "imperatives" in the IT industry.

you need the requirement. If you catch yourself mentioning field names, programming language and software objects in the Requirements Specification Document, you're in the wrong zone.

Likewise, avoid descriptions of operations. Ask "does the developer have control over this?" Requirements that include "the user shall" are almost always operational statements, not requirements.

4. Include Additional and Supporting Information

It is vitally important to separate the supporting information from the requirement statement. Trying to weave complex supporting information or data into a requirement statement can make the statement overly complex and unclear to the reader. Additional information can be referenced in the following manner:

- **Rationale statements** can be used to reduce ambiguity in your requirements document. They allow you to simplify your requirements statement while providing users with additional information. A short and concise sentence is usually all that is needed to convey a single requirement – but it's often not enough to justify a requirement. Separating your requirements from their explanations and justifications enables faster comprehension, and makes your reasoning more evident. When a requirement's rationale is visibly and clearly stated, its defects and shortcomings can be more easily spotted, and the rationale behind the requirement will not be forgotten. Rationale statements also reduce the risk of rework, as the reasoning behind the decision is fully documented and thus less likely to be re-rationalized.
- **Assumptions** should be articulated and you should ask yourself if the assumptions could be validated.
- **Directives** are words or phrases that point to additional information which is external to the requirement, but which clarifies the requirement. Directives typically employ phrases like "as shown in" and "in accordance with," and they often point to tables, illustrations or diagrams. They may also reference other requirements or information located elsewhere in the document.
- **Exception scenarios** are conditions in which a given requirement should not apply or should be altered in some way. On the other hand, if multiple exception scenarios were identified, it might be better to create a bulleted list of exceptions, to make the requirement easier to read.

5. Be Clear in Your Wording

Requirements should be specific, rather than vague, but vagueness is epidemic in requirements specifications. Customers may like a vague requirement, reasoning that if its scope is unbounded, they can refine it later when they have a better idea of what they want. Authors and engineers may not mind, since a slack requirement may appear to give them more "freedom" in their implementation. All eventually suffer, however, when the implementation misses the mark and extensive rework is required. To avoid vagueness:

- Do not use unspecific adjectives (weak words) such as easy, straightforward, or intuitive

- Do not express suggestions or possibilities (identified by might, may, could, ought)
- Use active voice (shall + present tense verb) and avoid passive voice (shall be + past participle)
- Define precisely what the system needs to do (in functional requirements) or to be (in non-functional requirements) in such terms that compliance can be readily observed, tested or otherwise verified. Include tolerances for qualitative values.
- Do not use “to be determined” or “to be resolved”. Instead, include the current best estimate and state the rationale as to why the value is an estimate.
- Don’t be swayed by those who want to keep requirements vague.

Keep in mind the costs of scrap and re-work while defining requirements. Also be mindful of the following:

Weak Words/Unclear Terms – also called subjective, vague or ambiguous words – are adjectives, adverbs and verbs that don’t have a concrete or quantitative meaning. Such words are thus subject to interpretation. Weak words include:

> Efficient, powerful, fast, easy, effective, reliable, compatible, normal, user-friendly, intuitive, few, most, quickly, versatile, robust, timely, strengthen, enhance, flexible, large, small, sufficient, safe, adequate, approximate, minimal impact, as appropriate, but not limited to, be able to, be capable of, useable when required.

Define your requirements in precise, measurable terms. Don’t specify that a system or feature will be intuitive, reliable or compatible; define WHAT will make it intuitive, reliable or compatible.

Passive Voice - Many adjectives that are also past participles of verbs – words like enhanced, strengthened and ruggedized – are notorious weak words, because they sound like engineering terms, but are weak in specificity. Changing from shall + passive to must + active clarifies the requirement

Negative Requirements - Use negative specifications primarily for emphasis, in prohibition of potentially hazardous actions. Then state the safety case in the rationale for the requirement. Don’t use negative specification for requirements that can be restated in the positive. Substitute shall enable for shall not prohibit, shall prohibit in place of shall not allow, and so on. Last, avoid double negatives completely-- use shall allow instead of shall not prevent, for example.

Compatibility - If the system being designed must be compatible with other systems or components, explicitly state the specific compatibility requirements. Don’t leave it up to the hardware and software engineers to determine what will make the system they’re designing “compatible” with a given device (and expect the test engineers to make the same determination). It’s up to you to define what it means to be compatible with the device in question.

6. Organize, Standardize and Templatize

Organize your requirements in a hierarchical structure. In component specifications, for example, a functional hierarchy is often used, with very broad functional missions at the top breaking down into sub-functions, and those sub-functions breaking down into successive tiers of sub-functions.

Use industry accepted identifiers and ensure that each requirement in every requirement document be tagged with a project-unique identifier. Requirements documents that do not employ such an identifier system are not only difficult to read and reference, they make traceability a nightmare.

Turn standardized sections into “boilerplate” to promote and facilitate consistency across projects. This is a major benefit of templates. These sections tend to remain little changed from project to project, and from team to team– evolving only slowly over time with changes in methodology and lessons learned – thus providing a stable platform for consistent requirements development, employee education and communication with customers.

A template should have, at a minimum, a cover page, section headings, essential guidelines for the content in each section and a brief explanation of the version (change) management system used to control changes made to the document. The template should also include standardized sections covering topics like verb (imperative) application, formatting and traceability standards, and other guidelines your organization follows in documenting requirements and managing its requirements documentation.

7. Make Sure Each Requirement is Testable

Requirements should be stated in such a way that an objective test can be defined for it. Writing your requirement with a specific test scenario in mind will help ensure that both design and test engineers understand exactly what they have to do. A good practice for insuring requirement testability, for example, is to specify a reaction time window for any output event the software must produce in response to a given input condition. The verification or test method, the means to test the fulfillment of the requirement, and the criteria for verification should also be included.

8. Write from a User perspective and Vet Requirements with a Diverse Team

Consider the needs of all potential stakeholders who will interact with the system. The list of these stakeholders may well go beyond what had been initially considered and should take into consideration all relevant domain experts, and even users. Identify your stakeholders early, consider their use levels, and write from their perspective.

Besides writing requirements from the perspective of a client or manager, evaluate requirements with a diverse team. This team should consist of designers and developers who will use the requirements to create the system, the testers who will verify compliance with the requirements, engineers who design, maintain or manage other systems that will support or interact with the new system, and end-user representatives. Any subsequent additions or changes to the requirements should undergo a similar evaluation as part of a formal change management system. This greatly increases the probability that the requirements will meet the needs of all stakeholders. Make note of which users were heavily considered for each requirement, so you can have that user provide focused feedback only on the requirements that are relevant to them.

9. Make sure the Requirements are Complete

Complete requirements contain several components and you should check your final requirements for completeness. Requirements include the following types:

- Functional
- Performance
- Interface
- Environment
- Training
- Personnel
- Operability and Safety/Security
- Appearance and Physical Characteristics
- Design

Make sure all described functions are necessary, and together, sufficient to meet the system needs, goals and objectives. Also consider reliability, maintainability and survivability, among other factors.

10. Make Sure Requirements are Traceable

Traceability in this context is about relationships between requirements at the same or different levels of detail, and between requirements and other lifecycle artifacts Each requirement should be able to be traced to a parent requirement or business need or, if it’s at the top level, to the project scope. Requirement Traceability helps you follow the life of a requirement (from idea to implementation), see how requirements impact one another, and understand requirement decomposition—from high level user needs to design specifications.

There are several different levels of traceability. Basic traceability establishes a relationship or link between one or more elements. Typed traceability adds the relationship type with its associated semantics. Rich traceability adds additional information on the traceability relationship such as rationale and assumptions.

Once traceability has been established there are multiple ways in which it can be viewed and reported on. The *traceability matrix* is the oldest and most commonly recognized method. The matrix allows you to see the intersection between two sets of requirements and a check or cross shows where a link exists but this method doesn’t scale particularly well since the matrix could become very large. A *traceability column* allows you to pick a starting point, and display the related systems requirements alongside the user requirement they are linked to. You can choose how much detail of the linked requirement is displayed, and even make it recursive, going down as many levels of requirements as you need/is practical to manage in a single view. Graphical displays, such as the *traceability tree,* are great for getting a bigger picture view of traceability rather than immediately focusing in on the details of a particular relationship. You can explore the traceability tree, zooming in/out or collapsing/expanding parts of the tree, or changing the focus (starting point) of the tree.

Traceability delivers value in your project by providing the context for a requirement (the business WHY), and illustrating the audit trail (why a requirement exists) or compliance (which requirement satisfies the regulation). It also helps show all the user requirements have been covered and can highlight gaps or can reveal over-engineering or “gold plating”. Finally, it allows for impact analysis so that, when one requirement changes or a design proves infeasible, you can identify all the related requirements, designs, tests, and work items that are potentially impacted by the change. This enables you to fully scope the impact of the changes.