Recommended Practice for Software Requirements Specifications (IEEE)

Author: John Doe

Revision: 29/Dec/11

Abstract: The content and qualities of a good software requirements specification (SRS) are described and several sample SRS outlines are presented. This recommended practice is aimed at specifying requirements of software to be developed but also can be applied to assist in the selection of in-house and commercial software products. Guidelines for compliance with IEEE/EIA 12207.1-1997 are also provided.

Table of Contents

1 Overview

OPEN

This recommended practice describes recommended approaches for the specification of software requirements.

It is divided into five clauses. Clause 1 explains the scope of this recommended practice. Clause 2 lists the references made to other standards. Clause 3 provides definitions of specific terms used. Clause 4 provides background information for writing a good SRS. Clause 5 discusses each of the essential parts of an SRS. This recommended practice also has two annexes, one which provides alternate format templates, and one which provides guidelines for compliance with IEEE/EIA 12207.1-1997.

1.1 Scope

OPEN

This is a recommended practice for writing software requirements specifications. It describes the content and qualities of a good software requirements specification (SRS) and presents several sample SRS outlines.

This recommended practice is aimed at specifying requirements of software to be developed but also can be applied to assist in the selection of in-house and commercial software products. However, application to already-developed software could be counterproductive.

When software is embedded in some larger system, such as medical equipment, then issues beyond those identified in this recommended practice may have to be addressed.

This recommended practice describes the process of creating a product and the content of the product. The product is an SRS. This recommended practice can be used to create such an SRS directly or can be used as a model for a more specific standard.

This recommended practice does not identify any specific method, nomenclature, or tool for preparing an SRS.

Dependant		
depends on	4	Considerations for producing a good SRS

2 References

OPEN

This recommended practice shall be used in conjunction with the following publications.

ASTM E1340-96, Standard Guide for Rapid Prototyping of Computerized Systems.1

IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology.2

IEEE Std 730-1998, IEEE Standard for Software Quality Assurance Plans.

IEEE Std 730.1-1995, IEEE Guide for Software Quality Assurance Planning.

IEEE Std 828-1998, IEEE Standard for Software Configuration Management Plans.3

IEEE Std 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software.

IEEE Std 982.2-1988, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software.

IEEE Std 1002-1987 (Reaff 1992), IEEE Standard Taxonomy for Software Engineering Standards.

IEEE Std 1012-1998, IEEE Standard for Software Verification and Validation.

IEEE Std 1012a-1998, IEEE Standard for Software Verification and Validation: Content Map to IEEE/EIA 12207.1-1997.4

IEEE Std 1016-1998, IEEE Recommended Practice for Software Design Descriptions.5

IEEE Std 1028-1997, IEEE Standard for Software Reviews.

IEEE Std 1042-1987 (Reaff 1993), IEEE Guide to Software Configuration Management.

IEEE P1058/D2.1, Draft Standard for Software Project Management Plans, dated 5 August 1998.6

IEEE Std 1058a-1998, IEEE Standard for Software Project Management Plans: Content Map to IEEE/EIA 12207.1-1997.7

IEEE Std 1074-1997, IEEE Standard for Developing Software Life Cycle Processes.

IEEE Std 1233, 1998 Edition, IEEE Guide for Developing System Requirements Specifications.8

Dependant		
depends on	4	Considerations for producing a good SRS

3 Definitions

OPEN

In general the definitions of terms used in this recommended practice conform to the definitions provided in IEEE Std 610.12-1990. The definitions below are key terms as they are used in this recommended practice.

3.1 contract: A legally binding document agreed upon by the customer and supplier. This includes the technical and organizational requirements, cost, and schedule for a product. A contract may also contain informal but useful information such as the commitments or expectations of the parties involved.

3.2 customer: The person, or persons, who pay for the product and usually (but not necessarily) decide the requirements. In the context of this recommended practice the customer and the supplier may be members of the same organization.

3.3 supplier: The person, or persons, who produce a product for a customer. In the context of this recommended practice, the customer and the supplier may be members of the same organization.

3.4 user: The person, or persons, who operate or

Dependant		
depends on	4	Considerations for producing a good SRS

4 Considerations for producing a good SRS

OPEN

This clause provides background information that should be considered when writing an SRS. This includes the following:

1. Nature of the SRS;
2. Environment of the SRS;
3. Characteristics of a good SRS;
4. Joint preparation of the SRS;
5. SRS evolution;
6. Prototyping;
7. Embedding design in the SRS;
8. Embedding project requirements in the SRS.

Dependant		
dependency for	1.1	Scope
dependency for	2	References
dependency for	3	Definitions

Relation		
related by	4.3	Characteristics of a good SRS

4.1 Nature of the SRS

OPEN

The SRS is a specification for a particular software product, program, or set of programs that performs certain functions in a specific environment. The SRS may be written by one or more representatives of the supplier, one or more representatives of the customer, or by both. Subclause 4.4 recommends both.

The basic issues that the SRS writer(s) shall address are the following:

1. *Functionality.* What is the software supposed to do?
2. *External interfaces.* How does the software interact with people, the system's hardware, other hardware, and other software?
3. *Performance.* What is the speed, availability, response time, recovery time of various software functions, etc.?
4. *Attributes.* What are the portability, correctness, maintainability, security, etc. considerations?
5. *Design constraints imposed on an implementation.* Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.?

The SRS writer(s) should avoid placing either design or project requirements in the SRS.
For recommended contents of an SRS see Clause 5.

4.2 Environment of the SRS

OPEN

It is important to consider the part that the SRS plays in the total project plan, which is defined in IEEE Std 610.12-1990. The software may contain essentially all the functionality of the project or it may be part of a larger system. In the latter case typically there will be an SRS that will state the interfaces between the system and its software portion, and will place external performance and functionality requirements upon the software portion. Of course the SRS should then agree with and expand upon these system requirements.

IEEE Std 1074-1997 describes the steps in the software life cycle and the applicable inputs for each step. Other standards, such as those listed in Clause 2, relate to other parts of the software life cycle and so may complement software requirements.

Since the SRS has a specific role to play in the software development process, the SRS writer(s) should be careful not to go beyond the bounds of that role. This means the SRS

- Should correctly define all of the software requirements. A software requirement may exist because of the nature of the task to be solved or because of a special characteristic of the project.
- Should not describe any design or implementation details. These should be described in the design stage of the project.
- Should not impose additional constraints on the software. These are properly specified in other documents such as a software quality assurance plan.

Therefore, a properly written SRS limits the range of valid designs, but does not specify any particular design.

4.3 Characteristics of a good SRS

OPEN

An SRS should be

1. Correct;
2. Unambiguous;
3. Complete;
4. Consistent;
5. Ranked for importance and/or stability;
6. Verifiable;
7. Modifiable;
8. Traceable.

Dependant		
depends on	5	The parts of an SRS
Relation		
relates to	4	Considerations for producing a good SRS

4.3.1 Correct

OPEN

An SRS is correct if, and only if, every requirement stated therein is one that the software shall meet.
There is no tool or procedure that ensures correctness. The SRS should be compared with any applicable superior specification, such as a system requirements specification, with other project documentation, and with other applicable standards, to ensure that it agrees. Alternatively the customer or user can determine if the SRS correctly reflects the actual needs. Traceability makes this procedure easier and less prone to error

(see 4.3.8).

4.3.2 Unambiguous

OPEN

An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation. As a minimum, this requires that each characteristic of the final product be described using a single unique term.

In cases where a term used in a particular context could have multiple meanings, the term should be included in a glossary where its meaning is made more specific.

An SRS is an important part of the requirements process of the software life cycle and is used in design, implementation, project monitoring, verification and validation, and in training as described in IEEE Std 1074-1997. The SRS should be unambiguous both to those who create it and to those who use it. However, these groups often do not have the same background and therefore do not tend to describe software requirements the same way. Representations that improve the requirements specification for the developer may be counterproductive in that they diminish understanding to the user and vice versa.

Subclauses 4.3.2.1 through 4.3.2.3 recommend how to avoid ambiguity.

4.3.2.1 Natural language pitfalls

OPEN

Requirements are often written in natural language (e.g., English). Natural language is inherently ambiguous. A natural language SRS should be reviewed by an independent party to identify ambiguous use of language so that it can be corrected.

4.3.2.2 Requirements specification languages

OPEN

One way to avoid the ambiguity inherent in natural language is to write the SRS in a particular requirements specification language. Its language processors automatically detect many lexical, syntactic, and semantic errors.
One disadvantage in the use of such languages is the length of time required to learn them. Also, many non-technical users find them unintelligible. Moreover, these languages tend to be better at expressing certain types of requirements and addressing certain types of systems. Thus, they may influence the requirements in subtle ways.

4.3.2.3 Representation tools

OPEN

In general, requirements methods and languages and the tools that support them fall into three general categories—object, process, and behavioral. Object-oriented approaches organize the requirements in terms of real-world objects, their attributes, and the services performed by those objects. Process-based approaches organize the requirements into hierarchies of functions that communicate via data flows. Behavioral approaches describe external behavior of the system in terms of some abstract notion (such as predicate calculus), mathematical functions, or state machines.

The degree to which such tools and methods may be useful in preparing an SRS depends upon the size and complexity of the program. No attempt is made here to describe or endorse any particular tool.

When using any of these approaches it is best to retain the natural language descriptions. That way, customers unfamiliar with the notations can still understand the SRS.

4.3.3 Complete

OPEN

An SRS is complete if, and only if, it includes the following elements:

- All significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interfaces. In particular any external requirements imposed by a system specification should be acknowledged and treated.
- Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations. Note that it is important to specify the responses to both valid and invalid input values.
- Full labels and references to all figures, tables, and diagrams in the SRS and definition of all terms and units of measure.

4.3.3.1 Use of TBDs

OPEN

Any SRS that uses the phrase "to be determined" (TBD) is not a complete SRS. The TBD is, however, occasionally necessary and should be accompanied by

- A description of the conditions causing the TBD (e.g., why an answer is not known) so that the situation can be resolved;
- A description of what must be done to eliminate the TBD, who is responsible for its elimination, and by when it must be eliminated.

4.3.4 Consistent

OPEN

Consistency refers to internal consistency. If an SRS does not agree with some higher-level document, such as a system requirements specification, then it is not correct (see 4.3.1).

4.3.4.1 Internal consistency

OPEN

An SRS is internally consistent if, and only if, no subset of individual requirements described in it conflict. The three types of likely conflicts in an SRS are as follows:

1. The specified characteristics of real-world objects may conflict. For example,
    1. The format of an output report may be described in one requirement as tabular but in another as textual.
    2. One requirement may state that all lights shall be green while another may state that all lights shall be blue.
2. There may be logical or temporal conflict between two specified actions. For example,
    1. One requirement may specify that the program will add two inputs and another may specify that the program will multiply them.
    2. One requirement may state that "A" must always follow "B," while another may require that "A and B" occur simultaneously.
3. Two or more requirements may describe the same real-world object but use different terms for that object. For example, a program's request for a user input may be called a "prompt" in one requirement and a "cue" in another. The use of standard terminology and definitions promotes consistency.

4.3.5 Ranked for importance and/or stability

OPEN

An SRS is ranked for importance and/or stability if each requirement in it has an identifier to indicate either the importance or stability of that particular requirement.

Typically, all of the requirements that relate to a software product are not equally important. Some requirements may be essential, especially for life-critical applications, while others may be desirable.

Each requirement in the SRS should be identified to make these differences clear and explicit. Identifying the requirements in the following manner helps:

- Have customers give more careful consideration to each requirement, which often clarifies any hidden assumptions they may have.

- Have developers make correct design decisions and devote appropriate levels of effort to the different parts of the software product.

4.3.5.1 Degree of stability

OPEN

One method of identifying requirements uses the dimension of stability. Stability can be expressed in terms of the number of expected changes to any requirement based on experience or knowledge of forthcoming events that affect the organization, functions, and people supported by the software system.

4.3.5.2 Degree of necessity

OPEN

Another way to rank requirements is to distinguish classes of requirements as essential, conditional, and optional.

- *Essential.* Implies that the software will not be acceptable unless these requirements are provided in an agreed manner.
- *Conditional.* Implies that these are requirements that would enhance the software product, but would not make it unacceptable if they are absent.
- *Optional.* Implies a class of functions that may or may not be worthwhile. This gives the supplier the opportunity to propose something that exceeds the SRS.

4.3.6 Verifiable

OPEN

An SRS is verifiable if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if, and only if, there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement. In general any ambiguous requirement is not verifiable.

Nonverifiable requirements include statements such as "works well," "good human interface," and "shall usually happen." These requirements cannot be verified because it is impossible to define the terms "good," "well," or "usually." The statement that "the program shall never enter an infinite loop" is nonverifiable because the testing of this quality is theoretically impossible.

An example of a verifiable statement is

> *Output of the program shall be produced within 20 s of event ¥ 60% of the time; and shall be produced within 30 s of event ¥ 100% of the time.*

This statement can be verified because it uses concrete terms and measurable quantities.

If a method cannot be devised to determine whether the software meets a particular requirement, then that requirement should be removed or revised.

4.3.7 Modifiable

OPEN

An SRS is modifiable if, and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style. Modifiability generally requires an SRS to

- Have a coherent and easy-to-use organization with a table of contents, an index, and explicit cross-referencing;
- Not be redundant (i.e., the same requirement should not appear in more than one place in the SRS);
- Express each requirement separately, rather than intermixed with other requirements.

Redundancy itself is not an error, but it can easily lead to errors. Redundancy can occasionally help to make an SRS more readable, but a problem can arise when the redundant document is updated. For instance, a requirement may be altered in only one of the places where it appears. The SRS then becomes inconsistent. Whenever redundancy is necessary, the SRS should include explicit cross-references to make it modifiable.

4.4 Joint preparation of the SRS

OPEN

The software development process should begin with supplier and customer agreement on what the completed software must do. This agreement, in the form of an SRS, should be jointly prepared. This is important because usually neither the customer nor the supplier is qualified to write a good SRS alone.

1. Customers usually do not understand the software design and development process well enough to write a usable SRS.
2. Suppliers usually do not understand the customer's problem and field of endeavor well enough to specify requirements for a satisfactory system.

Therefore, the customer and the supplier should work together to produce a well-written and completely understood SRS.

A special situation exists when a system and its software are both being defined concurrently. Then the functionality, interfaces, performance, and other attributes and constraints of the software are not predefined, but

rather are jointly defined and subject to negotiation and change. This makes it more difficult, but no less important, to meet the characteristics stated in 4.3. In particular, an SRS that does not comply with the requirements of its parent system specification is incorrect.

This recommended practice does not specifically discuss style, language usage, or techniques of good writing. It is quite important, however, that an SRS be well written. General technical writing books can be used for guidance.

4.5 SRS evolution

OPEN

The SRS may need to evolve as the development of the software product progresses. It may be impossible to specify some details at the time the project is initiated (e.g., it may be impossible to define all of the screen formats for an interactive program during the requirements phase). Additional changes may ensue as deficiencies, shortcomings, and inaccuracies are discovered in the SRS.

Two major considerations in this process are the following:

1. Requirements should be specified as completely and thoroughly as is known at the time, even if evolutionary revisions can be foreseen as inevitable. The fact that they are incomplete should be noted.
2. A formal change process should be initiated to identify, control, track, and report projected changes. Approved changes in requirements should be incorporated in the SRS in such a way as to
    1. Provide an accurate and complete audit trail of changes;
    2. Permit the review of current and superseded portions of the SRS.

4.6 Prototyping

OPEN

Prototyping is used frequently during the requirements portion of a project. Many tools exist that allow a prototype, exhibiting some characteristics of a system, to be created very quickly and easily. See also ASTM E1340-96.

Prototypes are useful for the following reasons:

1. The customer may be more likely to view the prototype and react to it than to read the SRS and react to it. Thus, the prototype provides quick feedback.
2. The prototype displays unanticipated aspects of the systems behavior. Thus, it produces not only answers but also new questions. This helps reach closure on the SRS.
3. An SRS based on a prototype tends to undergo less change during development, thus shortening

development time.

A prototype should be used as a way to elicit software requirements. Some characteristics such as screen or report formats can be extracted directly from the prototype. Other requirements can be inferred by running experiments with the prototype.

4.7 Embedding design in the SRS

OPEN

A requirement specifies an externally visible function or attribute of a system. A design describes a particular subcomponent of a system and/or its interfaces with other subcomponents. The SRS writer(s) should clearly distinguish between identifying required design constraints and projecting a specific design. Note that every requirement in the SRS limits design alternatives. This does not mean, though, that every requirement is design.

The SRS should specify what functions are to be performed on what data to produce what results at what location for whom. The SRS should focus on the services to be performed. The SRS should not normally specify design items such as the following:

- Partitioning the software into modules;
- Allocating functions to the modules;
- Describing the flow of information or control between modules;
- Choosing data structures.

4.7.1 Necessary design requirements

OPEN

n special cases some requirements may severely restrict the design. For example, security or safety requirements may reflect directly into design such as the need to

1. Keep certain functions in separate modules;
2. Permit only limited communication between some areas of the program;
3. Check data integrity for critical variables.

Examples of valid design constraints are physical requirements, performance requirements, software development standards, and software quality assurance standards.

Therefore, the requirements should be stated from a purely external viewpoint. When using models to illustrate the requirements, remember that the model only indicates the external behavior, and does not specify a

design.

4.8 Embedding project requirements in the SRS

OPEN

4.8 Embedding project requirements in the SRS

The SRS should address the software product, not the process of producing the software product.

Project requirements represent an understanding between the customer and the supplier about contractual matters pertaining to production of software and thus should not be included in the SRS. These normally include items such as

- Cost;
- Delivery schedules;
- Reporting procedures;
- Software development methods;
- Quality assurance;
- Validation and verification criteria;
- Acceptance procedures.

Project requirements are specified in other documents, typically in a software development plan, a software quality assurance plan, or a statement of work.

5 The parts of an SRS

OPEN

This clause discusses each of the essential parts of the SRS. These parts are arranged in Figure 1 in an outline that can serve as an example for writing an SRS.

While an SRS does not have to follow this outline or use the names given here for its parts, a good SRS should include all the information discussed here.

Dependant		
dependency for	4.3	Characteristics of a good SRS

5.1 Introduction (Section 1 of the SRS)

OPEN

The introduction of the SRS should provide an overview of the entire SRS. It should contain the following subsections:

- Purpose;
- Scope;
- Definitions, acronyms, and abbreviations;
- References;
- Overview.

5.1.1 Purpose (1.1 of the SRS)

OPEN

This subsection should

- Delineate the purpose of the SRS;
- Specify the intended audience for the SRS.

5.1.2 Scope (1.2 of the SRS)

OPEN

This subsection should

- Identify the software product(s) to be produced by name (e.g., Host DBMS, Report Generator, etc.);
- Explain what the software product(s) will, and, if necessary, will not do;
- Describe the application of the software being specified, including relevant benefits, objectives, and goals;
- Be consistent with similar statements in higher-level specifications (e.g., the system requirements specification), if they exist.

5.1.3 Definitions, acronyms, and abbreviations (1.3 of the SRS)

OPEN

This subsection should provide the definitions of all terms, acronyms, and abbreviations required to properly interpret the SRS. This information may be provided by reference to one or more appendixes in the SRS or by reference to other documents.

5.1.4 References (1.4 of the SRS)

OPEN

This subsection should

1. Provide a complete list of all documents referenced elsewhere in the SRS;
2. Identify each document by title, report number (if applicable), date, and publishing organization;
3. Specify the sources from which the references can be obtained.

This information may be provided by reference to an appendix or to another document.

5.1.5 Overview (1.5 of the SRS)

OPEN

This subsection should

- Describe what the rest of the SRS contains;
- Explain how the SRS is organized.

5.2 Overall description (Section 2 of the SRS)

OPEN

This section of the SRS should describe the general factors that affect the product and its requirements. This section does not state specific requirements. Instead, it provides a background for those requirements, which are defined in detail in Section 3 of the SRS, and makes them easier to understand.
This section usually consists of six subsections, as follows:

- Product perspective;
- Product functions;
- User characteristics;
- Constraints;
- Assumptions and dependencies;
- Apportioning of requirements.