Quality Software Requirements

By J. Chris Gibson

It has been stated that deficiencies in software requirements are the leading cause of failure in software projects.[1] If this is true then the contrapositive should also be true. This article presents a case study that supports this implication. The study is based on observations (and documentation) made of a software development project and the process used to produce the software requirements. The project focus was to develop a business critical application called ProcMon[2] (for Process Monitor) which upon completion would be charged with keeping watch over a client's automated order entry system. Our company (the software consultancy) was afforded an appropriate amount of time to develop a quality software requirement specification (SRS), and in contrast with prior experiences, the project proceeded smoothly through design, implementation, testing and delivery of the product. The project completed on time and under budget and it was my great fortune to have landed the project manager assignment for the ProcMon project.

Quality Properties of Requirements

ProcMon was a small budget project and as such, its team members had to perform multiple duties. In addition to project manager responsibilities I was responsible for coordinating the requirements development process. Previous experiences taught me that the quality of the software requirements specification (SRS) is directly connected to the success or failure of a software development project. Reference material cited and other sources typically describe software requirements quality in terms of the SRS. This article identifies quality with respect to the requirement (as a conceptual entity), and the requirements document (SRS) separately. For the ProcMon project I stuck with the following standard criteria for requirement quality:

- Design Independent. A software requirement is free of design and implementation decisions except in the form of a constraint (i.e. operating system constraint). A software requirement is design independent if it describes an aspect of a problem space without describing any implementation details of a solution to the problem space or aspect of the problem space (the old *what*, not *how* paradigm).
- Unambiguous. A requirement is unambiguous if it has only one possible interpretation. Much of this quality has to do with the use of terminology. Agreement on the definition of terms must be made and published in the SRS under a "Definitions" section of some sort. Also of course, the use of clear, concise, language should be used.
- Precise. A requirement is precise if it exactly defines a behavioral aspect of the software including data sets and ranges for outputs and inputs. For example, any requirement of some measurement must use discrete numeric values within appropriate range or levels, for instance a response time should not be described in a requirement as having to be "fast", but rather should be described as "*n* milliseconds" where *n* is some agreed upon numeric value.
- Traceable. A requirement is traceable if it is uniquely identifiable. Any document that references the requirement (i.e. software design description) must be able to make the

reference to an exact requirement as identified by some unique identifier. In the design and testing phases of the software development effort it is essential to know what requirement is being supported. The following are some useful methods for ensuring that the SRS contains traceable requirements.

1. Number each paragraph hierarchically. Sentences within the paragraph can be referenced by their order of appearance in the paragraph.
2. Number each paragraph and limit the paragraph to one specific requirement.
3. Number or otherwise identify (numbers, letters, names) each individual requirement.
4. Use a naming identifier i.e. the word "shall" to indicate a specific requirement.

Considering the diversity of these methods it becomes obvious that there is no one right way to make a requirement traceable. Select or devise a method that is appropriate for the project.

- Understandable. A requirement must be meaningful to the people who read it. A requirement is understandable if it accurately conveys the customer requirement to its intended audience.
- Verifiable. A requirement is verifiable as a behavioral aspect of the software. A requirement is verifiable if there is a quantifiable or observable effect of the software that is directly expressed by the requirement.
- Prioritized. Unless all requirements are of equal importance a requirement should be ranked by priority. And if all requirements are of equal importance it should be stated that requirements are not ranked because they all have the same priority. A requirement can be prioritized either by degree of stability or degree of functional necessity. Degree of stability is concerned with the number of modifications expected of a requirement during the development process. Functional necessity can be designated by such qualifiers as *essential*, *conditional*, and *optional*, or some numeric scheme, or by whatever method or terminology of ranking seems appropriate.

All of these qualities can be expressed in terms of the conceptual software requirement. The following quality properties are expressed with respect to the SRS (document):

- Complete. An SRS contains all the information needed to produce a design description of the software and no more. An SRS is complete if 1) everything that the software is supposed to do is represented in the document, 2) software responses to all possible classes of input in all possible situations is described, 3) all references are made, pages numbered and terms are defined. Measuring completeness (as with other quality aspects of the SRS as a whole) is an activity that can be repeated throughout the development of the software product. At any point in the development of the software requirements, circumstances may cause the creation of new requirements. One requirement may beget another. Completeness can be estimated with some degree of accuracy as provided by the various available metrics (i.e. C = U/IxS where U is the total number of known functions, I is the total number of known inputs and S is the total known system states)[3]. Such metrics (mathematic formula) are available for measuring other properties of quality as well.
- Consistent. An SRS is consistent if there are no contradictory statements between the requirement statements contained within.

- Organized. The SRS is well organized if its contents are arranged in such a way that readers can easily locate information. A well-organized SRS should follow industry standard, boilerplate document layout for a Software Requirements Specification. Specific (functional) requirements should be grouped by some common schema i.e. user class, stimulus, object, feature, or other such commonality.
- Modifiable. The SRS is modifiable if its content is organized in such a way that it is easy to change. The term "easy" is obviously subjective enough, but there are strategies that can ease document modification efforts. Modification ease is greatly enhanced if the document is traceable. Also an overhead saver in making modifications can be realized by numbering pages by section. This allows reprinting of only the modified section for hardcopy upkeep. Modifiability is important because requirements change throughout the life of a project.

There are other qualitative properties of a software requirement and an SRS, but the ones mentioned here are found in all of the referenced material and in the majority of other notable sources. Other qualities include that a requirement should be concise, electronically stored, not redundant and reusable to mention a few. Considering that these properties may not be critical, and are somewhat self-explanatory this article does not attempt to reduce or analyze them any further.

Conversely a poorly formed requirement or a poorly written SRS would be one that lacks one or more of these qualities or by some other deficiency fails to accurately represent a software solution within the problem domain as defined by its stakeholders. The primary source of the requirement specification should be derived from a formal analysis of the problem domain. The overall effectiveness of the requirements capturing phase of development and the resulting SRS depends on the accuracy of the problem analysis and definition.

The Requirements Engineering Process

Our client's problem domain became evident when unanticipated volumes overloaded the static variable memory model of their Internet order entry system. The ProcMon project was a sidebar project to the main project of revising the order entry system's memory allocation woes. Regardless of the generous time allotment we began the requirements engineering process immediately with interviews of management personnel to determining their expectations of the intended software product. The activities involved in cultivating requirements and producing our SRS included:

- Requirements Elicitation.
- Analysis (concept analysis).
- Modeling the requirements and writing the specification (SRS).
- Verification of the requirements.

Elicitation

The elicitation of data for the purpose of producing software requirements is possible through the use of a variety of methods and techniques such as surveys of, and interviews with users of the software. Elicitation (and subsequent analysis) is critical to the requirements engineering

process in that it is in this development stage that the primary requirements (user's needs) are identified. Often this includes negotiating with the customer on feasibility issues of attributes or other requirements. We needed data about the overall system and its operational characteristics in addition to information about the problem domain due to the nature of the concept analysis that we were planning to use. A concept analysis is defined by Fairley and Thayer as "the process of analyzing a problem domain and an operational environment for the purpose of specifying the characteristics of a proposed system from the user's perspective." [4] Data gathering activities used for the ProcMon in addition to interviews of management personnel included interviews with systems personnel (the intended end-users) to determine their expectations.

Analysis

The analysis phase constitutes the evaluation of existing conditions that represent the primary motivation of the problem domain as evidenced by data gathered during the requirements elicitation phase. Concept analysis is a user-centric approach that focuses on an integrated view of the overall system and its operational characteristics. The product of the concept analysis process is the Concept of Operations document (the ConOps [4]). The ConOps contains information that fully describes user needs, goals, operational environment, processes, and characteristics of the intended system. The elements of a ConOps as described by Fairley and Thayer [4] include:

- A description of the current system or situation.
- A description of the needs that motivate development of a new system or modification of an existing system.
- Modes of operation for the proposed system.
- User classes and user characteristics.
- Operational features of the proposed system.
- Priorities among proposed operational features.
- Operational scenarios for each operational mode and class of user.
- Limitations of the proposed approach.
- Impact analysis for the proposed system.

The ConOps should be written using a narrative prose and terminology of the customer or system procurators. The primary role of the ConOps is to communicate user needs to the system developers. Development of the ConOps should include the identification of *use cases*, which in turn are used to model the requirements. (Use case diagrams should be included in the SRS.) I selected a standard template for the ConOps document. Industry standard templates and definitions are available from standards organizations. Given the modest functionality of the ProcMon project in conjunction with using a standard document template I was able to write a rough draft of the ConOps in four days. On completion of the rough draft an inspection meeting was held to determine if all modes of operation, user classes, operational features, scenarios (use cases), etc. were completely represented. After a few minor adjustments and a three more days of rewriting, the ConOps was complete.

Modeling and Specification

Modeling requirements and writing the requirements specification is the focal point of the requirements engineering process. The process of writing the SRS for the ProcMon project proceeded from analysis to requirements modeling. Use case modeling was used to capture process requirements and entity-relationship data models were used (where applicable) to depict data requirements of the system. The following sequence of events occurred in the course of developing the ProcMon SRS.

1. Software configuration management procedures were reviewed and made applicable to the ProcMon project. A meeting was held to update new team members on version control policies and procedures for software documentation. Requirements modification procedures (use of SMR documents) and version control practices were also explained to the appropriate client personnel.
2. General system constraints were defined based on the client's policies concerning software development, the existing hardware platform, security considerations, the communication protocol available on the client's computer network, and interface requirements of the order entry system were reviewed. The needs of the parallel operations (order processing), what development environment would be used (including programming languages and third party products) were also discussed.
3. An SRS document template was derived from an easily accessible and well-defined industry standard [5]. The team opted to organize the functional-detail section of the SRS by *stimulus* because of the nature of problem domain. The outline example represents the template configuration prior to having its subsection titles altered to reflect project specifics.

```
1. Introduction
    1.1 Purpose
    1.2 Scope
    1.3 Definitions
    1.4 References
    1.5 Document Overview
2. General Description
    2.1 Product Perspective
    2.2 Product Functions
    2.3 User Characteristics
    2.4 Constraints
    2.5 Assumptions and Dependencies
3. Specific Requirements
    3.1 External Interface Requirements
       3.1.1 User Interfaces
       3.1.2 Hardware Interfaces
       3.1.3 Software Interfaces
       3.1.4 Communication Interfaces
    3.2 Functional Requirements
       3.2.1  Information Flow
       3.2.2  Process Descriptions
       3.2.3  Data Requirements
    3.3 Performance Requirements
    3.4 Design Constraints
    3.5 Software System Attributes
    3.6 Other Requirements
```

4. I wrote a draft of the subsections for the 1: Introduction, 2: General Description, and 3.1 External Interface Requirements based on the ConOps and other data gathered thus far and distributed a copy to each team member along with a copy of the ConOps and a format for the details of the Functional Requirements section.
5. Each team member was assigned one or more general requirements for decomposition into specific functional requirements. The approach to writing these sections uses natural language prose in conjunction with UML use cases, and ER diagrams for data representation. Steps involved included:

    - Describe the requirement stimulus and define the purpose of each associated functional requirement.
    - Identify sources of inputs, outputs, valid data ranges and formats. Indicate persistence requirements as applicable.
    - Identify and model data requirements including entity types, attributes and relationships using ER diagrams.
    - Formalize use cases derived by analysis using the Unified Modeling Language (UML) Use Case diagrams.
    - Identify new use cases with respect to the defined stimulus (if applicable).
    - Describe *primary* functional requirements based on use cases and other variables exposed by the concept analysis. Refer to diagrams where applicable.
    - Likewise describe *derived* functional requirements, as they are made evident.

6. Inspection meetings of each team member's requirements were held to review and verify their respective requirements sections and to explore the possibilities of alternative solutions and identify any overlooked requirements. Inspection meeting participants included myself, the author of the requirements section, one other team member, and a representative of the client user group. The inspection meeting includes participant descriptions of the requirements to allow the other participants to compare their respective interpretations of the requirements under review. Different perspectives are useful in revealing omissions and ambiguities. (For more information on requirements inspection see "Inspecting Requirements" by Karl Wiegers.[6])
7. After extensive inspections and reworking requirements to satisfy changes deemed necessary the functional requirements sections were collected, edited for clarity, style, etc., and inserted into the master document. (Finished diagrams were included with their respective requirements section.)

As a matter of course derived requirements surface as primary requirements are defined. Primary requirements can often be reduced to lower level (derived) requirements, which can eliminate ambiguity. (For more information on primary and derived requirements see Harwell.[7]) Each requirement was uniquely identifiable by the numbering scheme provided by the SRS template. Other general guidelines for writing good requirements include:

- Seek to satisfy all known properties of quality.
- Be succinct.
- Do not use subjective terminology such as "fast" or "robust" (unambiguous property).
- Do not use terms out of their context as defined in the Definitions section (consistency).
- Avoid redundancy.

Verification

Software requirements verification ensures that the allocation of the overall system requirements is appropriate and correct. The verification process includes an examination of the project documentation and an evaluation of product testability. Activities of verification are pervasive throughout the development process (The inspection meetings are part of the verification process). The activities of software verification include the following:

- Evaluation of the ConOps. Determine whether the requirements satisfy the defined product concepts. Evaluation should include operable aspects of system functionality, performance, and feasibility (i.e. hardware compatibility etc.) Identify constraints and update the ConOps to reflect constraints that may have been overlooked.
- Initial test planning. Verification of requirements should include identification of each type of testing strategy that is appropriate (unit testing, integration, distribution, etc.).
- Evaluation of the SRS. The SRS is inspected for accuracy, completeness, and the other quality properties. Verify whether fault tolerance requirements are appropriate and feasible. Determine whether and how well the requirements achieve system objectives. Assess requirement criticality. Determine whether requirements comply with organization standards. This step can be considered satisfied by the individual inspections of requirements performed during the Modeling and Specification stage or a larger scale inspection meeting covering all requirements of the SRS can be held.
- Software interface analysis. Verify that software interfaces (human user and hardware interfaces) are feasible, accurate, and posses other quality properties.

In conclusion, this article presents an overall picture of the art of writing quality, natural language software requirements, and provides information about writing requirements based on the ProcMon case study. Software is inherently complex. Producing the software requirements for a given system will entail a level of difficulty that reflects the intended software product's overall complexity. Writing a requirement document that satisfies all aspects of quality described in this article is a labor-intensive task. But working towards satisfying all of the quality properties is the only way to write good software requirements. And the benefits of writing good requirements vastly outweigh the difficulties encountered in the effort.

References

[1] Hofmann, H., Lehner, F., (2001) "Requirements Engineering as a Success Factor in Software Projects", IEEE Software July/August 2001, Los Alamitos CA.

[2] Gibson, J., (1998), ProcMon Software Requirements Specification, Protera Software Corporation, Surfside Beach Texas, USA.

[3] Davis, A., Overmyer, S., Jordan, K., Caruso, J., Dandshi, F., Dinh, A., Kincaid, G., Ledeboer, G., Reynolds, P., Sitaram, P., Ta, A., and Theofanos, M., "Identifying and Measuring Quality in A Software Requirements Specification", (1993) Proc. 1st International Software Metrics Symposium, IEEE Computer Society Press, Los Alamitos California USA, pp. 141-152.

[4] Fairley, R., Thayer, R., Bjorke, P., (1994) "The Concept of Operations: The Bridge from Operational Requirements to Technical Specifications", Proc. 1st International Conference on Requirements Engineering, IEEE Computer Society Press, Los Alamitos California USA, pp. 40-47.

[5] Std 830-1998, IEEE Standards Software Engineering, 1999 Edition Volume Four: Resource and Technique Standards, pp. 11-24.

[6] Wiegers, K., (2001), "Inspecting Requirements", StickyMinds.com website publication.

[7] Harwell, R., (1997), "What Is A Requirement", Software Requirements Engineering Second Edition, edited by Richard Thayer and Merlin Dorfman, IEEE Computer Society Press, Los Alamitos California USA.