Software Quality Improvement

Software Requirements Engineering: Practices and Techniques

Ronald Kirk Kandt

November 7, 2003

Jet Propulsion Laboratory
California Institute of Technology

Table of Contents

Table of Tables

1. Introduction

1.1. Motivation

In 2001, the Software Technology Infusion element of the Software Quality Improvement Project surveyed over twenty software practitioners and managers to identify the practices and tools that they used [24]. As a result of that survey, members of the Software Technology Infusion element determined that they could have the greatest impact on software engineers at Jet Propulsion Laboratory (JPL) by focussing their activities on software verification and validation and requirements engineering. This report documents the effort that has been performed to date with respect to requirements engineering.

The goals of this effort were to:

- Identify the current practice of requirements engineering at JPL;
- Analyze the current practice and identify problems in it;
- Identify best practices to overcome these problems and avoid others that may arise; and
- Evaluate existing software tools that can support requirements engineering.

The approach that was used to satisfy these goals were to:

- Interview system and software engineers to determine how they currently perform work;
- Analyze the results of these interviews to identify best practices and unfulfilled needs;
- Examine existing requirements documents to identify issues that can result in the creation of either wrong or defective software; and
- Review the literature for best practices.

An outcome of this approach is that the author observed that the problems associated with requirements documents developed by JPL personnel are consistent with those of industry. Some of these include the following.

- Requirements were often incompletely or vaguely stated.
- Assumptions and rationales for defined requirements were rarely stated.
- Practitioners rarely were trained in how to define good requirements or how to use provided tools.

This is not an all-encompassing set of problems, and these problems do not necessary apply to every person defining requirements at JPL. They are simply a common set of problems representative of a significant portion of the population that define system and software requirements. These issues are discussed in more detail later in the document, but before discussing them let me first provide definitions and identify characteristics of good requirements engineering.

1.2. Definitions

A *requirement* is a necessary, quantifiable, and verifiable capability, function, property, characteristic, or behavior that a product must exhibit to solve a real-world problem, or a constraint that it must satisfy or be satisfied during the development of a product. *Requirements engineering* is the process of developing requirements. It consists of a structured set of activities that results in the production of a *requirements document*, which is a formal statement of the system requirements.

Requirements are generally imposed by multiple stakeholders from different organizations and at different levels of the operating environments where people work. Thus, requirements may be technical or non-technical. Non-technical requirements include agreements, conditions, and contractual terms. Examples of these are the identification of mile-

stones, delivered products, delivery and milestone dates, and acceptance criteria. Technical requirements, on the other hand, are either functional or non-functional.

Functional requirements define the functional capabilities, or behavior, of a product. *Nonfunctional requirements* identify the constraints that a product must satisfy, which are mandated product qualities [37]. Non-functional requirements often identify CPU utilization, data, interface, memory, operational, performance, sizing, and timing needs [27]. Non-functional requirements specified less often identify availability, distribution, interoperability, maintainability, portability, reliability, reusability, robustness, safety, security, supportability, and usability needs of a product. Typical kinds of reusable requirements include those that constrain the system or system operations, identify the style of presentation, and reflect organizational policies. Since it is generally difficult to understand the impact of non-functional requirements on a product, satisfying them is generally much harder than for functional ones. Furthermore, many of these properties emerge over time, often when individual subsystems are integrated into a larger system.

Functional and non-functional requirements apply to a product while process requirements specify how a product is to be produced. Typical process requirements specify the programming languages, database management systems, and other software tools that are to be used. Sometimes, requirements may be levied on the actual development process, such as “the contractor shall perform software inspections on every design document before coding begins.”

1.3. Common Problems

Many problems arise during requirements engineering. These problems include not defining needed requirements and inconsistently or ambiguously defining them. Some of the causes of such problems result from using the wrong people to define requirements [20, 27, 28]. These people often:

- misunderstand their responsibilities,
- lack adequate experience or knowledge,
- suffer from unclear reasoning and biases,
- incorrectly understand the assumptions or facts underlying requirements, and
- select solutions before they understand the user’s needs or requirements.

In addition, stakeholders often [27]:

- do not completely understand their needs,
- do not fully understand computer capabilities and limitations,
- have conflicting views, and
- do not participate in requirements engineering.

A consequence of these problems is the definition of unstable requirements, which is one of the most common sources of cost and schedule overruns and project cancellations [23, 29, 43]. Hence, volatile requirements generally have significant impact on product development and either must be eliminated before development occurs or managed as they arise during development. However, since some domains can only be understood over time and organizational goals often change because of broader organizational change, organizations must learn to embrace change as a necessary part of survival since the tractability and invisibility of software exposes it to continual requirements churn [5, 7, 8]. Furthermore, since individual skills, team experience, and proper domain knowledge are important factors in requirements definition, organizations should attempt to develop teams of highly skilled software engineers and managers with high levels of domain knowledge to obtain more predictable performance [3, 19].

Organizations with sound requirements engineering practices spend ten to fifteen percent of total project funds to develop requirements [19, 20]. Generally speaking, spending less than ten percent of total project funds increases total mission cost because too many introduced defects must be corrected, whereas spending more than fifteen percent of total project funds tends to yield modest gains in the quality of developed artifacts at great cost. Experience

shows that projects that invest ten percent or more of the total project development funds in requirements engineering achieve cost overruns of zero to fifty percent, whereas those organizations that spend five percent or less exhibit overruns of one- to two-hundred percent. However, even if ten to fifteen percent of total project effort is allocated to requirements engineering, requirements engineers must use good practices and processes to be efficient and effective. The benefits of a sound methodology for requirements engineering more than recoup the investment. These benefits include:

- elimination of unneeded features,
- faster rework,
- less product training,
- fewer operational errors,
- lower development, maintenance, and operations cost, and
- shorter schedules.

1.4. Overview

The remainder of this paper discusses several issues. Section 2 identifies the findings of a small-scale assessment of the requirements engineering process at JPL, based on both interviews of systems and software engineers and the examination of existing requirements documents. Section 3 identifies practices for elicitating, analyzing, writing, and validating requirements, as well as monitoring the requirements engineering process. These practices can be used as a basis for defining a process for defining system and software requirements. This section also illustrates several practices with requirement statements that appeared in the examined documents, along with suggestions for improving them. Section 4 discusses existing methods for the elicitation and analysis of requirements. Section 5 presents a recommended high-level process for defining requirements. Section 6 summarizes the main body of the paper. Appendix A provides the questionnaire that the author used to conduct the requirements engineering assessment and Appendix B identifies a verification checklist that can be used to assess the quality of requirements. Finally, Appendix C contains several examples of problematic requirement statements.

2. An Assessment of Requirements Engineering Performance at JPL

2.1. Characteristics of Interviewed Personnel

Ten people were interviewed that defined, used, and managed the definition of requirements. These people work in Divisions 31, 33, 36, and 38 and on data analysis, flight, and ground applications. About half of these people were requirements engineers that had twenty years of experience defining requirements. All of these people have spent at least ten years defining requirements while employed by JPL. However, few of these people had received training in requirements engineering. One person received training when she obtained a bachelor's degree in Systems Engineering. Another person recently attended a class in Requirements Engineering, but has not had the opportunity to use that training.

These people were selected to be interviewed based on personal contacts. For example, the author met one person in a requirements training activity. Several other people were known to the author or his associates to have defined the requirements for the subsystems of several JPL missions. In one case, the interviewed person is a representative to the oversight group that advises and monitors the Software Quality Improvement Project. In another case, a software requirements engineer was recommended by a group manager; this person defines all the requirements for all projects of that group. The types of systems that these people helped to define include both ground and flight, hardware and software, and at the highest and lowest levels of system definition. In sum, the type of systems for which these people defined requirements were quite diverse.

2.2. Interview Findings

Definition of project scope. People define project scope in a variety of ways, although it is usually defined by the customer. Sometimes the requirements engineer has the luxury of negotiating the project scope with the customer, but the statement of scope is usually communicated as a statement of work or as a response to a solicitation. In the case of internally-developed multi-mission software, the project scope is generally negotiated with the development team and defined as a unit of change to existing products. No one identified the existence of a vision statement, per se, although it is possible that such a statement may be embedded in an originating solicitation.

Identification of stakeholders. Rarely do people make an attempt to identify the stakeholders explicitly. In their opinion, it is generally clear who they are. Knowledge of the stakeholders is based on who the past customers have been and through announcement of opportunities. Typically, stakeholders include science and development teams, users, and testers.

Requirement elicitation and negotiation techniques. There are four basic means that requirements engineers use to define requirements. First, requirements engineers simply accept requirements directly from the customer. Sometimes acceptance may involve negotiation. Second, requirements engineers generally interview stakeholders during informal weekly meetings and clarification sessions. Third, requirements engineers organize brainstorming sessions to elicit knowledge, although this generally occurs while designing system architectures. Fourth, requirements engineers use personal knowledge based on prior missions to define the requirements. Sometimes, the requirements are reviewed by the customer, which may cause some editing of requirements specifications.

One person discussed a failure in using the RAPID software development methodology (which appears to be an agile method). The use of this brainstorming approach resulted in too many opinions about each requirement topic and so the focus of each meeting tended to be too broad. These meetings were originally run by the contractor, but subsequently the JPL requirements engineer assumed control over each meeting. Each meeting now has a specific focus, and participants discuss only the items of the meeting's agenda. Only four or five people actively participate in each meeting, although a couple of developers attend to provide immediate feedback on the viability of proposed ideas. Finally, defined criteria for each meeting must be satisfied before a meeting can be held. If these conditions are not satisfied, the meeting is postponed. The result of this new approach is an increased ability to resolve issues.

In multi-mission product development, inconsistencies between the requirements of different customers are negotiated. The end result is that each customer may have its desired requirements totally or partially satisfied or not satisfied at all. Furthermore, since an existing product is used as a baseline in multi-mission development, requirements engineering occurs incrementally.

Forms of representation. People often produce context and block diagrams to identify the needs of their customers, although they do not consistently use these techniques. People indicated that scenarios are beneficial for identifying both existing and planned processes at both abstract and concrete levels. However, documents describing a concept of operations [1, 40] containing scenarios are rarely used, perhaps because such documents provided to software teams often contain incorrect scenarios and are often delivered too late to be of use. For one large and long-duration project, developers have intimate knowledge of the software and have a clear understanding of new requirements. Thus, members of this project do not specify or use a description of a concept of operations. Instead, they prototype software to demonstrate the correctness of algorithms.

The team of one interviewed person has had very good experience with documents that describe a concept of operations. This team develops such documents for internal use, and these documents have proven extremely valuable when explaining system functionality to customers. This team also has developed a unique way of viewing scenarios that identifies when functionality must be introduced into the system and for whom. This view is simply a table where user roles identify one axis, software development phases identify the second axis, and the contents of each cell corresponds to a collection of scenario names.

Identification of requirement characteristics. Most people distinguish between guidance, functional requirements, and non-functional requirements, although few people address quality attributes of requirements in any way. Of the

quality attributes, performance was cited most often, although in many instances there is no way to verify that a performance requirement has been satisfied. People sometimes, although rarely, defined maintainability, reliability, and usability requirements too. Since non-functional requirements are seldom identified, omissions or defects in non-functional requirements are typically identified during acceptance testing. In sum, performance and maintenance issues are the only types of requirements that were identified by more than one person.

About one-third of the interviewees document the assumptions and rationales underlying requirements. However, there is no standard way of doing this. Assumptions may be documented at the end or the beginning of a requirements document. In other instances, they accompany individual requirements. Similarly, rationales may be associated with individual requirements or included as phrases within requirement statements. One person justified imprecision in the following manner: "I have observed that most requirements are vague and I believe they are intentionally written so because it permits the necessary leeway to state that requirements were satisfied."

About one-third of the people prioritize requirements and use them to schedule deliveries. One person gathers the stakeholders together and lets them negotiate the priorities. In one multi-mission development organization, the team negotiates requirements with the customer by providing several alternatives. Two alternatives always identify what can be done within the requested schedule and how long it will take to deliver all the requested requirements. NASA Headquarters, in its *JPL FY 2002 Final Performance Evaluation*, states that an identified deficiency of the Planetary Data System is that there is "no mechanism for prioritizing requirements that includes participation of the user community." This comment not only identifies the importance of prioritization, but also stakeholder involvement.

Requirements engineers rarely specify how requirements should be verified, although several people thought it would be valuable. About half the interviewees subject requirements documents to peer review, and a few people indicated that they expect peer review processes to detect omitted or duplicated requirements. These people indicated that no guidance is provided to them or others when they review requirements, although they did identify guidelines that they mentally follow. Note, however, that each person only uses a subset of the following guidelines.

- Write requirements to handle exceptions, when they are important.
- Write testable requirements.
- Write requirements statements as active sentences.
- State only one requirement in each requirement statement.
- Do not allow requirements statements to include TBDs.
- Do not allow requirements statements to include weak phrases.

One person thought the biggest benefit of requirements inspections are that they help to build a shared understanding of the desired system among team members. Another person indicated that his project is suppose to perform a review of requirements, but admitted that the team does not do so today. However, this same person stated that his team does thorough systems testing, including examination of user documentation.

Management. Software requirements baselines and derived configurations are not defined or managed in a disciplined manner, although requirements documents are published. Furthermore, requirements are not used to set schedules or delivery dates. Instead, customers always specify schedule and delivery dates, although in multi-mission projects software development tends not to be date-driven. That is, projects tend to slip the dates to do the job right. In addition, people that manage or oversee the development of requirements typically monitor only the requirements development schedules. One person indicated that the use of templates would help to eliminate stylistic issues that make monitoring of requirements definition difficult.

Several practitioners attempt to use tools to help manage requirements. System engineers primarily maintain requirements in DOORS, whereas most software engineers document requirements using Word and Excel. A few software engineers also use RequisitePro. One software team uses Bugzilla, a change management tool, to manage requirements, which is an appropriate, yet novel, application of a change management tool. Each change request entered in Bugzilla represents an addendum to a statement of work and identifies one new capability to add to the system.

Finally, requirements are always uniquely identified, whether through human or automated means, and histories are generally maintained through annotations within these tools.

General weaknesses. Several weaknesses in the process of defining requirements were noted.

- Requirements engineers and managers have little insight on how to prevent the introduction of unneeded features or capabilities. A couple of people wrongly assume that prototyping and continually talking to stakeholders will prevent this from happening.
- It is unclear how requirements engineers decide to allocate specific requirements to subsystems. Sometimes a software requirements engineer will negotiate with his counterpart on the hardware side, although software requirements engineers largely seem to perform this task based on personal knowledge. One person tries to use a formal reasoning process identifying risks and outcomes of cascading decisions, but expressed difficulty in trying to describe the process to others.
- Requirements are not necessarily used as a basis for design, and designs are rarely traced to requirements. However, at least one person has attempted to do this in Excel. Similarly, requirements seldom are used as a basis for testing. However, one person stated that the customer specifies demonstrations that must occur at specific times during development. Each one of these demonstrations tests a specific set of requirements. Thus, these demonstrations provide a good mechanism to monitor the ability of software personnel to satisfy expectations of the program office personnel.

Tools. Two requirements engineering tools used to develop requirements at JPL are DOORS and RequisitePro. RequisitePro is generally easy to use, can import Word documents, and performs impact analysis using traceability matrices. RequisitePro also provides extremely flexible report generation capabilities since output may be directed to Word, Excel, raw text files, and formats supported by Visual Basic. However, this flexibility requires programming in Visual Basic. Its drawbacks are that its initial set-up must be well thought out and it does not link well with design tools, such as Rational Rose. However, RequisitePro can be used to link requirements to scenarios in Rational Rose. The tool does not help with the analysis of requirements statements, although it does provide various metrics, such as the number of changed requirements per unit of time.

The primary benefit of DOORS is that it captures requirements in a database management system. Its major deficiency is that it is difficult to create documents in the formats that meet the needs of requirement engineers. Within DOORS, traceability is captured through links, although establishing traceability can be a tedious process (as with RequisitePro). In addition, requirements engineers need to plan how they set-up the initial document hierarchy because it is difficult to restructure it later (as with RequisitePro). Further, few people knew how to allocate requirements to multiple components within DOORS, although DOORS does permit this. The result of this lack of knowledge on the part of requirements engineers is that more than twenty-five percent of the requirements in four examined requirements documents are duplicates of one another. Finally, the users complained that using Virtual PC to run DOORS on a MAC is horribly slow.

Another person indicated that existing requirements tools are not flexible enough to meet unanticipated needs, which often occur. This person would like to have a repository with a predefined model of software artifacts, as well as methods for defining new views of the repository contents, both of which could be extended by his team. Echoing the finding of a previous study [24], this team feels that existing requirements engineering tools are not very helpful.

Miscellaneous issues. Several issues were also raised in discussions with the interviewed people. The important ones follow.

- The specification of requirements must improve before they can be contractually binding on subcontractors.
- Common spacecraft architectures are possible and it is hoped that MDS helps to provide this.
- Operational requirements are generally defined too late, which cause perturbations in schedules and development efforts. This occurs because project personnel initially concentrate on launch activities and ignore operational activities.

- After the definition of an initial requirements baseline, most missions delete about five percent and change about ten percent of the initially defined requirements. In addition, the number of requirements defined after the initial baseline is about five percent of the number of requirements in the initial baseline. Thus, this is consistent with requirement volatility in industry, which runs about twenty-five percent in total [23]. Multi-mission development efforts, however, demonstrate much fewer requirements changes.

2.3. Document Examination Findings

One Level 3 and three Level 4 requirements documents were examined. No guidance or instruction was provided to the personnel that defined the requirements for one of these documents. The author does not know whether any guidance or instruction was given to the teams that developed the remaining three requirements documents. However, such training has only recently been provided by JPL, and no evidence was found that indicates any training has been provided in the past.

The Level 3 document was the basis for the derivation of one Level 4 document. For this pair of documents there are 380 written Level 3 and 4 requirements. Of these, 51 Level 4 requirements exactly restate Level 3 requirements. Another 49 Level 4 requirements also restate the Level 3 requirements except that the system name is replaced by the subsystem name. That is, a new Level 4 requirement is defined for a subsystem, instead of allocating a Level 3 requirement to the subsystem. The consequence of restating requirements is that it is more difficult to perform impact analysis, estimate development costs, and reliably compute earned value.

In the second Level 3 requirements document, there are 83 written high-level requirements. Few, if any, of these were non-functional requirements. None of the high-level requirements were refined into lower-level requirements. Consequently, many of the requirements are vague and cannot be verified. Most, if not all, of these requirements need to be elaborated. Furthermore, many requirements were not concisely stated because they included rationale in the requirement statements. This is not to say that rationales should not be included as part of a definition of a requirement; they should be. However, the rationale should be separately stated from the statement of a requirement.

The third Level 4 requirements document defines 504 written Level 4 requirements. Several requirements were ignored by the software team. For example, It was required that “the Avionics subsystem shall use Systeme Internationale - Metrics for all phases of the mission.” However, the software team claimed there was not time to provide such capability and that, instead, it would test and inspect that the intent was met. In addition, this document suffered from 5 redundant requirements, and many more had been removed previously.

The combined 967 requirements suffered in many ways. Following is a listing of the noted issues and the number of times each was observed.

- 223 requirements are defined using weak words and phrases.
- 83 requirements define multiple requirements.
- 53 requirements are not stated concisely or simply.
- 37 requirements are incomplete.
- 23 requirements use negations.
- 9 requirements exclude higher-level requirements.
- 7 requirements state definitions, but are not requirements.
- 3 derived requirements do not follow from their parents.
- 2 requirements were not elaborated.

Furthermore, there are numerous instances of the following issues — so many that the author ceased counting them!

- Requirements being placed on a design, instead of the implementation.
- Vague and imprecise requirements.

- Derived requirements seemly inconsistent with higher-level requirements.
- Passively stated requirements that did not indicate the system component responsible for fulfilling a requirement.
- Rationales included in requirement statements.

The author also observed the following.

- Requirements often allocate resources to derived requirements, which can be a source of constraint violations although none were noted in the examined requirements document.
- Data dictionaries were not specified in the requirements documents.
- In one case, the development team satisfied a requirement, but feels that "it should never be used during flight because there are too many interactions whose consequences cannot be predicted!"
- Numerous requirements were missing, especially those that could address anomalies and exceptions.

3. Requirements Engineering Practices

The following practices are proposed to help requirements engineers define good requirements. These practices represent industry *best practices* or are new practices proposed to overcome noted deficiencies in examined requirements documents. Many of these practices are used by those interviewed.

3.1. Elicitation Practices

Requirements elicitation is the process of identifying and consolidating various stakeholder needs [7]. A *stakeholder* is a person affected by a system, has an investment in it, or can affect its success. Typical stakeholders include customers, operations personnel, regulators, software engineers, systems engineers, test engineers, and users. A *stakeholder need* is a business or operational problem that should be eliminated or an opportunity that should be exploited. Following is a discussion of several important practices for elicitating requirements.

Identify and involve stakeholders. It is important to identify stakeholders because they have their own perspective, agenda, priorities, drivers, and unique information concerning their environment, project feasibility, and solution strategies [20, 27]. By gathering and disseminating this diverse information, all the stakeholders can share a common vision, from which they can set realistic expectations and develop high-quality requirements [15]. To develop high-quality requirements, requirements engineers must identify the stakeholder needs and involve them in the definition of a concept of operations and requirements. While doing this, requirements engineers must periodically ensure that operational concepts and requirements are consistent and feasible by integrating them and resolving conflicts as they arise [7, 20].

Stakeholders should be involved in the definition of operational concepts and requirements for three key reasons. First, project failure often occurs because developers do not acquire a detailed understanding of the user's needs, and one way of overcoming this is by their active participation [19, 21]. Second, productivity is about fifty percent higher when there is a high level of participation by the customer in specifying requirements [43]. This is because frequent communication between stakeholders and developers improves the stability of requirements [46]. In fact, the frequency of communication with the stakeholders is more important than using a requirement definition methodology, performing inspections, or having stakeholders on requirements definition teams [46].[1] Third, stakeholder participation results in less rework, avoids the introduction of unnecessary features, and better satisfies the needs of the customers [20]. Without the participation of stakeholders, important needs may be ignored, and stated requirements may be incorrect, inconsistent, or ambiguous.

1. This study also indicated that organization and project size were not critical factors that contributed to requirement volatility.

Identify the reason for developing a system. Identify the purpose for creating a new system [27]. The product purpose is the highest-level customer requirement that identifies one or more business needs [37]. All other requirements must support these needs. To identify the purpose of a product, consider the following questions. Has a new business opportunity arisen that necessitates a change? Will a new system improve the current operation? Will a new system implement new functionality? Has user needs, missions, objectives, environments, interfaces, personnel, or something else changed? What are the shortcomings of the current system, if one exists, that necessitate a new system? If no system exists, then justify why one is needed, as well as the proposed features. In sum, systems should solve only those problems that provide a business advantage [37].

Define a clear, crisp project vision. A vision statement should explain what a product will and will not do, and describe the benefits, goals, and objectives that the product will satisfy [20, 22, 39]. Such a vision will help a team focus on creating an outstanding product with clear benefits. To determine whether a vision is clear, pick a few individuals from a variety of areas on a project and ask each one to briefly outline the project vision. If each person selected cannot immediately identify, in just a few sentences, the key goals and customers for the project, the project is in trouble. This may occur because the vision has not been well communicated to the team, or the team does not agree with or believe in the vision. Whatever the cause, the lack of a shared vision is a fundamental flaw because its personnel lack the overall guidance to assess the merit of features and prioritize the correction of defects. Thus, a project vision helps to resolve issues before requirements are written and shortens the requirement review process.

Define a glossary. A glossary uniquely defines the meaning of key words and phrases, which helps to avoid the misinterpretation of requirement statements. It also permits requirements to be more concisely stated.

Identify applicable operational policies. Operational policies constrain a system by providing guidance that constrain decision-making activities by imposing limitations on the operation of a system. Typical kinds of policies specify the required skills and characteristics of operational personnel, how they use a system, its operational modes, periods of interactive or unattended operations, and the type of processing it performs.

Identify user roles and characteristics. Accurately identify the general characteristics of the users and the way they interact with a system [7, 39]. General characteristics include the responsibilities, education, background, skill level, activities, and modes of operation of each user role. Formal and informal interactions among the various user roles also should be described, if they are relevant to the operation of a system. Especially important are the interactions among user groups, operators, and maintainers. Normally, users fulfill several roles, each requiring a common set of features and performing similar types of operations.

Describe systems similar to the "to be" system. Such systems may be those that other organizations have developed or an existing system that is being replaced by a new one. Regardless, each description should be written using terminology understandable by a typical user. More specifically, it should be free of computer jargon. Graphical representations of any kind can be used as long as they concisely describe the behavior of a system and distinguish between automated and manual processes. Each description should include the following attributes or characteristics [22, 27].

- The provided capabilities, functions, and features.
- The adopted strategies, tactics, methods, and techniques.
- The implemented operational modes, especially degraded and emergency modes.
- Existing operating environment characteristics and interfaces to external hardware and software systems.
- All system components and their interconnections.
- All known current operational risk factors.
- Actual performance characteristics, if known. Such characteristics should consider peak and sustainable speed, throughput, and volume requirements.

Identify all external interfaces and enabling systems. An enabling system is anything that a system needs to perform its mission but is external to it. For example, an enabling system for JPL projects is the Deep Space Network. Typical external interfaces include other software, databases, and hardware, such as input and output devices [39].

For each interface and enabling system, describe its purpose, source, format, structure, content, and method of support [20]. Use context diagrams to identify the external interfaces and enabling systems, with arcs that indicate events, not data flows [20, 39]. Defining external interfaces and enabling systems helps to expose potential problems by revealing system boundaries, which are typically a major source of defects. When identifying external interfaces and enabling systems, a requirements engineer should determine how they can adversely affect a system across the interface, as well as how they can change. For those that change, the risk associated with each change should be managed.

Define a concept of operations. An operational concept is a description of how a product is used and what it does in a typical day [20, 27]. Over time, a concept of operations should be refined to provide more detailed information. Operational concepts generally describe numerous scenarios, but seldom more than fifty of them.

Scenarios identify what a system does, how it operates or will operate, and how it is used or will be used. Each *scenario* identifies a specific situation that occurs in an operational environment. As such, a scenario is a description of one or more end-to-end transactions involving the system operating within its environment. Each transaction describes one or more actions and outcomes, of which exceptions are legitimate outcomes. In addition, each scenario should describe the state of a system before entering it, information about other activities occurring at the same time, the state of the system after completing the scenario, and the users that are interested in the functionality described by the scenario [27]. At a later time, the responsibility for performing the operations implied by the scenario will be assigned to a system component. Common scenarios should cover the following [20].

- The development, verification, deployment, installation, training, operations, maintenance, upgrading, and disposal phases of the life cycle.
- The viewpoints of all the stakeholders, including those of the developers, inspectors, testers, end users, customers, trainers, and maintainers.
- Nominal operations and environments.
- Off-nominal operations and environments, such as extreme and hazardous conditions.
- The expected inputs and outputs, the non-occurrence of expected inputs and outputs, and the occurrence of incorrect inputs or outputs of all interfaces.

Scenarios should be developed by the requirements engineers with the assistance of users [27]. The users identify weaknesses or flaws in the scenarios and the requirements engineers document these problems and ask questions about user actions, how tasks are performed, and what would happen if alternative approaches were taken.

Each scenario should be categorized along several dimensions. The relevance of the scenario should be defined as being either essential, desired, or optional. The technology risk associated with each scenario should be identified as being high, moderate, or low. Further, analysts should assess the confidence that they have in the estimated effort for each scenario as either high, moderate, or low. Later, the analysts should identify the dependency among the scenarios, estimate the time and cost to realize each one, and assign them to development cycles. Those scenarios with the highest priority and greatest risk should be developed first. Guidelines for determining the proper granularity of a scenario follow.

- The estimated cost for realizing a scenario should use at most five percent of the available funds to develop the system. If the scenario is larger than this, the scenario should be refined into two or more scenarios.
- The estimated cost for realizing a scenario should use at least one percent of the available funds to develop the system. If a scenario is smaller than this, the analyst should consider whether its identification is constructive.

Following this guidance will result in the generation of about thirty to thirty-five scenarios. If more than fifty scenarios are identified and these guidelines were followed, then the specified system should be decomposed into two or more systems.

The benefits of scenarios are many. They help stakeholders better understand a system, thus eliminating the most common problem that occurs during the definition of requirements — not fully understanding the desired or necessary behavior of a system [11, 36]. As a result, this shared understanding helps to resolve requirement debates [41] and provides a basis for early requirement validation [20, 27]. In addition, scenarios can be used to reduce the gap between users needs and expectations, assign security policies, analyze project risks, and develop project plans.

Several studies demonstrate the usefulness of scenarios. One experiment showed that fifty-eight percent of requirements changes (defect corrections) were derived from analyzing scenarios and twenty-eight percent of the questions that occurred while analyzing requirements could be answered only by analyzing scenarios [35]. Most of the discussions resulting from scenarios raised possibilities that existing requirements did not cover, rather than considering options or arguing over decision rationales. In another study, scenarios and usability testing reduced the number of usability defects by seventy percent and improved customer satisfaction [28]. In sum, operational scenarios elaborate the problem domain so that personnel better understand a planned system, develop a shared vision of the “to be” system, and develop a better software system. Some assert that the most important thing to do in software development is to study potential user tasks and write down the results as scenarios [28].

Emphasize the definition of vital non-functional requirements. When defining requirements, requirements engineers should pay particular attention to operational, performance, user interface, and site adaptation requirements since these are extremely vital to the successful operation of a system, and are often overlooked [20]. When operational requirements are defined for a product, an organization will spend less time training its users and its users will usually spend less time performing tasks. Similarly, when requirements engineers identify specific performance requirements, project personnel can select appropriate hardware, technologies, and algorithms and validate a system's performance. Important performance requirements include the speed, availability, response time, and recovery time of various software functions [39]. Requirements engineers also should identify screen formats, page layouts, the content of reports and menus, and the availability of programmable function keys [39]. One study found that prototyping user interfaces is one of the three most important practices that can be performed during the requirements definition phase because it lessens the number of omitted requirements before product development [28]. Finally, requirements engineers should define the requirements for any data or initialization sequences specific to a given site, mission, or operational mode and specify those features that should be modified to adapt software for a particular installation [39].

Include specific quality and reliability targets in the requirements. An organization must define the minimum level of quality and reliability that is acceptable [20, 27]. If not, it runs the risk of producing an inferior product that does not meet the needs of its customers. Similarly, organizations should define quality at least in terms of incoming defects after deployment, mean time between failures, and defect removal efficiency.

Segment requirements into key categories. Individual requirements should be linked to one or more critical requirements categories such as safety, security, and performance, which permits people to identify whether important types of requirements have been ignored [20].

3.2. Analysis Practices

Requirements analysis is the process of identifying the appropriateness, completeness, quality, and value of a set of requirements. Following is a discussion of several important practices for analyzing requirements.

Develop conceptual models. A conceptual model is an abstraction used by a software engineer to understand a system before its construction [28]. Abstraction involves the selective examination of certain aspects of the real world and the desired system, usually at a reduced level of fidelity, that isolate important aspects while ignoring other, seemingly unimportant ones. Of course, what is and is not important at any time is dependent on the needs of the sys-

tem users. By their nature, abstractions are incomplete and inaccurate. Hence, a good model of a system represents the aspects of a problem critical to its solution.

Although modeling is an ancient concept, correspondence modeling in the software profession is relatively new. *Correspondence modeling* involves building models so that there is a one-to-one correspondence between artifacts of the real world and those of the modeled world. The field of artificial intelligence calls correspondence modeling model-based reasoning, whereas the software profession calls it object-oriented analysis and design. There are several schemes to describe models. One of the most popular schemes is the Unified Modeling Language (UML) [4].

The modeling process involves several important activities. First, important concepts of the real world must be identified by examining the scenarios of the concept of operations, external interfaces, and enabling systems. These concepts should be defined using the standard terminology of the problem domain and named using a single noun or an adjective and a noun. Second, the relationships among the concepts should be identified. To identify class relationships, compare each pair of classes and determine the relationship of one class to the other. If related, determine if there is more than one relationship. Third, identify those concepts that are closely coupled with one another and group them into units. Such units typically characterize a problem domain or an environment. Fourth, define the attributes of a concept; one for each unique type of feature. Finally, define the behaviors that a concept provides to other concepts.

The benefits of developing a conceptual model are several. First, software engineering personnel can use a model to check the specifications and requirements of a new system. Second, software organizations can use a model to educate people and provide them with a general understanding of the structure and operation of the modeled problem domain. Third, programmers can derive working systems directly from a model. Fourth, modeling makes it easier to inject faults into software simulations, which eases debugging and testing activities. Fifth, models help to identify development risks. Sixth, conceptual models tend to be stable over time and can form a basis for software reuse.

Record assumptions. Assumptions, once written, should be confirmed or corrected before too long [36]. If an assumption cannot be validated, it is a risk that must be analyzed and monitored [20]. (See “Analyze risks” on page 15 and “Monitor the status of software requirements following a defined procedure” on page 20.)

Allocate requirements in a top-down manner. Allocate requirements to both system components and development cycles [20]. Allocating requirements to system components is the first step in planning the development of a system. Allocating requirements to development cycles is the second step, but this can be done only after requirements have been prioritized (See “Prioritize software requirements” on page 14.).

Prioritize software requirements. Prioritizing requirements based on cost, dependency, and importance yields many benefits [7, 19, 20, 27]. First, an organization can use prioritization to select Commercial Off The Shelf (COTS) products that best satisfy the most important requirements [42]. Second, prioritization provides a precise technique for selecting the most important requirements to implement, which may lead to a reduction of the number of implemented functional requirements [28]. Third, priorities can be used to define the capabilities of each software release or build.

The most common way that people prioritize software requirements is to use a numerical assignment technique, where a person identifies the absolute importance of each requirement one at a time. Unfortunately, this approach generally results in too many requirements being highly ranked. An alternative approach is to identify the relative value between every pair of requirements, which means that a person makes $(n^2 - n)/2$ comparisons, and use that information to rank priorities. The Analytic Hierarchical Process (AHP) [38] is based on this approach and uses the eigenvalues of the comparison matrix to represent the priority of each requirement. The benefit of AHP is that its redundancy makes it less sensitive to errors and permits the measurement of the error, which is calculated as $(\lambda_{max} - n)/(n-1)$, where λ_{max} is the maximum principal eigenvalue and n is the number of requirements. A variant of the AHP is called the Incomplete Pairwise Comparison (IPC) method, which randomly eliminates half of the comparisons without adversely affecting the result [6, 17, 18]. One case study indicates that these pair-wise comparison techniques are more accurate, efficient, and informative than the naive numerical assignment technique and that there

is greater consensus among analysts when they use it [25]. Yet, the number of comparisons are so large that to do pair-wise comparisons among system requirements as just described is not practical. However, an alternative approach is to use a scheme where the importance of subsystems or scenarios are prioritized, followed by a prioritization of the requirements allocated to those subsystems or scenarios.

Capture requirement rationales and discussions of them. The rationale for each requirement must explicitly state why a requirement is needed. When identifying the rationale, one often will find that the rationale is really the statement of a desired functional requirement, whereas the original statement is simply one realization that satisfies it [20]. So, identifying rationales is one way to find the true functional requirements. In addition, rationalization allows other people to annotate a requirement with supportive or contradictory information that can identify unrecognized issues and offer alternatives [34]. Thus, the benefit of documenting the rationales of requirements is that they generally improve the quality of defined requirements, shorten the review of requirements, improve impact analysis, and capture corporate knowledge [20].

Analyze risks. Common requirement risks include unknown or changed regulations and interfaces, missing requirements, and technical feasibility, cost, and schedule uncertainty. Consequently, when identifying requirements, engineers should identify requirements that do not have these risks, as well as identifying those requirements that could have significant impact [20]. When risk has been identified, project personnel need to perform feasibility studies and risk assessments.

3.3. Specification Practices

A *requirements specification* should establish an understanding between customers and suppliers about what a product is suppose to do, provide a basis for estimating costs and schedules, and provide a baseline for validation and verification [39]. Specifications are most often written in natural language because this is the only form of communication that is currently contractually binding [44]. The use of natural language, unfortunately, does not offer enough structure to produce good requirements. Hence, practitioners should adopt practices to improve the specification of requirements. Some of these practices follow. However, following these practices still does not guarantee that good requirements will be produced. The development of good requirements requires a lot of analytic thought. Attempting to specify a rationale for a requirement is one way to encourage such thought. (See “Capture requirement rationales and discussions of them” on page 15.) When identifying a rationale, one often finds the true requirement.

Uniquely identify each statement and ensure that each statement is unique. Traceability should be both forward to higher level requirements and backward to lower level ones. Also, each requirement should be stated only once. Unfortunately, experience has shown that many requirements are duplicated. (See “Document Examination Findings” on page 9.)

Differentiate between requirement, goal, and declaration statements. A typical problem with requirements is that their authors use words other than “shall” to specify requirements — words which are not legally binding and seldom explicit [44]. To ensure clarity, requirements must use “shall” to specify contractually binding requirements, “will” to specify facts or declare purpose, and “should” to identify goals or non-mandatory provisions [20]. Consider the following example.

EXAMPLE 1. An attempt at ignoring requirements.

Statement:	The [subsystem] design shall *be consistent with and responsive to* all design requirements in JPL D-19272.
Issue:	The JPL D-19272 document specifies several design requirements. As such, a design must comply with those requirements. The phrase “be consistent with and responsive to” implies that compliance is not necessary, which opposes the definition of a requirement. If the requirements have not been waivered then the statement should be written as follows.
Revision:	The [subsystem] design shall *meet* all design requirements in JPL D-19272.

Define a verification method for each requirement. All requirements should be verified by test, demonstration, analysis, or inspection [14, 20]. Identifying the verification method helps to ensure that each requirement is verifiable and practical, and that resources are available to verify it. When functional tests are used, identify them in conjunction with requirements definition to prevent their omission and to ensure that the functional tests validate the intent of the system requirements.

State the needed requirements without specifying how to fulfill them. Avoid specifying design decisions in requirements specifications unless the intention is to impose a constraint on the design activity [20]. Such a constraint would be justified by the use of an external or enabling system, an existing product, or a desirable design practice. For instance, requirements could specify the use of specific design frameworks and patterns to help create robust system architectures.

State only one requirement per requirement statement. Compound requirements make the changes to them harder to analyze and trace. An example of a compound requirement follows.

EXAMPLE 2. An unclear, compound requirement.

Statement:	The [system] shall generate and deliver data to the spacecraft as specified in [a document].
Issue:	Two requirements are stated: one to generate data and the other to deliver it.
Issue:	It is unclear whether descriptions of the data or the meaning of generating or delivering data is defined in the identified document.
Revision:	The [system] shall generate the data specified in [a document]. The [system] shall upload the data specified in [a document] to the spacecraft.

State requirements as active sentences. Sometimes goals are stated, and it is not clear what is responsible for achieving it. For example, consider the following requirement.

EXAMPLE 3. A passively-stated requirement.

Statement:	The [subsystem] shall not be exposed to direct sunlight when the [external system] is unpowered.
Issue:	The requirements statement does not identify the system component that is responsible for ensuring that the [subsystem] is not exposed to direct sunlight.
Revision:	The [system] shall ensure that the [subsystem] is never exposed to direct sunlight when the [external system] is unpowered.

Do not use negations within requirements statements. Negations are typically harder for people to understand and may cause the incorrect interpretation of a requirements statement. Another reason for avoiding negations is that it is often difficult or impossible to test such statements. For instance, consider the following requirement.

EXAMPLE 4. A requirement stated in the negative form.

Statement:	The [system] shall provide *no* less than 4 MB of data storage in non-volatile memory.
Issue:	The requirement is stated in the negative form.
Revision:	The [system] shall provide *at least* 4 MB of data storage in non-volatile memory.

Avoid using words that end in "ly". Such words (e.g., quickly, optionally, timely) are generally imprecise and result in poorly written requirement statements [20].

Avoid using weak words and phrases. Such words and phrases allow the expansion or contraction of requirements beyond their intent. Sometimes these weak words and phrases occur when using indefinite pronouns and unqualified adjectives and adverbs. Table 1 identifies several words and phrases that often appear in requirement specifications and should be avoided.

TABLE 1. Words and phrases to avoid using within requirements statements.

• accommodate	• capability of	• normal
• adequate	• capability to	• not limited to
• and/or	• easy	• provide for
• as a minimum	• effective	• robust
• as applicable	• etc.	• sufficient
• as appropriate	• if practical	• support
• be able to	• maximize	• these
• be capable of	• may	• this
• can	• minimize	• when necessary

The following two examples illustrate flawed requirements statements using a weak word and phrase identified in Table 1.

EXAMPLE 5. A statement identifying an ability, but not a requirement.

Statement:	The [system] shall *be able to* acquire the [data] with an instantaneous field of regard variation of 3 mrad per second.
Issue:	An expression of an ability to perform is different that a requirement to perform.
Revision:	The [system] shall acquire the [data] with an instantaneous field of regard variation of 3 mrad per second.

EXAMPLE 6. A weakly-stated requirement.

Statement:	The [subsystem] shall *support* 21 potentiometers with an absolute angular range of 335 degrees.
Issue:	The use of a weak word - support - does not clearly identify the requirement. Will the subsystem provide these potentiometers? If not, will the subsystem provide power for them, measure their output, or provide physical space for them?
Revision:	The [subsystem] shall provide data storage for 21 potentiometers. The value of each potentiometer will be defined as any value between 0 and 335, inclusive, and accurately representing 6 significant digits.

Write complete statements. A requirements specification should fully describe each requirement; hence, no requirement should be identified as *to be determined*. However, if an assumption must eventually become a requirement then state it as a requirement, where the portion of the requirement that is assumed is enclosed in square brackets [20]. When such an assumption is made, the requirements specification must describe the conditions causing it, who is responsible for its elimination, and when it must be eliminated. Following is one example of an unfinished requirement and its hypothetical completion.[1]

EXAMPLE 7. An unfinished requirement.

Statement:	The flight software shall update the event timers [[as needed]] during EDL.
Issue:	This requirement is incomplete and no one even assumed how often the event timers needed to be updated. This lets the developer update them as often as he determines is appropriate, which may be never or continuously.
Revision:	The flight software shall update each event timer every 10 milliseconds during EDL.

1. Double brackets are used to indicate the assumptions since a single bracket is being used in this document to represent a non-terminal language token.

Write statements that clearly convey intent. A requirements specification should accurately and precisely reflect the intent of its author [20]. Following is one example of a common type of problem. Namely, these requirements affect the design of a system, although the intent was to affect an implementation.

EXAMPLE 8. A requirement imposed on a design.

Statement:	The [subsystem] shall *be designed* to be used and operated at a single processor speed setting of 20 MHz.
Issue:	This requirement is imposed on the design of a subsystem, instead of on the delivered system.
Revision:	The [subsystem] shall use a processor clock cycle of 20 MHz.

Do not exclude higher-level requirements. For example, consider the following requirement.

"All [a subsystem] flight software shall be integrated by the Flight Software subsystem team."

This requirement was derived from the following requirement, and wrongfully eliminates the testing requirement.

"The [system] shall perform integration and testing on all delivered hardware and software."

Fully specify all requirements. A requirements specification should be complete [27]. It must describe all functionality of a product, all real world situations that a product will encounter and its responses to them, and the responses to all valid and invalid inputs. Following are several kinds of requirements that should be specified in a requirements document.

- Specify a nominal value, precision, accuracy, and allowable range for all data elements and inputs. This ensures that applications do not violate interface requirements. It also prevents invalid data that is out of range or beyond the representational capability of the hardware.
- Specify the minimum and maximum time that a process or thread should wait before the first input. If a software system has not received input within a reasonable amount of time after startup, it most likely has not properly initialized itself and it should take corrective action to repair itself or notify an operator of the problem [30]. Similarly, if the time of receipt occurs too soon after start up, proper initialization may not have occurred.
- Specify time-out conditions and contingency actions for all input and output streams. One contingency action that should generally be specified is the length of a time-out. For example, a program can buffer inputs to prevent sending excessive outputs. This, however, could result in either a short- or long-term buffer overrun. That is, a system could receive a nearly instantaneous high number of inputs or it could receive a sustained and higher than expected input rate that causes the accumulation of inputs. If a system does not properly handle these situations, catastrophic failures may occur.
- Specify the minimum and maximum expected execution times for time-dependent computations and contingency actions for their violation. If a computation uses less time than expected, a failure may have occurred. If a computation has taken longer than expected then the system may be in a deadlock situation or the system may be waiting for input from a defective component, module, or external system. In either case, something may have gone wrong, and adherence to this practice tends to improve system reliability and safety.
- Specify the conditions where software systems can return to normal processing load after encountering an anomaly. Adherence to this practice prevents a system from reentering an anomalous situation. After detecting a capacity violation, for example, the system should not begin normal processing too quickly because the circumstances that caused the capacity violation may still exist and cause the system to violate the capacity limitation once again. Note that adherence to this practice implies the identification of all expected anomalies before design begins.

Identify the relationships among requirements. As each requirement is written, it should be linked to operational concepts or higher-level requirements and other work products [19, 20]. While doing this, requirement engineers should continually ask whether lower-level requirements satisfy higher-level requirements or whether any lower-level requirements dictate internal interfaces. Also, lower-level requirements should have explanations describing

why a requirement is traced to multiple higher-level requirements or not traced to one requirement. In the later case, a requirement that is not derived from a higher-level requirement may indicate that it does not meet business objectives or it could indicate an undefined higher-level requirement [20]. On the other hand, if a requirement is linked to several requirements, any change will probably have a major impact on the system. Hence, such requirements must be carefully assessed and monitored. Following are some common problems that have been observed in requirements documents.

- Derived requirements do not follow from their parents. Therefore, it is important to link derived requirements to higher-level requirements so that the derivation can be validated. For example, in one requirements document "The [system] mass shall not exceed 7.25 kg including thermal control hardware" was linked to the derived requirement, "The [subsystem] shall respond to its functional and hardware faults."
- Requirements are not elaborated. For example, the requirement statement, "The [system] software shall provide time management services" was never refined to identify the specific services to be provided.
- Derived requirements seem inconsistent with higher-level requirements. For example, the source requirement, "The latency of [subsystem] measurement delivery to [system] flight software shall not exceed 50 msec" seems to be violated by the derived requirement, "The [subsystem] shall deliver measurements to [system] not to exceed a latency of 20% of a 500 msec [system] cycle."

To identify these problems, requirements engineers should answer the following questions.

- Are all functions, structures, and constraints traced to requirements, and vice versa?
- Have the requirements been allocated to functions of the system?
- Do the requirements indicate whether they are imposed or derived?
- Have the derived design goals and implementation constraints been specified and prioritized?
- Is each requirement uniquely defined?

Write consistent statements. A requirements specification must be consistent [27]. Hence, a requirements engineer should try to resolve conflicts among requirements or characteristics of real-world objects. Similarly, terminology must be consistently used in a requirements specification. One way to achieve this is to use a data dictionary. Finally, performance requirements must be compatible with the required quality features (e.g., reliability). Heuristics for finding conflicting requirements include examining requirements that use the same data, requirements of the same type, and requirements using the same scales of measurement.

3.4. Validation Practices

Requirements validation attempts to demonstrate that artifacts satisfy established criteria and standards. For requirements engineering, validation attempts to show that requirement statements adhere to the specification practices identified earlier.

Inspect requirements using a defined process. People should review software requirements following a defined procedure to validate that the requirements accurately reflect the needs of the stakeholders [7, 8, 19]. This defined procedure should include asking the questions contained in Appendix B [20, 27, 31]. The review process should proceed in six basic steps [20]. Although this review process will not catch all defects, developers usually have the knowledge to interpret most requirements correctly [28].

1. Review for editorial content by two people. Resolve issues.
2. Review for goodness.
   - Two people should review the requirements using the identified practices. The chief architect should be one of the reviewers because he or she is the one person who has an intellectual understanding of the entire software system. The chief architect understands the interactions of the system components and is responsible for maintaining the conceptual integrity of the entire system, which is vital to achieve product quality.

The lack of such a person or the unavailability of such a person for an inspection is an indication that the development effort has problems.

- The reviewers should use a defect-based inspection technique when requirements are created, have significantly changed, or a change affects the next cycle of development or a pending development effort.
- The reviewers and project manager or team lead must resolve requirement issues and document each resolution. Afterwards, the original reviewers should examine the changed requirements.
- If significant change is needed or the quality objectives are not met then the requirements should be reviewed a second time. In this case, use two different reviewers.
- Use the Chao and Jackknife estimators to determine whether a second review is necessary.

3. Review for content by all stakeholders [19]. Resolve requirements issues and document each resolution.
4. Review for appropriateness all selected reused and third-party software by personnel with intimate knowledge of such software [28]. Resolve requirement issues and document each resolution.
5. Review for risk by two senior engineers and managers.
6. Review for editorial content by two people.

3.5. Management Practices

There are a few important practices that managers must perform. They primarily involve planning, monitoring interactions among people, and tracking development progress.

Use bilateral agreements to ensure a common understanding of the requirements and to control requirement creep. Such agreements should identify the responsibilities of the customer and supplier [20].

Monitor the status of software requirements following a defined procedure. The status of each requirement should be available, as well as the change activity for each one. Further, summary data should be collected for the total number of changes that others have proposed, approved, and added to the software requirements.

Measure the number and severity of defects in the defined requirements. These metrics help to identify the quality of the requirements engineering activity so that the organization can improve it [20].

Control how requirements are introduced, changed, and removed. The average project experiences about a twenty-five percent change in requirements after the requirements have been defined for the first system release, which causes at least a twenty-five percent schedule slip [23]. Several studies also have shown that volatility in requirements contribute to the inefficient production of low-quality software [11, 13]. Consequently, requirements should be managed using a defined configuration management process that uses change control boards and automated change control tools that manage each requirement as a separate configuration item [10, 27].[1] Using a configuration management tool permits personnel to identify which requirements have been added, removed, or changed since the last baseline, who has made these changes, when they were made, and the reason for making them. In addition, by maintaining requirements in a configuration management tool, they can be traced to the artifacts that realize them.

4. Requirements Engineering Methods

4.1. Elicitation Methods

The elicitation of requirements should be accomplished by interviewing stakeholders and involving them in the development of scenarios. This involvement may include examining various reports, such as feasibility studies, mar-

1. Such guidance partially justifies a model-based approach to software engineering.

ket analyses, business plans, and analyses of competing products [7, 19], using facilitated group meetings, or observing users work. Two important methods of elicitation are provided by graphical Issue-Based Information System (gIBIS) [9] and Joint Application Design (JAD) [45]. JAD has historically been a manual process appropriate for idea generation and group consensus building, whereas gIBIS is a practical method for documenting ideas.

gIBIS. gIBIS supports constructive communication by making its users think about a framework of issues, positions, and arguments. The benefits of this are several. First, important information is captured in a consistent form so that it can be analyzed or revisited at a later time by matching issues. Thus, open issues are not forgotten and tangents are disposed of quickly. Second, the framework helps to focus thinking on the critical parts of the problem and helps to improve the detection of incomplete and inconsistent thinking. Third, it helps to make assumptions and definitions explicit.

However, this structuring can be disruptive during the early phases of a problem when some amount of brainstorming is usually needed. It also suffers from other weaknesses. First, it does not support other object types, such as goals and requirements. Second, there is no support for resolving issues and reaching consensus. That is, the argumentation process should result in some outcomes (decisions), constraints generated by making decisions, and design objects. Third, gIBIS does not support a hierarchy of issues-positions-arguments. Fourth, it does not permit the identification of old positions, or those that have been superseded by newer ones. Finally, it does not identify those issues that have been most frequently examined.

JAD. JAD helps a software engineering team develop a set of stable requirements, whose rate of change is much less than those more commonly found with other methods for defining requirements. When using JAD, the project requirements are the joint responsibility of representatives of the client, development, and user communities. JAD focuses on the group decision-making process and getting the right people involved from the start. JAD enhances idea generation and evaluation, communication, consensus building, and shared ownership. Consequently, JAD should be used to define purposes and objectives, determine if they are achievable, and gain commitment from the stakeholders [37].

4.2. Analysis Methods

Quality Function Deployment (QFD). QFD is a structured process for capturing the needs of the customer through customer interaction and brainstorming among the various stakeholders [33]. QFD assigns priority to product requirements and minimizes changes since defects are prevented up-front instead of during a down-stream activity. QFD helps to focus decisions on quality primarily by using matrices to discover interrelationships between stakeholder needs, requirements, and design and implementation methods. It also helps requirements engineers to track detailed “what” expectations against “how” realizations. The planning phase of QFD is composed of nine steps.

1. Identify the customers, the product, and the development time horizon.
2. Gather high-level customer requirements and refine each one into a set of actions.
3. Gather final product control characteristics (architecture features) to meet customer requirements.
4. Develop relationship matrix between customer requirements and product control characteristics.
5. Rate importance of features.
6. Compute final product control characteristic competitive evaluation for each control characteristic.
7. List key selling points (requirements) of product.
8. Develop target values for each of the product control characteristics based on key selling points, importance ratings, and current product strengths and weaknesses.
9. Finalize the selection of the product control characteristics.

One experiment showed that capturing quality requirements using QFD is easier than with other methods [12].

5. A Requirements Engineering Process

Requirements definition is a discovery process involving problem analysis and understanding [27]. Problem analysis is the activity where the requirements engineer learns about the problem to be solved, the needs of the users, and the constraints on the problem solution. As such, requirements engineering involves cycles of building up followed by major reconstruction [32]. As requirements are acquired, analyzed, and added to the requirements model, its complexity grows. At critical points, however, the requirements model should be simplified and restructured to reflect a new perception of the requirements problem. Thus, requirements do change, and the process of defining requirements is naturally iterative and opportunistic [2, 3, 7, 19, 27, 34]. The critical points where change occurs are a result of critical, unexpected insights. As a consequence of restructuring the requirements, the problem domain and customer needs are better understood and results in a more elegant requirements model. Notice that this differs from the conventional notion of how requirements are defined.

This behavior results primarily from people trying to cope with complexity, of which there are three kinds: essential, incidental, and accidental. Essential complexity represents the intrinsic understanding of a problem as indicated by a model. This type of complexity grows over time as the problem solution is more fully developed. Incidental complexity involves the complexity created by the representational form.[1] It identifies the poor fit between the structure of the model and the real world. As the model grows, it becomes harder to add new items to the model. The incidental complexity grows exponentially over time. At crisis points, the restructuring of a model reduces incidental complexity significantly. Accidental complexity represents the hidden knowledge of a model that becomes explicit only during reconceptualization that occurs during model restructuring at crisis points. After restructuring, it becomes part of essential complexity. Thus, requirement engineering is insight driven, not systematic.

Most of this volatility should occur at the beginning of a project; at a minimum, volatility should only occur within those requirements where design and implementation has yet to occur. If volatility does not gradually decrease over time, then the project is out of control. Hence, an organization should define a structured process for creating and monitoring requirements that measures requirements volatility [35, 46]. In addition, a requirements definition process should include a disciplined application of scientific principles and practices for elicitating, analyzing, specifying, verifying, and managing requirements [7, 8].

During large system development, requirements engineers may use several iterations to define a system, where each one can correct, elaborate, refine, or remove existing requirements. Once the requirements for a system are fairly stable, then each team that defines the requirements of a subsystem may proceed, while monitoring any changes to the requirements of the system. In addition, each team specifying the requirements of a subsystem must also monitor the requirement definition process of those subsystems that impact the subsystem it is defining.

The general process for engineering requirements follows [20].

1. Scope the product by defining needs, goals, and objectives, mission or business case, high-level operational concepts, customer requirements, constraints, schedules, budgets, authority, and responsibility.
2. Develop an operational concept that describes how the product behaves and is used throughout the product's life.
3. Identify interfaces between a product and the outside world, and clarify its boundaries, inputs, and outputs.
4. Write the requirements.
5. Capture rationale of each requirement and expose assumptions and incorrect facts.
6. Define requirements according to system and subsystem divisions, ensuring that requirements are allocated to the appropriate level and are traceable to higher-level requirements.
7. Assess the verification statement of each requirement, identifying the verification technique and needed facilities and equipment.

1. This is one reason for using domain specific languages, which removes representational details.

8. Format requirements and supporting information so that requirements are easily found.
9. Baseline requirements after they are validated.
10. Iterate as necessary and time permits.

The inputs to the requirements engineering process include the following [27].

- Existing information describing the systems to be replaced or external systems that must interact with the system to be developed.
- Description of the needs of stakeholders.
- Organizational standards that describe development practices, quality goals, etc.
- External regulations.
- General information about the problem domain.

The outputs of the requirements engineering process include the following [27].

- A description of the requirements that stakeholders understand and have agreed on.
- In some cases, a detailed specification of the system functionality.
- A set of models that describe the system from different perspectives.

The requirements users include the following [27].

- Customers, who specify and inspect the requirements.
- Managers, who use the requirements to bid on a system development effort and plan the development process.
- Software engineers, who use the requirements to understand what the system is.
- Test engineers, who use the requirements to develop validation tests.
- Maintenance engineers, who use the requirements to better understand the system and he interrelationships between the subsystems.

6. Summary

This study suggests that JPL personnel could do a much better job, overall, of eliciting, analyzing, specifying, validating, and monitoring requirements. To support an improvement effort, this paper has primarily sought to identify numerous practices that can improve a requirements engineering process and provide a standardized process for defining requirements. The potential impact of implementing these practices can be the construction of stable, precise, unambiguous, and correct requirements that have the potential for significantly reducing project development costs, primarily by reducing the amount of work that must be redone. The key practices may be summarized briefly:

- Clearly state a need for developing a proposed system.
- Use stakeholders in the requirements engineering process, both to achieve a better understanding of the problem domain and to review the correctness of the proposed requirements. Vital stakeholders include the expected users of a system.
- A concept of operations should initially express the problem domain, although a domain model should be derived from it later.
- Specify the various types of non-functional requirements, such as security requirements, in addition to the functional requirements.
- Identify the rationale, involved risk, and verification method for each requirement.
- Write clear, consistent, and precise requirement statements that avoid the use of weak words and phrases.

Following these practices should yield a significant reduction in the number of defects that remain in requirements specifications. Some of the interviewees followed many of these practices, but the examination of existing requirements documents suggests that these practices are not widely used.

Acknowledgements

The author appreciates the contributions of the ten anonymous people he interviewed. Without their input, this work could not have been done. The author would also like to thank Bruce Bullock, Steve Larson, Milton Lavin, and Frank McGarry for their constructive criticism of earlier drafts of this document.

The research described in this document was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

DOORS is a registered trademark of Telelogic. Rational Rose and RequisitePro are registered trademarks of IBM. Excel, Visual Basic, and Word are registered trademarks of Microsoft.

References

[1] American Institute of Aeronautics and Astronautics, “Guide for the Preparation of Operational Concept Documents,” American Institute of Aeronautics and Astronautics, ANSI/AIAA G-043-1992, 1993.

[2] Blum, B. I., *Beyond Programming: To a New Era of Design*, Oxford University Press, 1996.

[3] Boehm, B. and Egyed, A., “Software requirements negotiation: some lessons learned,” *Proceedings of the International Conference on Software Engineering*, 1998, pp. 503-506.

[4] Booch, G., Rumbaugh, J., and Jacobson, I., *Unified Modeling Language User Guide*, Addison-Wesley, 1998.

[5] Brooks, F. P. Jr., *The Mythical Man-Month: Essays on Software Engineering Anniversary Edition*, Addison-Wesley, 1995.

[6] Carmone, F. J., Kara, A., and Zanakis, S. H., “A Monte Carlo Investigation of Incomplete Pairwise Comparison Matrices in AHP,” *European Journal of Operational Research*, vol. 102, no. 3, 1997, pp. 538-554.

[7] Christel, M. G. and Kang, K. C., “Issues in Requirements Elicitation,” Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI-92-TR-012, 1992.

[8] Chudge, J. and Fulton, D., “Trust and co-operation in system development: applying responsibility modelling to the problem of changing requirements,” *Software Engineering Journal*, May 1996, pp. 193-204.

[9] Conklin, J., Begeman, M. L., “gIBIS: A Hypertext Tool for Exploratory Policy Discussion,” *ACM Transactions on Office Information Systems*, vol. 6, no. 4, Oct. 1988, pp. 303-331.

[10] Crnkovic, I., Funk, P., and Larsson, M., “Processing requirements by software configuration management,” *Proceedings of the EUROMICRO Conference*, 1999, vol. 2, pp. 260-265.

[11] Curtis, B., Krasner, H., and Iscoe, N., “A Field Study of the Software Design Process for Large Systems,” *Communications of the ACM*, vol. 31, no. 11, November 1988, pp. 1268-1287.

[12] Elboushi, M. I. and Sherif, J. S., “Object-Oriented Software Design Utilizing Quality Function Deployment,” *Journal of Systems and Software*, 1997, pp. 38-143.

[13] Gibbs, W. W., “Software’s Chronic Crisis,” *Scientific American*, September 1994, pp. 86-95.

[14] Glass, R. L., *Software Reliability Guidebook*, Prentice-Hall, 1979.

[15] Grünbacher, P., Halling, M., Biffl, S., Kitapci, H., and Boehm, B. W., “Repeatable Quality Assurance Techniques for Requirements Negotiations,” *Proceedings of the Hawaii International Conference on System Sciences*, 2003.

[16] Hamilton, D., Covington, R., and Kelly, J., "Experiences in applying formal methods to the analysis of software and system requirements," *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques*, 1995, pp. 30-43.

[17] Harker, P. T., "Incomplete Pairwise Comparison in the Analytic Hierarchy Process," *Mathematical Modelling*, vol. 9, no. 11, 1987, pp. 837-848.

[18] Harker, P. T., "Alternative Modes of Questioning in the Analytic Hierarchy Process," *Mathematical Modelling*, vol. 9, no. 35, 1987, pp. 353-360.

[19] Hofmann, H. F. and Lehner, F., "Requirements engineering as a success factor in software projects," *IEEE Software*, vol. 18 no. 4, July/August 2001, pp. 58-66.

[20] Hooks, I. F, and Farry, K. A., *Customer-Centered Products: Creating Successful Products through Smart Requirements Management*, AMACOM, 2001.

[21] Humphrey, W. S., *Managing technical people: innovation, teamwork, and the software process*, Addison-Wesley, 1997.

[22] Institute for Electrical and Electronics Engineers, "IEEE Guide for Information Technology – System Definition – Concept of Operations (ConOps) Document," IEEE Computer Society, IEEE Std. 1362-1998, 1998.

[23] Jones, C., *Assessment and Control of Software Risks*, Prentice-Hall, 1994.

[24] Kandt, R. K., Kay-Im, E., Lavin, M. L., and Wax, A., "A Survey of Software Tools and Practices in Use at Jet Propulsion Laboratory," Jet Propulsion Laboratory, Internal Document D-24868, 2002.

[25] Karlsson, J., "Software requirements prioritizing," *Proceedings of the International Conference on Requirements Engineering*, 1996, pp. 110-116.

[26] Kelly, J. C., Sherif, J. S., and Hops, J., "An Analysis of Defect Densities Found During Software Inspections," *Journal of Systems and Software*, vol. 17, no. 2, February 1992, pp. 111-117.

[27] Kotonya, G. and Sommerville, I., *Requirements Engineering: Processes and Techniques*, Wiley, 1998.

[28] Lauesen, S. and Vitner, O., "Preventing Requirement Defects: An Experiment in Process Improvement," *Requirements Engineering*, vol. 6, 2001, pp. 37-50.

[29] Lederer, A. L. and Prasad, J., "Nine Management Guidelines for Better Cost Estimating," *Communications of the ACM*, February 1992, pp. 51-59.

[30] Leveson, N. G., *Safeware: System Safety and Computers*, Addison Wesley, 1995.

[31] Lutz, R. R., "Targeting Safety-Related Errors During Software Requirements Analysis," *Journal of Systems and Software*, vol. 34, 1996, pp. 223-230.

[32] Nguyen, L., Armarego, J., and Swatman, P. A., "Understanding Requirements Engineering: a Challenge for Practice and Education," School of Management Information Systems, Deakin University, School Working Papers 2002/10, 2002.

[33] Pardee, W. J., *To Satisfy and Delight Your Customer: How to Manage for Customer Value*, Dorset House, 1996.

[34] Potts, C. and Bruns, G., "Recording the Reasons for Design Decisions," *Proceedings of the International Conference on Software Engineering*, pp. 418-427, 1988.

[35] Potts, C., Takahashi, K., and Antón, A. I., "Inquiry-Based Requirements Analysis," *IEEE Software*, March 1994, pp. 21-32.

[36] Potts, C., "Invented requirements and imagined customers: requirements engineering for off-the-shelf software," *Proceedings of the IEEE International Symposium on Requirements Engineering*, 1995, pp. 128-130.

[37] Robertson, S. and Robertson, J., *Mastering the Requirements Process*, Addison-Wesley, 1999.

[38] Satay, T. L., *The Analytic Hierarchy Process*, McGraw-Hill, 1980.

[39] Software Engineering Standards Committee, "IEEE Recommended Practice for Software Requirements Specifications," The Institute of Electrical and Electronics Engineers, IEEE Std. 830-1998, 1998.

[40] Software Engineering Standards Committee, "IEEE Guide for Information Technology – System Definition – Concept of Operations (ConOps) Document," IEEE Std. 1362-1998, 1998.

[41] Stark, G., Skillicorn, A., and Smeele, R., "A micro and macro based examination of the effects of requirements changes on aerospace software maintenance," *Proceedings of the IEEE Aerospace Conference*, vol. 4, 1998, pp. 165-172.

[42] Tran, V., Hummel, B., Liu, D.-B., Le, T. A., and Doan, J., "Understanding and managing the relationship between requirement changes and product constraints in component-based software projects," *Proceedings of the Hawaii International Conference on System Sciences*, 1998, vol. 6, pp. 132-142.

[43] Vosburgh, J. B. et al., "Productivity Factors and Programming Environments," *Proceedings of the International Conference on Software Engineering*, 1984, pp. 143-152.

[44] Wilson, W. M., Rosenberg, L. H., and Hyatt, L. E., "Automated Analysis of Requirement Specifications," *Proceedings of the International Conference on Software Engineering*, 1997, pp. 161-171.

[45] Wood, J. and Silver, D., *Joint Application Design: How to Design Quality Systems in 40% Less Time*, John Wiley, 1989.

[46] Zowghi, D. and Nurmuliani, N., "A study of the impact of requirements volatility on software project performance," *Proceedings of the Asia-Pacific Software Engineering Conference*, 2002, pp. 3-11.

A. A Process Assessment Questionnaire

Following are twenty-five sets of questions that the author used to assess the requirements engineering process at JPL. These questions were derived from numerous sources that identify requirements engineering issues, best practices, and assessment methods. For each set of questions, the author asked the interviewed person the first question. If the response was not appropriate or not attained then the author asked the remaining questions of a set to obtain the desired information.

1. How much experience do you have in requirements engineering? How much of that time were you involved in requirements engineering? Have you received training in requirements engineering? Have you been a developer or do you have operational experience?
2. How is the project's scope defined, communicated, and used? Do you identify the project scope, or is it done by someone else? Is there a project vision statement? Is there a standard way for defining a vision statement? Do all stakeholders have access to this statement? Do you use a vision statement (including identification of project scope) to evaluate the relevance of proposed product features, capabilities, and requirements?
3. Do you identify stakeholders? How are stakeholders identified and characterized? Are the characteristics of distinct user classes summarized in a requirements specification?
4. How do you elicit requirements? Do the developers already know what to build? Does management think they can provide the stakeholder perspective? Do you survey or interview customers? Do you use structured interviewing techniques? Do you identify specific individuals that represent each stakeholder class and involve them in the definition of requirements? Do you watch people perform their tasks or model their current work processes?
5. Do you identify the business needs of the customers and their current systems, if they exist? What techniques are used to prepare a partial solution and verify a mutual understanding of the problem? Do you produce scenarios or concept of operations documents? If so, do they generally reflect the stakeholder views? Do you derive functional requirements from scenarios? Do you produce a context diagram? Do you build simple prototypes and ask users for feedback? Do you plan prototyping? Do you deliver prototype code that is extended?
6. Do you distinguish between guidance, functional requirements, and non-functional requirements?
7. How are nonfunctional requirements, such as software quality attributes, elicited and documented? What are the software quality attributes that are generally considered? Do you document performance, safety, maintainability, usability, reusability, and security requirements? Are the customers involved in the identification of important product quality attributes?
8. Do you record the assumptions or rationales underlying requirements?
9. Do you generally prioritize requirements? If so, how are priorities for the requirements established? Do the customers identify which requirements are most important to them? Is consensus reached among the stakeholders? Do you use an analytical process to rate the customer value, cost, and technical risk of each scenario, feature, or functional requirement? If not, why?
10. Do you identify how requirements should be verified?

11. How are the requirements validated? Do you and some stakeholders hold informal reviews? Do you inspect your requirements documents and models with participants that include customers, developers, and testers? Do you write test cases against the requirements and models?
12. How do you determine the quality of requirements statements? Have you ever used checklists to help improve the quality of requirements statements? If so, can you provide an example checklist?
13. How do you detect omitted or duplicated requirements? Do you specify requirements to handle exceptions?
14. Have you ever been in a position where you had to monitor the development of requirements? If so, how did you do it? Did you measure the number or severity of the defects? Did you attempt to classify the defects? If so, what defect categories did you use?
15. Once the initial requirements are defined, what percentage of them are subsequently deleted or changed? How many new requirements are defined, in percentage terms, relative to the initial requirements?
16. How do you reduce the introduction of unnecessary features or capabilities into a product? How do you determine what they are? Are software requirements traced back to their origin? If so, do you have full two-way tracing between every software requirement and some voice-of-the-customer statement, system requirement, scenario, business rule, architectural need, or other origin?
17. How are the system requirements allocated to the software portions of the product? Is software expected to overcome shortcomings in hardware? Do software and hardware engineers help negotiate the functions performed by each subsystem and how they are allocated? Does a system engineer analyze the system requirements and decide those that will be implemented in each software subsystem? Are software requirements traced to system requirements? Are subsystem interfaces explicitly defined and documented?
18. How are requirements used as the basis for developing project plans? Is the delivery date set before you begin gathering requirements? Can you change the project schedule or the requirements? Does the first iteration of the project plan specify a task to gather requirements? Is the rest of the project plan developed after you have a preliminary understanding of the requirements? Do you base the schedules and plans on estimates of the needed resources to implement the required functionality? Are plans updated as requirements change?
19. How are the requirements used as a basis for design? Is each functional requirement traced to a design element? Is there full two-way traceability between individual functional requirements and design elements?
20. How are the requirements used as the basis for testing? Do the testers test what the developers said they implemented? Do you write system test cases against scenarios and functional requirements? Do testers inspect requirements specifications to ensure that the requirements are verifiable? Are system tests traced to specific functional requirements? Is system testing progress measured in part by requirements coverage?
21. How is a software requirements baseline defined and managed for each project? Do customers and managers sign off on the requirements? Do you keep requirements baselines current as changes are made over time?
22. How are changes to the requirements managed? Do you freeze the requirements after the requirements phase is complete, yet permit informal change agreements? Do you use a defined format for submitting change requests and a central submission point? Does a project manager or change control board decide which changes to incorporate? Are changes made according to a documented change-control process?
23. How are the software requirements documented? Do you maintain a history of oral, e-mail, and voice-mail communications, interview notes, and meeting notes? Do you write unstructured narrative textual documents, with or without diagrams? Do you write requirements in structured natural language at a consistent level of detail using a standard template?
24. How are different versions of the requirements documents distinguished? Do you use a sequence number for each document version? Do you distinguish between draft and baselined versions and major and minor revisions? How are the individual functional requirements labeled? Do you use bulleted and numbered lists? Do you use a hierarchical numbering scheme? Is every requirement uniquely identified with a constant value?
25. What software have you used to specify requirements? What were the benefits and drawbacks of each tool? What features or capabilities would you like to see in a requirements engineering tool? Can you customize reports using the tool? Does the tool provide any analytical advice? Would you like the tool to help analyze requirements documents? If so, what would you like it to do?

B. A Verification Checklist

The following checklist identifies numerous questions worthy of asking when examining a requirements document, although no advice is given in how to respond to an answer. For example, the first question of this appendix is "Were all the analysis and specification practices followed?" An answer of "no" would generally indicate a problem, although one may not exist. That is, in a unique situation, one may have determined that one or more practices were not relevant for the current activity or yielded little benefit given the operating environment. In other words, the reader must subject each of these questions and the responses to them to a reasonableness test that is predicated on the operating environment and project constraints. No guidance could be given to handle all, or even the majority, of possible situations.

General

- Were all the analysis and specification practices followed?
- Is each requirement necessary?
- Is each requirement testable?
- Is each requirement weakly coupled (i.e., understandable without examining other requirements)?
- Are the requirements mutually consistent?
- Are the requirements reasonable and realistic, given the actual operating environment, budget, schedule, available technology, and other solution constraints?
- Are these requirements consistent with the requirements documented elsewhere?
- Do requirements inconsistently use terms?
- Are the requirements organized in a sensible manner?
- Are any requirements missing?
  - Have availability requirements been defined?
  - Have installation requirements been defined?
  - Have maintainability requirements been defined?
  - Have performance requirements been defined?
  - Have portability requirements been defined?
  - Have reliability requirements been defined?
  - Have resource and performance margin requirements been identified?
  - Have safety requirements been defined?
  - Have security requirements been defined?
- Do the requirements overly constrain the system design?
- Does a requirement necessitate the use of non-standard, unusual, or unique hardware or software?
- Are the functional requirements sufficiently complete to begin design?

Data

- Are all data elements described?
- Has the data flow been described?
- Is every data element created, referenced, updated, and deleted?
- Has all data been checked for consistency?
- Has the mapping between local views of data and global data been shown?
- Has the management of stored and shared data been described?
- Have accessors for data elements been defined?
- Are there any special integrity requirements on the data?
- Have the types and frequency of occurrence of operations on data been specified?

- Have the modes of access (e.g., random, sequential) for data been specified?

Exception Processing

- Is error checking and recovery required?
- Are sufficient delays used by error recovery processes?
- When performance degradation is the preferred error response, is the degradation predictable?
- Does every path from a hazardous state lead to a lower-risk state?
- Are feedback loops specified to compare the actual and expected computational results?

Input/Output

- Is the latest time specified when an input will be considered valid?
- Are minimum and maximum arrival rates specified for each input?
- Can input received before start-up, while off-line, or after shutdown affect the start-up behavior?
- Are inputs identified that if not received can lead to a hazardous state or prevent recovery?
- Are all the outputs produced by a function used by another function or transferred across an external interface?

Interface

- Have all the interfaces been identified?
- Have the interfaces been described in enough detail to begin design?
- Are the inputs and outputs for all the interfaces sufficient and necessary?
- Are the interface requirements between hardware, software, personnel, and procedures described?
- Which interfaces are likely to change during development or after deployment?

Miscellaneous

- Have clear criteria for accepting or rejecting a system been established?
- Have the criteria for assigning requirement priority levels been defined?
- Have all assumptions and constraints been completely listed?
- Have the contents, formats, and constraints of all the displays been described?
- Is the terminology consistent with the terminology of the customer?
- Have the target language, development environment, and run-time environment been chosen?
- What bad things can occur that may have significant impact?

C. Additional Examples of Problematic Requirement Statements

Following are numerous examples of requirements statements that suffer from requirements issues. For each one, a recommended alternative is provided, which may or may not be appropriate given the original author's intent.

EXAMPLE 9. An incomplete, compound requirement.

Statement:	In the [system] "shadow" operational mode, the [subsystem] shall relay data to [an external system] at 2 Hz.
Issue:	The requirement specifies both a functional requirement and a performance requirement.
Issue:	The requirement does not specify the data to be transmitted.
Issue:	The requirements document containing this requirement did not identify any requirements to handle anomalous conditions. For example, should the subsystem buffer data that would cause the specified rate to be exceeded? If so, what should be the buffer size?
Revision:	The [subsystem] shall provide the data *specified in Table T of Document D* to [an external system] while in "shadow" operational mode. The [subsystem] shall provide the data *specified in Table T of Document D* to [an external system] while in "shadow" operational mode at 2 Hz.

EXAMPLE 10. A weakly-defined, compound requirement.

Statement:	The [subsystem] shall *be capable of* recycling within 10 minutes after a launch abort.
Issue:	The requirement identifies both a functional and non-functional requirement.
Issue:	The requirement uses a weak phrase - be capable of - which can be interpreted in many ways.
Revision:	The [subsystem] shall provide a command to recycle itself. The [subsystem] shall enable the recycle command in 10 minutes or less after launch.

EXAMPLE 11. An imprecise, compound requirement.

Statement:	The [subsystem] shall provide a commandable and programmable alarm clock which allows the wake up of the CPU at any time on any sol, with a range of 36 hours from the current time.
Issue:	The requirement defines several requirements.
Issue:	The requirement is not stated clearly or concisely.
Revision:	The [subsystem] shall provide an alarm clock. The [subsystem] alarm clock shall enable the interruption of the CPU at any time on any sol. The [subsystem] alarm clock shall be manipulated using an API providing the following commands: • wakeup. The [subsystem] alarm clock shall permit the specification of time to be 36 hours into the future.

EXAMPLE 12. An functional requirement with a hidden performance requirement.

Statement:	The [system] shall hand over pointing control to [an external system] for a fixed duration of sunpoint.
Issue:	The performance requirements is not stated. That is, what is the length of the duration?
Revision:	?

EXAMPLE 13. A compound requirement stated as a passive sentence.

Statement:	The mission clock time, via VME registers, shall be available to the flight software within 5 ms after the deassertion of SYSRESET.
Issue:	The statement does not identify the system component responsible for making the time of the mission clock available to the flight software.
Issue:	The statement also identifies two requirements: one functional and the other non-functional.
Revision:	The [S subsystem] shall set the VME registers to the mission clock time after the deassertion of SYSRESET. The VME registers shall contain the mission clock time within 5 ms after the deassertion of SYSRESET.

EXAMPLE 14. An incomplete, passive requirement statement.

Statement:	Independent timers shall be provided for every time critical event.
Issue:	This requirement statement does not specify what component provides the timers.
Issue:	This requirement also does not specific the time critical events, nor does it identify what an independent timer is. That is, how does an independent timer differ from a dependent timer? Is a unique behavior associated with an independent timer?
Revision:	The [system] shall provide one timer for each time critical event. Time critical events will be defined as: • [...identify each critical $event_i$...]

EXAMPLE 15. A negatively-stated, compound requirement.

Statement:	*No* ground enable to set the time in either the Timing Unit or the Mission Clock shall exist.
Issue:	This states a requirement in the negative form. In this case, the requirement may not be testable because it may not be possible to verify that a "ground enable" was not used.
Issue:	This states two function requirements.
Revision:	The Timing Unit shall set the time using a "positive enable".[a] The Mission Clock shall set the time using a "positive enable".[a]

a. It is assumed that this requirement states how physical hardware is initialized. It is unclear whether a specific component or any component can initialize either physical device.

EXAMPLE 16. An imprecise, negatively-stated requirement.

Statement: The flight software shall program all event timers *no sooner than* 1.5 days before the last timed EDL event.
Issue: This states a requirement in the negative form.
Issue: It also references the "last timed EDL event," which may not be deterministically known until it happens. Further, specifying the "expected landing time" requires less knowledge by the reader to properly interpret the requirement.
Revision: The flight software shall program all event timers at most 1.5 days before the estimated landing time.

EXAMPLE 17. An imprecise exclusionary requirement.

Statement: The [subsystem] shall support UHF down linking to the [spacecraft] in all protocols and modes that it supports except beacon, canister, and MBP.
Issue: If the number of protocols or modes grow, the realization of this requirement may become incorrect because new test cases may not be defined to test newly added elements to the set of non-excluded items.[a] Hence, enumerate all elements of a set.
Issue: The requirement is stated using a weak verb. It is not clear what "support" means.
Revision: The [subsystem] shall transfer data to the [spacecraft] using the following UHF protocols and modes.
- [protocol$_1$:...the valid modes...]
- [...]
- [protocol$_n$:...the valid modes...]

a. This means that code should also be implemented in a non-exclusionary fashion.

EXAMPLE 18. An exclusionary requirement.

Statement: The flight software shall *not* use altimeter data, in flight, until heat shield separation + 15 seconds.
Issue: What data should it use? Actually, the likely intent of this requirement is to indicate when to begin processing of the altimeter data.
Revision: The flight software shall begin processing altimeter data 15 seconds after the separation of the heat shield from the spacecraft.

EXAMPLE 19. A statement identifying a capability, but not a requirement.

Statement: The [system] shall *be capable of* tolerating power interruptions with no subsequent damage to the system.
Issue: An expression of a capability to perform is different that a requirement to perform.
Issue: The statement also does not specify the duration of a power interruption that the system can tolerate.
Revision: The [system] shall function during power interruptions of *X* milliseconds or less.[a]

a. This requirement, however, ignores the additional requirement that a power interruption should not damage a system, whatever that means. In addition, this requirement also specifies the duration of a power interruption, which was lacking in the original requirement, and worthy of mentioning since the intent most likely was not to operate during power interruptions of infinite length.

EXAMPLE 20. A statement identifying a capability, but not a requirement.

Statement: The [system] flight software shall *have the capability to* execute all immediate commands specified in [document].
Issue: An expression of a capability to perform is different that a requirement to perform.
Revision: The [system] flight software shall execute all immediate commands specified in [document].

EXAMPLE 21. A weakly-stated, compound requirement.

Statement:	The [subsystem] shall be assembled by personnel using *appropriate* garments and personnel control.
Issue:	Are appropriate garments a specific color, length, or type? Are they composed of a specific material? What is appropriate personnel control? Does it mean that certain behaviors are required, whereas others are unacceptable?
Issue:	The requirement was not stated in an active form. One of the benefits of stating requirements actively is that it eases the categorization of requirements. For example, this requirement can be rewritten to more easily identify that the subject of the requirement is assembly personnel instead of the subsystem.
Revision:	Personnel shall assemble the [subsystem] while wearing clothing that does not conduct electricity. Personnel shall conduct themselves in a professional manner in the [subsystem] assembly area. A professional manner will be defined as: • [...identify each behavior...]

EXAMPLE 22. The use of a weak word.

Statement:	The flight software shall provide *visibility* into memory page status which shall be available via a ground request following a reboot.
Issue:	Does this mean that the flight software will provide an API that permits the manipulation of various measures of the status of a memory page? Does it mean that a GUI must be provided that displays the memory page status to a display window? The intent is unclear, although it may be satisfied by the following revision.
Revision:	The flight software shall save the preexisting status of memory during a reboot. The flight software shall send the last recorded status of memory to the ground software when requested.

EXAMPLE 23. An imprecise statement of a capability, but not a requirement.

Statement:	The Avionics hardware and software in the system test beds shall *be capable of* testing mission sequences in a *flight-like manner*.
Issue:	This requirement expresses a needed capability, but does not impose a requirement on a system.
Issue:	What is a "flight-like manner?" Such vagueness leaves this phrase open to a wide array of interpretations.
Revision:	The system test bed shall simulate the following subsystems, as specified below. • [...identify each subsystem...]

EXAMPLE 24. Guidance masquerading as a requirement.

Statement:	The [system] shall, *to the extent possible*, adhere to the principles articulated in 'Design, Verification/Validation, and Operations Principles for Flight Systems' (JPL D-17868). The System shall maintain a list of noncompliance with any principles with proper justification for the noncompliance.
Issue:	The phrase "to the extent possible" indicates that the documented principles do not have to be followed; therefore, it is guidance.
Issue:	The second sentence defines an additional requirement that should be separately stated, possibly in a different section of the document from the first requirement.
Revision:	The [system] should adhere to the principles articulated in 'Design, Verification/Validation, and Operations Principles for Flight Systems' (JPL D-17868). The Program Manager shall document the principles that are violated, each specific violation, and the rationale for each violation.

EXAMPLE 25. A doubly vague requirement statement.

Statement:	The flight software shall be *tolerant* to double bit errors in the recovery data stored in the RAD6K EEPROM.
Issue:	This requirement, as stated, can be interpreted in a variety of ways, which may not reflect the intent of the person who authored the requirement. What does "tolerant" mean? Does it mean that a datum containing a double-bit error is ignored? Does it mean that the datum is corrected before being processed? Does it mean that the datum is neither ignored or corrected, but processed anyway? Does it mean that an error or warning message is produced? Further, was it meant to impose a requirement on hardware or software?
Issue:	The size of the datum that is supposed to "tolerate" a double bit error is not specified, unless it is implicitly defined by the meaning of "recovery data."
Revision:	The flight software shall compute a checksum for each word of physical memory that will correct double bit errors stored in the RAD6K EEPROM.

EXAMPLE 26. An imprecise requirement with an embedded assumption.

Statement:	The flight software shall *consider* the IMU delta-velocity and delta-theta data invalid for 2 seconds after the IMU is requested to execute a BIT.
Issue:	What is the consequence of data being invalid for 2 seconds? Did the author really mean 2 seconds after a request to execute a BIT operation or 2 seconds once execution has begun? Is "the flight software shall ignore IMU delta-velocity and delta-theta for 2 seconds after the IMU executes a BIT" more appropriate than the original wording?
Revision:	The flight software shall stop processing IMU delta-velocity and delta-theta data when the IMU executes a BIT. The flight software shall start processing IMU delta-velocity and delta-theta data 2 seconds after the IMU executes a BIT. The flight software shall start processing IMU delta-velocity and delta-theta data following system boot.

EXAMPLE 27. A vague, compound requirement.

Statement:	The flight software shall *reinforce* the enabled state and loaded time for each event timer at least once per second.
Issue:	Does "reinforce" mean "reset?" What is "enabled state" and "loaded time"? These terms should be defined in a glossary who specified in a data dictionary.
Issue:	This statement identifies two requirements.
Revision:	The flight software shall provide a mechanism to set the "enabled state" flag. The flight software shall set the "enabled state" flag at least once per second. The flight software shall provide a mechanism to set the "loaded time" flag. The flight software shall set the "loaded time" flag at least once per second.

EXAMPLE 28. A ambiguous requirement.

Statement:	Heaters *used* by software thermal control shall be *configurable*.
Issue:	Software does not use heaters although software may control them. Is this what the author meant? How can heaters be configured? Does configuration have anything to do with the location of the heaters on a spacecraft? Thus, this requirement is not a clear, crisp statement that is easily understood by a person. Consequently, it could be interpreted in a multitude of ways.
Revision:	The thermal software shall control the heaters. The thermal software shall provide an API that lets the following attributes to be changed. • [...provide descriptions of each interface...]

EXAMPLE 29. A useless requirement.

Statement:	The [system] shall meet all performance requirements when the earth or the moon is present in the [subsystem] Field of Regard (FOR).
Issue:	Does this mean that performance requirements do not have to be met when the earth or moon is not present in the subsystem FOR? If not, what is the relevance of the statement?
Revision:	*Delete the requirement.*

EXAMPLE 30. A meaningless requirement.

Statement:	The [system] shall meet all performance requirements over an input voltage range of 22 to 34 volts.
Issue:	Is this saying that the performance requirements do not have to be met when the voltage is not in this range? Or, is this requirement identifying a constraint on the power supply?
Revision:	*Delete the requirement.*

EXAMPLE 31. An imprecise requirement.

Statement:	There shall be a mechanism for the event timer to verify that the firing times received by the event timers were not corrupted during transmission.
Issue:	This requirement is not stated precisely.
Revision:	The [S subsystem] shall validate the firing time of each event timer.

EXAMPLE 32. An unclear requirement.

Statement:	The average [system] ‘standby’ power shall not exceed 26 W.
Issue:	Does this mean that it is permissible to have 0 W standby power for 5 days, if 156 W was available on the sixth day?
Revision:	?

EXAMPLE 33. An incomplete requirement using a common weak phrase.

Statement:	The [subsystem] shall perform validity checking of all data that is transferred to it from [an external system]. *As a minimum*, this shall include a checksum.
Issue:	In general, the phrase “as a minimum” indicates that the requirements are incompletely specified, which allows the developer to include unnecessary features or exclude needed ones.
Revision:	The [subsystem] shall validate all data received from [an external system] using a checksum.

EXAMPLE 34. An incompletely specified requirement.

Statement:	Following a reset, the flight software shall always reload the flight software code, flight software parameters, and *all other required data* stored in non-volatile storage.
Issue:	What are “all other required data?” If this data is specified somewhere else then a specific reference should be provided that uniquely identifies it. If this data is not specified elsewhere then the interpretation of this phrase is left to the reader, who may interpret it differently than its author.
Issue:	This requirement should have been written as three separate functional requirements, and possibly one non-functional requirement imposing an ordering constraint among the three functional requirements, which is left open to interpretation in the original statement.
Revision:	The flight software shall reload itself immediately following a reset. The flight software shall reload the flight software parameters immediately following a reset. The flight software shall reload the following data. • [...identify each datum...]

EXAMPLE 35. A vague requirement.

Statement:	The flight software shall *attach ancillary data* to all data products.
Issue:	What does "attach" mean? Does it mean "associate"?
Issue:	What is "ancillary data? Nor does it identify whether ancillary data differs for each type of product. Thus, it is possible that this is a meta-requirement that should be decomposed into multiple requirements per product type.
Issue:	What data operations does the application need to manipulate the ancillary data, if any? This was not stated in the requirements document that contained this requirement. (See "Fully specify all requirements" on page 18.)
Revision:	The flight software shall provide the following attributes for every datum: • [...identify each datum...]

EXAMPLE 36. An open-ended, ill-specified requirement.

Statement:	The flight software shall measure and collect *performance and execution statistics* as a component of determining correct flight software execution.
Issue:	What performance and execution statistics are to be collected and measured? The requirement makes no attempt to specify these statistics, which means that the requirement engineer or the user may not be provided with what they expect.
Revision:	The flight software shall measure the following: • [...identify each measure...][a] The flight software shall store the following measures: • [...identify each measure...]

a. Tables should not be used to identify the measures because the use of a table makes it difficult to detect changes in the actual requirements.

EXAMPLE 37. An assumed requirement.

Statement:	The [subsystem] shall support [[16]] sequenced commands per second.
Issue:	This requirement was not finalized and it was assumed that 16 was required.
Revision:	The subsystem shall support 12 sequenced commands per second.

EXAMPLE 38. A design requirement.

Statement:	The flight software shall *be designed* such that a new flight software image can be *established* and executed without modifying a default flight software image in non-volatile storage.
Issue:	This statement imposes a requirement on the system design, instead of the delivered system.
Issue:	What does "established" mean? Does this mean "uploaded, stored, and validated?" If so, this means that this requirement needs to be restated by at least the following seven requirements.
Revision:	The [subsystem1] shall upload flight software images to [subsystem2]. The [subsystem2] shall store uploaded flight software images in unused non-volatile storage. The [subsystem2] shall signal an error when insufficient non-volatile memory is available for storing an uploaded flight software image. The [subsystem2] shall validate uploaded flight software images. The [subsystem2] shall shutdown the executing flight software image before executing a new one. The [subsystem2] shall execute flight software images. The [subsystem3] shall command [subsystem2] to execute a specific flight software image.