Practical Software Requirements: Engineering and Management

Karl E. Wiegers

8/15/98

Table of Contents

Part 1. Software Requirements: What and Why

Chapter 1. The Essential Software Requirement (5000 words)

Objectives of This Book

The purpose of the book is to help the reader learn about, and be able to apply, effective techniques for the requirements engineering and management activities in his or her organization. Several specific and tangible benefits the reader can achieve by applying the principles and practices described in the book are presented in this section.

Some Caveats

It can be difficult to begin applying new ways of working, because the day-to-day project pressures do not abate while we try to make long-term investments in better processes. It can also be tempting to try to apply a variety of new techniques at once, without carefully considering whether they are appropriate for the problems at hand. This book is intended to provide readers with a more robust tool box of techniques, which can be applied to many, but probably not all, of the requirements-related problems they encounter.

This is not a comprehensive treatise of all that is known about development of software requirements. The approaches described here are not methodology-specific, but rather represent techniques that are well established (if not always applied) in the software industry. There are no magic solutions to your software requirements problems here, just descriptions of sound practices that have worked for others.

Case Studies

The requirements-related techniques and project work products described in this book will be illustrated with examples drawn primarily from two case studies. One case study is a medium-sized information system called the chemical tracking system. Don't worry, you don't need to know anything about chemistry, but this is a nice, real project example for illustrating requirements practices. The other case study is a Web development project to create a website with which users can make airline reservations and perform related activities.

When Bad Requirements Happen to Nice People

Several of the problems a development organization or its customers can encounter because of inadequate requirements engineering or requirements management are described. Requirements-related risks and how to manage them are addressed more fully in Chapter 4.

Every Project Needs Requirements

All projects, even those written for a single user, need to have some attention paid to requirements. Even products for which there is no apparent customer, such as utilities used by a development group, should have their needs identified. Developers who

have worked on both software written for in-house use and that which is sold understand the need for emphasizing requirements in both situations.

Software Requirements Defined

Several definitions of software requirements are presented. The most common emphasis in such definitions is on describing, as fully as possible, the expected external behavior of the software system. Requirements do not include design details, implementation details, project planning information, or testing information.

Three Levels of Requirements

Three levels of requirements are delineated: business requirements (captured in project vision or statement of scope), the user view (captured in use cases) and the engineering view, used by development, testing, QA, and related functions (functional requirements, captured in the SRS). Some examples of each are provided.

Requirements Terminology

One of the problems with the software industry is the lack of common definitions for terms we use to describe aspects of our work. Confusion arises when we do not use adjectives in front of key words, like "requirements" or "prototype", thereby conjuring different expectations and understanding among the listeners. Some key terms used in this book are defined here.

Classes of Software Requirements

Four classes of requirements are described:

1. functional requirements (describe the behaviors the system must exhibit and the tasks it lets the user perform);
2. nonfunctional requirements (including interfaces, standards, regulations, performance, quality attributes)
3. inverse requirements (things the product must not do or permit); and
4. known design and implementation constraints.

Projects may also have other kinds of requirements, such as development environment requirements or implementation requirements for releasing a product and moving it into the support environment. These kinds of requirements are not discussed further in this book.

Benefits Achievable from a High-Quality Requirements Process

Effective requirements processes can reduce rework, reduce the expectation gap between what the user anticipates receiving and what the developer constructs, emphasize a collaborative approach to product development, save time by avoiding unnecessary functionality, minimize the adverse impact of requirements changes, reduce maintenance and support costs, and facilitate system testing.

Characteristics of Excellent Requirements Specifications

Eleven characteristics an excellent requirements specification (SRS) should have are described. Each requirement should be clear, complete, consistent, correct, feasible, modifiable, necessary, prioritized, traceable, unambiguous, verifiable. No requirements specification is likely to ever exhibit all of these desirable characteristics for every

requirement statement, but writing and reviewing requirements with these characteristics in mind will lead to improved requirements documents.

Chapter 2. Requirements from the Customer's Perspective (3000 words)

Who is the Customer?

The "customer" is sometimes thought of as the person or organization that is acquiring or paying for the software project. In this book, I use the terms "customer", "user", and "end user" interchangeably, since requirements must be gathered from people who will actually use the product (end users). The risk of gathering incorrect requirements increases if the source of voice-of-the-customer (VOC) input is farther removed from actual hands-on end users. Beware of listening to the loudest voice you hear, if it doesn't actually represent end users, but rather some other project stakeholder, such as a funding authority, senior business manager, or someone with a political agenda.

The Customer-Development Partnership

If the software doesn't allow customers to do their job, and if it doesn't satisfy their implicit, nonfunctional expectations, everyone will be unhappy. The software development business is at least as much about communication as it is about computing. Excellent requirements demand effective two-way communication and collaboration between requirements analysts and customers.

Too often, the relationship between development and customers (or customer surrogates, such as marketing) becomes adversarial. We can only succeed on such a collaborative effort when all parties involved know what they need to be successful, and when they understand and respect what their collaborators need to be successful. This section addresses some things customers and developers can expect from each other as part of the requirements definition and management processes. These points may be presented in the form of a "Software Customers' Bill of Rights" and a "Software Requirements Customers Bill of Responsibilities".

What Customers Can Expect from Developers

Customers can expect the developers to speak the language of their business when discussing requirements; they should not be expected to master computer jargon. However, they should not expect developers to become domain experts and understand the nuances and implicit aspects of the customer's application domain. Customers should rely on developers to present them with implementation alternatives and with feasibility limitations. Customers should expect developers to collect, organize, and sift through the information provided to them to structure the requirements statements precisely and clearly.

What Developers Can Expect from Customers

Developers expect customers to be willing to educate them on application domain concepts and terminology. Developers expect customers to be as specific and precise as possible when providing input on requirements, and be able to spend time iteratively clarifying and fleshing out requirements. Developers must rely on customers to make timely decisions when requested to do so. Developers also can expect customers to take the time to review requirements documents, and to set priorities for product features.

Nonfunctional Requirements

Users typically focus on functional requirements: the things the software will let you do. In addition, though, users often have implicit expectations about certain behavioral and less tangible aspects of the product: quality attributes. As a consequence, they might be more likely to be missed or overlooked. These nonfunctional requirements can be difficult to define, yet they often make the difference between a product that simply does what it is supposed to, and a product that truly delivers customer delight. If we don't think about and discuss these quality characteristics early on, then we're just lucky if the product exhibits the characteristics the users implicitly expect. Chapter 11 discusses the kinds of quality attributes, ways to collect them, and ways to document them in more detail.

Punctual Precision, or Deferred Decisions?

Customers sometimes are comfortable hiding behind ambiguous requirements, as that appears to give them the perennial opportunity to change their minds, or to reinterpret what the requirements analyst thought he heard them say. However, these ambiguities must be resolved at some point during development, and it is most effective if customers help make those decisions, rather than relying on developers to guess correctly. Developers don't normally understand the application domain and user needs as thoroughly as a customer representative, and consequently won't always make the decisions the customers would prefer. Telepathy is not a sound technical foundation for a software development project.

Both customers and developers must avoid the temptation to hide behind ambiguity and vagueness in the requirements.. It's fine to include TBD (to be determined) markers in the requirements specification, indicating that additional research or thinking is needed. However, it's highly risky to proceed with design and implementation when TBDs remain in the requirements section being implemented.

What About Sign-Off?

Though not used universally, the concept of signing off on a requirements document is discussed. Sign-off is most commonly associated with baselining the requirements agreement at a specified point in time. All participants must know what sign-off means. Is it a meaningless ritual that can be ignored in the future when convenient? Or is it an agreement that the document being signed represents our best understanding today, and future changes can be made by following a defined change process and renegotiating resources and commitments?

Chapter 3. Good Practices for Requirements Development (4500 words)

Requirements Engineering versus Requirements Management

Some authors call the entire discipline of software requirements "requirements engineering," while others call the whole thing "requirements management." I find it useful to separate the two subdisciplines. Requirements engineering includes: eliciting needs from users representing all user classes; understanding actual user tasks and objectives; understanding the relative importance of quality attributes; negotiating implementation priorities; and translating user needs into written specifications and models. This is where the transition to requirements management takes place.

Requirements management starts after the requirements are collected, but before the developers have agreed to accept them and build them into a product. Customer

acceptance is only half of the equation to approve requirements. Requirements management includes: reviewing requirements before accepting them; defining the requirements baseline and controlling changes to it; keeping project plans current with the requirements; and tracking requirements stability. You can begin applying many requirements management practices immediately, no matter what life cycle stage your project is in.

Requirements Development Good Practices

Approximately 45 good practices for requirements engineering and management are stated and briefly described, along with the kinds of problems they are best suited for addressing. References are provided to parts of this book or other sources containing additional detail. These practices are organized into the following sections: training; gathering requirements; analyzing requirements; documenting requirements; validating; managing requirements; and project management.

Chapter 4. Improving the Requirements Process (5000 words)

Current Requirements Practices Self-Assessment

The reader is referred to Appendix A, which contains a questionnaire she can use to calibrate her organization's current practices around requirements development. Based on the responses to this self-assessment, the reader can choose which portions of the book are most pertinent as a starting point.

Key Requirements Process Components

Some of the elements of an organization's software process that enable effective and repeatable requirements development are described here. Examples of many of these work products and procedures are provided elsewhere in the book. Key requirements engineering process components include : use case template, SRS template, requirements specification guidance, procedure for requirements specification, procedure for interface specification, example SRS and interface specification, requirements allocation guidance, SRS inspection checklist. Key requirements management process components include: requirements management policy, requirements change procedure, change control board operation procedure, impact analysis checklist and worksheet.

Impact of the Requirements Processes on Other Stakeholders

Changing your requirements process means the interfaces you present to other stakeholder communities for your project may also change. Various stakeholders are participants in the requirements process, and so their roles in, and contributions to, the process may also change. Expect some resistance to these process changes, since no one likes to be forced out of their comfort zone. For example, a requirements change control process may well be viewed as a barrier thrown up by development to make it harder to get changes made. In fact, though, it should be used to provide structure and order to the change process, and to permit good business decisions to be made by better informed people. Be prepared to educate stakeholders about why the changes are being made, and how the changes will affect them.

Gaining Commitment to Change

Tips are provided for how to convince managers, customers, marketing, and other stakeholders of the need for documenting project software requirements and for improving your current requirements processes. Without requirements, how do we know when we're done? The cost to fix a defect increases rapidly the later it is found

in the development process. Any development organization can probably find many examples of requirement-based problems from previous projects to use as justification for investing in improved requirements approaches. The potential return on investment from improved requirements engineering and management is discussed.

Building a Process Improvement Roadmap

An approach is suggested for developing a roadmap for implementing improved requirements practices, based on the requirements practices self-assessment questionnaire and techniques that might address the problems revealed. The importance of developing a risk management plan for your requirements process improvement activities is called out.

Chapter 5. Risk Management and Software Requirements (3500 words)

Know Your Enemy: Software Risk Management

Risk management provides a standard mechanism for identifying risk factors, documenting them, evaluating their potential severity, and identifying mechanisms for mitigating those risks. This section provides a concise presentation of risk management, with examples of how it can be applied to requirements topics.

Typical Requirements Risks

Some of the common project risk factors pertaining to software requirements are itemized here, with references to sources of additional detail.

Risk Management Planning

A project's risk management plan should include any risk factors pertaining to requirements. Guidance is provided on how to go about identifying project risk factors, evaluating their severity, documenting them, and controlling them.

Case Study: Sample requirements-related risk statements from one or more case study projects are presented.

Part 2. Software Requirements Engineering

Chapter 6. Establishing Project Scope (3000 words)

Defining the Vision

The scope of the project or product must be defined very early. The project's vision and statement of scope incorporate the high-level business objectives for the product. All use cases and functional requirements developed must align with, and enable achievement of, these business requirements. The vision helps get all project participants working with a common understanding of the outcome. A template for the statement of scope is presented.

Keeping the Scope in Focus

The project's vision and statement of scope provide the reference frame to be used for assessing whether proposed requirements (or changes to requirements) are appropriately included in the project or not.

The Context Diagram

The context diagram defines the interfaces between the either the problem being addressed, or the system being developed, and the outside world. The nature of information and material flows across these interfaces can also be defined. It is essential to define this boundary between the current problem and the rest of the world as part of the scope definition of the project.

Case Study: statement of scope for chemical tracking system; statement of scope for website

Chapter 7. Finding the Voice of the Customer (3000 words)

Where Do Requirements Come From?

Eight typical sources of software requirements are identified:
1. documents describing current or competing products
2. system-level requirements for a product containing both hardware and software
3. problem reports and change requests for a current system being replaced
4. interviews and discussions with representative users
5. marketing surveys and user questionnaires
6. observations of potential users of the new product as they do their current job or use an existing system
7. feedback from evaluation of prototypes
8. task analysis of the objectives users need to accomplish with the new product

User Classes

User classes represent distinct groups of users of a product. They may vary in frequency of use, features used, experience and education levels, or security privilege levels. It's important to identify and characterize the different user classes for a product early in the project, and to attempt to gather requirements from representatives of each user class.

Case Study: user classes for chemical tracking system, example of how to document user classes and their characteristics; user classes from airline reservation website project

Identifying Suitable User Representatives

User representatives need to be found for projects in different situations: internal information systems (IS), commercial software, integrated systems, web development, contracted software. While it's usually easiest for IS development, reps can often be acquired to contribute to the other kinds of projects. Use people from current beta testing sites, build on existing customer relationships, develop focus groups of current users, or assess the demographics of current visitors to your web site to locate candidate representatives.

The Product Champion

Product champions are individuals representing discrete user classes who serve as the primary interface between customers and developers. The product champion approach provides a way to get the voice of the customer as close as possible to the ear of the developer. The responsibilities expected of the champions are stated, as are ways to manage customer involvement on larger projects or geographically separated user communities with multiple champions representing diverse user classes. The focus is on collecting, documenting, and understanding the user's business requirements that will must be addressed by the software product.

Case Study: identifying product champions from chemical tracking system; relationship among them; product champion expectations and sample dialog with champions about agreeing on what role they'll play

Chapter 8. Hearing the Voice of the Customer (5000 words)

Requirements Gathering Techniques

Several techniques for gathering requirements from customer representatives are presented, including interviews, observation, workflow analysis, and questionnaires. Joint Application Design (JAD) sessions have been widely used to develop software requirements using facilitated sessions involving customer and development representatives. Examples of JAD-type sessions will be described in the case studies in this chapter.

As the requirements analyst gathers the voice of the customer, he must classify the many bits of information collected into several categories: business requirements, use cases, functional requirements, business rules, possible solutions, quality attributes, or extraneous information.

Usage Scenarios and Use Cases

Usage scenarios represent specific ways we anticipate the customers will use the product. Use cases are an abstraction of specific scenarios, which encapsulate the kinds of tasks users will expect to be able to accomplish with the product. Each use case should encompass the normal course behavior of the task, alternative course paths and exception conditions. Use cases are developed early in the project's life, and they are used to derive functional requirements and test cases.

Case Study: several sample use cases from the chemical tracking system and from the web development project

Use Cases and Software Functional Requirements

Use cases represent the high-level view of requirements: the user view, or the business requirements. The software functional requirements are derived from these business requirements; they constitute the engineering (development, testing) view of the requirements. The use case approach makes it easier to separate these two views than do previous requirements gathering and analysis techniques. Functional requirements should be traced back to individual use cases or other VOC input sources.

Case Study: example of a use case and corresponding functional requirements from the chemical tracking system

Use Case Identification and Elaboration

A case study is used to illustrate how use cases can be identified in facilitated workshops (like JAD sessions) involving an analyst and several customer representatives. Some sample use cases are shown. The benefits of the use case approach are described, compared to previous techniques that focus more on the product's features than on what users need to be able to do with the product. Watch out for requirements that users so intuitively expect to be present that they don't express those needs.

Case Study: scenario of how a use case workshop from the chemical tracking system went

Documenting Use Cases

A suggested template is provided for documenting individual use cases, with guidance for how to use it. The template includes a description of the normal course of events that characterize the execution of the use case, as well as alternative courses and exceptions that can arise. There is a many-to-many relationship among use cases and functional requirements. Three possible ways to relate the use cases and the corresponding functional requirements are presented.

Case Study: sample use cases from the chemical tracking system and from the web development project are documented using the template

Chapter 9. Capturing the Requirements (5000 words)

The Need for Multiple Views

According to requirements authority Alan Davis, no single view of the requirements provides adequate understanding. We need to use a variety of textual and graphical representations of the requirements to help us identify inconsistencies, ambiguities, errors, and omissions. Guidance is provided as to how to determine which views and representations are most valuable for different situations.

The Software Requirements Specification

Despite its shortcomings, structured natural language remains the most practical way of documenting requirements for most software products. An adaptation of the IEEE SRS template as a way to collect requirements information, and an interpretation of the IEEE 830 standard (including its shortcomings) are presented.

Case Study: sample contents for the various SRS sections are presented.

Guidelines for Writing Requirements

Suggested approaches for how to write requirements statements are presented. The suitable level of requirements granularity, schemes for uniquely labeling each requirement, and examples of good and bad requirements statements are included. Ways to represent sets of similar requirements in tables or list form are illustrated. The notion of discretely testable requirements as a way to judge the appropriate level of granularity is presented, with examples.

Chapter 10. A Picture is Worth 1024 Words (5000 words)

Graphical views of the requirements include data flow diagrams, entity-relationship models, state-transition diagrams, dialog maps, and object class models. These models are useful both for elaborating and exploring the requirements, and for designing solutions. This book does not go into detail on some of the modeling techniques that are thoroughly treated in other sources, but this chapter contains short presentations of the major methods and some examples. The reader is provided with an indication as to what kinds of problems might be modeled most appropriately with one of the various methods, with pointers to selected resources from which they can learn more.

Case Study: each of these models is illustrated with a simple example from the chemical tracking system

Data Flow Diagram

The data flow diagram is a fundamental tool of structured analysis. It illustrates the major transformational processes of a system (physical or software), the data stores, and the flows of data or material among the processes and the stores.

Entity-Relationship Diagram

The entity-relationship diagram is a way to build a conceptual model of a system, representing physical or logical entities, their attributes, and relations among them.

State-Transition Diagram

The state-transition diagram depicts the various states the system can be in, and the allowed state changes and conditions under which each state change can take place.

Dialog Map

The dialog map is a tool for modeling a user interface architecture (conceptual or physical) at a high level of abstraction, such that the navigation links among dialog elements can be reviewed for correctness. The dialog map can be verified using the system test cases developed from use cases. By tracing the flow of execution of each test case, we can identify missing or incorrect requirements, correct errors in the dialog map itself, and refine the test cases.

Case Study: chemical tracking system dialog map and use of test cases to verify dialog map; dialog map for website; use of dialog maps in use case workshops to clarify requirements for a website

Object Class Model

Class models depict the relationships among the object classes that are identified during object-oriented analysis, as well as the functionality that can be performed on the data contained in those objects.

Chapter 11. Software Quality Attributes (2500 words)

Nonfunctional Requirements

Beyond the functionality they contain, excellent software products display a set of quality characteristics that represent the most satisfactory balance of several competing attributes. Users will not generally express these nonfunctional needs spontaneously, so analysts must prompt the necessary user input to reach an understanding of these attributes so we can use them as design criteria. In the rush to please the customer, restraint should be used to provide solidly functional systems without unnecessary constraining attributes (such as unreasonable performance expectations) that drive up development cost while providing little additional value.

Quality Attributes

Definitions of about 25 such quality attributes are presented, as are tradeoffs that often have to be made between them. Some of these attributes are visible to the user; examples include correctness, performance, reliability, robustness, and usability. Other attributes are more important to developers, including maintainability, testability, understandability, and portability.

Defining Quality Attributes

A technique for eliciting information on quality attributes from users is presented. Several examples are shown of how to document these quality attributes in a testable and measurable way for different kinds of projects.

Case Study: quality attributes for chemical tracking system (user view); quality attributes for a "graphics engine" application (developer view)

Chapter 12. Risk Reduction Through Prototyping (3000 words)

Prototyping: What and Why

A prototype is a partial implementation of a proposed new product. It can be used to clarify and complete the requirements, to explore design alternatives, and (if done by intent) to grow into the ultimate product. Examples are provided of how prototyping can help flesh out and refine requirements, and the discipline required to use rapid prototyping for this purpose.

Prototyping and Risk

Prototyping is normally considered to be a technique for reducing the risk of failure on a software project, where "failure" can be defined as either building the wrong product, or building the right product badly or late. However, prototyping introduces its own risks, the biggest being that a stakeholder will see a running prototype and conclude the product is nearly completed. Part of the problem arises because we don't use adjectives in front of the work "prototype," so different people have very different expectations about them., including pressure to deliver a throwaway prototype, and the temptation to keep adding new functionality such that a simple prototype evolves into something much more elaborate than necessary to meet the prototyping objectives. Expectation management with stakeholders is a part of successful prototyping.

Horizontal and Vertical Prototypes

The horizontal prototype is like a movie set, with false fronts of user interface screens displayed and some navigation operational, but little or no real functionality. These are sometimes called "mock-ups." The vertical prototype, or "proof of concept," implements a slice of application functionality and is used to determine whether a proposed architectural approach is sound.

Throwaway and Evolutionary Prototypes

A throwaway prototype is built to answer questions, resolve uncertainties, and improve requirements quality. It is specifically intended to be discarded after it has served its purpose. A risk is the temptation to keep adding more functionality to the throwaway prototype, "just to see what it will look like."

The evolutionary prototype is intended to provide a solid foundation for growing the product over time. These are designed and built very differently. More characteristics and usage of both kinds of prototype are presented. Evolutionary prototyping is a fundamental component of the spiral software development life cycle model and of some object-oriented development processes such as Objectory. The first increment of an evolutionary prototype can be thought of as a pilot release. Lessons will be learned from testing and initial evaluation, and the pilot may be backed out or extensively modified. Once the necessary adjustments have been made, development can proceed with the next iteration, on the way to eventual implementation of the full product.

Prototyping Tools

Several tools that can be used for prototyping are mentioned, including Visual Basic, commercial prototyping toolkits, scripting tools, and Web-based approaches using HTML.

Paper Prototyping

Simple, non-executable prototypes can be cheap, fast, and effective. They also provide a good way to deal with the risk that an evaluator of an executable throwaway prototype will conclude the product is nearly done. Ideas for, and references to excellent sources on, paper prototyping are presented.

Evaluating Prototypes

User evaluation of prototypes can be improved by creating a prototype evaluation script that guides the user through a sequence of tasks and asks specific questions to obtain the information the prototype is intended to collect. This is a valuable supplement to the more general evaluation invitation of, "Tell me how this looks to you." It is easy for those evaluating a prototype that includes faked data to become distracted by the data and not focus on the structure and intent of the application as manifested in the prototype.

Case study: sample prototype evaluation scripts for the chemical tracking system and website are provided

Chapter 13. Setting Requirements Priorities (2500 words)

Why Prioritize Requirements?

Requirements priorities are necessary to permit project tradeoffs to be made as needed throughout the project. They can allow the project manager to adjust scope to fit the realities of schedule, budget, and staff restrictions, by dropping or delaying to a subsequent release lower priority functions when new, higher priority requirements are accepted or other project conditions change. Three levels of priority are recommended: required for release 1.0, required for subsequent release, would be nice to have some day.

Games People Play With Priorities

It's often difficult to persuade customers to set priorities, if they know that low priority requirements will likely not be implemented. Some people have the attitude that priorities are unimportant, because if we wrote it in the SRS, we intend to build it. Even when priorities are set, there may be so many at high priority that in practice, the project manager really doesn't have any degrees of freedom to work with.

Prioritizing Based on Value, Cost, and Risk

A simple scheme to help analysts evaluate the relative value, cost, and risk associated with each proposed requirement, feature, or use case is presented. This scheme is loosely derived from Quality Function Deployment concepts. The attractiveness of a feature is directly proportional to the value it provides, and inversely proportional to the cost and the technical risk of implementing it. Value includes both the benefit to the customer if the feature is present, and the penalty paid if it is not. Features having the highest risk-adjusted value/cost ratio should have the highest priority, all other things being equal (they aren't, of course; some requirements are more equal than others). Some features are exempt from this analysis, because they simply must be

included regardless of the cost or impact. An example is a feature required for compliance with government regulations.

Case Study: example of how to establish priorities for selected features from the chemical tracking system

Chapter 14. Validating the Requirements (4500 words)

Reviewing the Requirements

The importance of reviewing requirements documents early and often, formally and informally, is presented. Formal inspection of requirements documents is perhaps the highest leverage software quality practice available. Reviews are an excellent technique for identifying ambiguous requirements, and requirements that are not adequately testable, or are not clearly enough defined to be used as the basis for design, or are in fact design specifications themselves.

Requirements Review Methods

Both informal and formal reviews are important quality activities. specific techniques for having a group of the author's peers search for defects in requirements specifications. An overview of the formal inspection process is presented, with guidance for how to apply them to requirements documents. A checklist of typical requirements defects is included. Appropriate participants for requirements reviews are described, and their roles defined.

Requirements Review Tips and Traps

Typical problems people encounter when attempting to review requirements are pointed out, such as excessively large review teams. Techniques are suggested for overcoming common requirements review problems, such as reviewing too late in the development process, dealing with very large documents, dealing with geographically separated review participants, and so on.

Testing the Requirements

Crystallizing the Vision With Test Cases

Writing test cases forces one to state precisely the expected behavior of a software system under specific conditions. Hence, writing black-box functional test cases is a powerful technique for eliminating ambiguity and vagueness in requirements. The simple act of writing the test cases will reveal many flaws in the requirements. Having the customer representative team walk through the test cases with the analyst and developers is an excellent way to help these key stakeholders achieve a shared vision of the product's expected behaviors.

Deriving Test Cases from Use Cases

Conceptual test cases can be derived very early in the project from the use cases. The test cases can be used to verify both textual requirements specifications and models, such as dialog maps. Such test cases, based on usage scenarios, can serve as the foundation for customer acceptance testing, in addition to comprehensive formal system testing.

Case Study: Sample test cases derived from use cases for the chemical tracking system are shown, and the method by which they were used to find errors in both the requirements and the test cases themselves is described.

Chapter 15. Beyond Requirements Development (2000 words)

From Requirements to Code

This chapter addresses the process of moving from requirements through the rest of development: design, coding, testing. There is a gray area between requirements and design, but we want to keep our requirements documents as free as possible from implementation bias. The importance of, and techniques for, staying out of design during the requirements process are discussed.

As developers work to implement requirements, they will encounter points of ambiguity and confusion that need to be resolved. Ideally, these can be taken back to the customers to provide additional details and make decisions. Any assumptions, questions, guesses, or interpretations the developer makes should be documented and reviewed with customer representatives if it is not possible to resolve such issues in real time.

Requirements-Based Testing

The need to develop system (black box, functional, behavioral) test cases against the requirements is presented. We have to make sure we're testing the product against what the product was intended to do as documented in the requirements, not against what the design or code says. All requirements must have test cases mapped to them. Test progress can be measured in part by coverage of the requirements during testing.

Part 3. Software Requirements Management

Chapter 16. Requirements Management Principles and Practices (4000 words)

Principles and Goals of Requirements Management

Once gathered, documented, and reviewed, the requirements must be managed over time. The primary objectives of requirements management as presented in the Software Engineering Institute's Software Capability Maturity Model are discussed. The CMM itself is not emphasized, and the somewhat stilted CMM terminology is "folksied up" to make it easier for normal people to understand. The primary emphasis of requirements management is on change management.

Practices for Requirements Management

Dealing with changing requirements consumes the bulk of the requirements management effort. Requirements may be added, modified, or deleted from the requirements specification over time. Requirements change practices, including impact analysis and the decision making process, are addressed in other chapters, as is the use of a requirements traceability matrix.

The Requirements Baseline

Requirements changes are made against a reference, baseline requirements agreement. Baselining is normally associated with approval of the requirements documents. The baseline is a snapshot of the requirements agreement at a specified

time. It represents the shared understanding that these requirements constitute our best description at this time, and that future modifications in the requirements documents can be made only through following a defined requirements change control procedure.

Measuring Change Activity

Measurement of change activity is a way to assess the stability of the requirements and identify process improvement opportunities to minimize the adverse impact of changes on the project.

Measuring Requirements Management Effort

Projects should track the effort they devote to requirements management activities. This provides visibility into whether the intended actions to manage the requirements over time are actually being performed. It also provides the opportunity to better plan the requirements management effort that should be budgeted for future projects. If we do not use historical data from previous projects to plan future projects, our estimates will forever be guesses.

Tracking Requirements Status

Another aspect of requirements management is to track the status of each requirement throughout development. This can be done with the help of a database or a commercially available requirements management tool.

Version Control of Requirements Specifications

A final technique for managing the requirements specification is to use configuration management tools and practices for version control. Ways to identify the various versions of the SRS are described.

Chapter 17. Managing the Change Backlog (4000 words)

The Threat of Scope Creep

Controlling scope creep requires that every proposed new requirement be evaluated against the stated scope or vision of the product to see if it belongs in the product or not. To minimize the adverse impact of change on the project because of changing requirements, each project should follow a defined change control process.

The Change Control Process

Basic principles of change control of requirements as part of software configuration management are presented. The elements of an change control process are described, including a sample state transition diagram that describes the life cycle of a change request. Appropriate tools are described to support the process (remember, a tool is not a process), including the value of automated e-mail communications. Pointers to sources of appropriate tools are included.

The Change Control Board

The change (or configuration) control board (CCB) is a best practice for software development.This is the body of people, be it one individual or a diverse group, that is empowered to make binding decisions about which proposed requirements changes (and enhancements and defects) will be incorporated into the product. Its composition, roles, and operating principles are described. A sample operating procedure for a CCB is included.

Chapter 18. Requirements Change Impact Analysis (2000 words)

Change Isn't Free

The perception that changes are free leads to scope creep. Developers have responsibility to estimate the impact of proposed changes so appropriate, informed business decisions can be made.

Impact Analysis Checklist

An important part of responsible requirements management is to assess each proposed requirement change for its cost and impact on the project. This checklist poses many questions to help the person doing this assessment to identify all the work that might be associated with making the proposed change.

Impact Analysis Worksheet

This worksheet lets the person assessing the impact of a proposed requirements change estimate the amount of labor that will be involved for the tasks identified with making the change. By performing such an analysis, we can do a much better job of judging the consequences of incorporating a proposed change, such that the impact on the project's schedule and cost are more accurately projected.

Case Study: an example of how a requirements change request for the chemical tracking system is proposed and handled is presented

Chapter 19. Requirements Traceability (2000 words)

Tracing Requirements Through Development

Requirements traceability involves both linking each software requirement to its source (system requirement, use case, specific VOC source), and to downstream development life cycle deliverables: design elements, source code and procedures, test cases. Several ways to construct a requirements traceability matrix (RTM) and several benefits of the RTM are discussed.

Tracking Interrelated Requirements

Another form of requirements traceability is to keep track of interconnections among individual requirements. This information helps identify the propagation of change that can result when a specific requirement is deleted or modified. It also helps relate dependencies among multiple requirements.

Case Study: a portion of a sample requirements traceability matrix for the chemical tracking system is shown

Chapter 20. Tools for Requirements Management (2000 words)

Some commercial tools that are available for assisting with the requirements management process are described. Benefits and caveats of the tools are presented. Pointers to sources of additional information about the tools are included (e.g., Web URLs, if we think they're stable enough to publish). As tool availability and functionality change rapidly, this chapter will not go into great detail on specific currently available tools. The general capabilities of requirements management tools will be emphasized instead.

Appendix A: Current Requirements Practice Self-Assessment (1500 words)

This questionnaire addresses 20 software requirements engineering and management practices, and it offers a scale for the reader to evaluate his current organization practices. Pointers are provided to other parts of the book to get help on specific weaknesses identified by the questionnaire.

Appendix B: Solving Requirements Problems (4000 words)

The appendix contains a table of common requirements engineering and management problems that software projects might encounter. These are drawn from small group discussions in the many classes I have taught on "In Search of Excellent Requirements." Possible solutions to the problems are presented, in the form of the best practices described earlier in the book.