

# GETTING STARTED

- **Chapter 30** Agile Requirements Methods
- **Chapter 31** Your Prescription for Requirements Management

## DEDICATION

Over the course of many years, we and others who have contributed to this book have taught, and have been taught by, thousands of students interested in improving project outcomes by doing a better job of managing their software requirements. In this second edition of the book, we've been more prescriptive, but we still recognize that there is no one right way to perform requirements management. No one single elicitation technique applies in every circumstance; no one single process fits all teams. Projects have varying degrees of scope and complexity. Application types vary tremendously and come from many different industries.

Yes, requirements management is a broad topic, and it is also very deep. A recurring theme from the classroom is that students feel the need to have a more prescriptive process—a recipe, if you will—for applying what they learned in class. “You’ve told us too much,” our students might say. “Just give us a single generic process that we can start with,” they continue. “We know it’s not that simple, but we’ll be happy to modify it as necessary for *our* project. We need a more prescriptive starting point, a step-by-step process so that we can better apply what we learned. *Just tell me how to get started!*”

OK, you’ve got it. These next two chapters are dedicated to these students and to those of you who share their point of view and this common “user need.”

## WHAT YOU'VE LEARNED SO FAR

Before we can kick-start your project requirements process, however, let's summarize what you've learned in the book so far.

### Introduction

In the introductory chapters, we looked at why our industry often does a poor job of delivering quality applications on time and on budget. Some of the root causes of this problem are clear. *Lack of user input, incomplete requirements and specifications, and changing requirements and specifications* are commonly cited problems in projects that failed to meet their objectives.

Perhaps developers and customers alike have a common attitude that “since we can't determine everything we want in advance, it's better to get started with implementation now because we're behind schedule and in a hurry. We can pin down the requirements later.” Since, even in this requirements text, we've agreed that it isn't possible to know everything in advance, that doesn't seem so inappropriate. But all too often, this well-intentioned approach degenerates into a chaotic, out-of-control development effort, with no one on the team quite sure what the user really wants or what the current system really does.

To address these issues, we've recommended an encompassing philosophy of requirements management, which we defined as

*a systematic approach to eliciting, organizing, and documenting the requirements of the system, as well as a process that establishes and maintains agreement between the customer and the project team on the changing requirements of the system.*

Since the history of software development—and the future for at least as far as we can envision it—is one of ever-increasing complexity, we also understand that well-structured and well-trained software teams must address the software development problem. Every team member will eventually be involved in helping manage the requirements for the project. These teams must develop the requisite skills to understand the user needs, to manage the scope of the application, and to build systems that meet these user needs. The team must work *as a team* to address the requirements management challenge.

In addition, we discussed that we must take an “iterative and incremental” approach to the problem, an approach that recognizes that not all require-

ments can be determined in advance. By delivering increments to the users, we can further refine our understanding of their needs and better evolve the system to meet their real needs.

### Team Skill 1: Analyzing the Problem

In Team Skill 1, we introduced a set of skills your team can apply to *understand the problem to be solved before too much is invested in the application*. We introduced a simple, five-step problem analysis technique that can help your team gain a better understanding of the problem to be solved.

1. Gain agreement on the problem definition.
2. Understand the root causes of the problem.
3. Identify the stakeholders and the users whose collective judgment will ultimately determine the success or failure of your system.
4. Determine where the boundaries of the solution are likely to be found.
5. Identify and understand the constraints that will be imposed on the team and on the solution.

All in all, following this process will improve your team's ability to address the challenge ahead, *providing a solution to the problem to be solved*.

We also noted that a variety of techniques could be used in problem analysis. Specifically, we looked at business modeling, a problem analysis technique that works quite well in complex information systems that support key business processes. The team members can use business modeling to both understand the way in which the business evolves and to define where within the system they can deploy applications most productively. We also recognized that the business model we defined will have parallels in the software application, and we use this commonality to seed the software design phases.

For embedded-system software applications, we applied systems engineering as a problem analysis technique to help decompose a complex system into more manageable subsystems. This process helps us understand where new software applications should come to exist and what purpose they serve. However, we complicate the requirements challenge somewhat by defining these new subsystems, for then we must determine the requirements to be imposed on them.

## Team Skill 2: Understanding User and Stakeholder Needs

We started Team Skill 2 by introducing three “syndromes” that increase the challenge of understanding the real needs of users and other stakeholders. The “Yes, But,” the “Undiscovered Ruins,” and the “User and the Developer” syndromes are metaphors that helped us better understand the challenge ahead and provided a context for the elicitation techniques we developed in this team skill.

We also recognized that since we rarely have been given effective requirements specifications for the systems we are going to build, in order to do a better job of building these systems, we have to go out and *get* the information we need to be successful. *Requirements elicitation* is the term we used to describe this process, and we concluded that the team must play a more active role.

To help the team address these problems and better understand the real needs of users and other stakeholders, we then presented a variety of techniques:

- Interviewing
- Requirements workshops
- Brainstorming and idea reduction
- Storyboarding

Although no one technique is perfect in every circumstance, each represents a proactive way to push your knowledge of user needs forward and thereby convert “fuzzy” requirements into requirements that are “better known.”

## Team Skill 3: Defining the System

In Team Skill 3, we moved from understanding the needs of the user to defining the solution. In so doing, we took our first steps out of the problem domain, the land of the user, and into the solution domain, wherein our job is to define a system to solve the problem at hand.

We invested most of our time in developing the use-case technique since it can do most of the “heavy lifting.” Use cases have a number of advantages over other techniques, including the way the use cases persist in the project to drive testing strategy and the development of the test cases themselves. We also discussed that complex systems require comprehensive strategies for managing requirements information, and we looked at a number of ways to organize requirements information. We recognized that we really have a hierarchy of in-

formation, starting with user needs, transitioning through features, then into the more detailed software requirements as expressed in use cases and supplementary specifications. Also, we noted that the hierarchy reflects the level of abstraction with which we view the problem space and the solution space.

We then “zoomed in” to look at the application definition process for a stand-alone software application and invested some time in defining a Vision document for such an application. We maintain that the Vision document, with modifications to the particular context of a company’s software applications, is a crucial document and that *every* project should have one.

We also recognized that without someone to champion the requirements for our application and to support the needs of the customer and the development team, we would have no way to be certain that the hard decisions are made. Requirements drift, delays, and suboptimum decisions forced by project deadlines are likely to result. Therefore, we decided to appoint someone or to anoint someone to serve as *product manager*, someone to own the Vision document and the features it contains as well as to drive agreement on some of the commercial factors that convert an application into a *whole product solution*. In turn, the champion and the team empower a change control board to help with the really tough decisions and to ensure that requirements changes are reasoned about before being accepted.

#### Team Skill 4: Managing Scope

In Team Skill 4, we examined the endemic problem of project scope. It is not unusual to see projects initiated with *two to four times* the amount of functionality the team can reasonably implement in a quality manner. We shouldn’t be surprised by this; it is the nature of the beast. Customers want more, marketing wants more, and the team wants more, too. Nevertheless, we have to manage this psychology aggressively if we intend to deliver *something* on time.

In order to manage scope, we looked at various techniques for setting priorities, and we defined the notion of the baseline, an agreed-to understanding of what the system will do, as a key project work product. We learned that if scope and the concomitant expectations exceed reality, in all probability, some bad news is about to be delivered. We decided on a philosophy of approach that engages our customer in the hard decisions. After all, we are just the implementers, not the decision makers; it’s our customer’s project. So, the question is, “What, exactly, *must* be accomplished in the next release, given the resources that are available to the project?”

Even then, we expect to do some negotiating. We briefly mentioned a few negotiation skills and hinted that the team may need to use them on occasion.

We cannot expect that the process described so far will make the scope challenge go away, any more than any other single process will solve the problems of the application development world. However, the steps outlined can be expected to have a material effect on the scope of the problem, allowing application developers to focus on critical subsets and to deliver high-quality systems incrementally that meet or exceed the expectations of the user. Further, engaging the customer in helping solve the scope management problem increases commitment on the part of both parties and fosters improved communication and trust between the customer and the application development team.

With a comprehensive project definition, or Vision document, in hand and scope managed to a reasonable level, we at least have the *opportunity* to succeed in the next phases of the project.

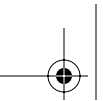
### Team Skill 5: Refining the System Definition

In Team Skill 5, we first took a more rigorous look at requirements and commented on some of the issues that arise in transitioning from requirements to design. In addition, we refined the use cases to sufficient specificity so that we can both implement them and use them later to develop the test cases that will determine when the requirements have been met. We also discussed the importance of nonfunctional requirements, including the system's usability, reliability, performance, and supportability, as well as the design constraints that may be imposed on our process. We described how to organize them in a supplementary specification that, along with the use-case model, completes our information model of the system we are building.

### Team Skill 6: Building the Right System

Designing and implementing the correct system is the biggest job of all. In Team Skill 6, we described how to use the developed use cases to drive implementation via the design construct of the *use-case realization*. We also described how to use the use cases to develop a comprehensive testing strategy by deriving the test cases directly from them.

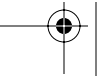
We also developed the concepts and described the challenges associated with *requirements traceability* and demonstrated how that technique can improve the quality and reliability outcomes of a development effort.



In addition, building the right system right also depends on the team's ability to *manage change effectively*. Since change is just part of life we must plan for change and develop a process whereby we can manage it. Managing change helps us make sure that the system we built *is* the right system and, moreover, that it continues to *be* the right system over time.

Lastly, we looked at a framework for assessing requirements quality, as well as overall project quality, within an iterative development process, and we used this to create guidelines your team can use to help assure you will deliver a quality result.

With all that behind us, we're almost ready to look at that requirements prescription our students have been clamoring for. Before we do, however, we have one last meaty topic to cover: the topic of picking an overall requirements method you can apply in your specific project context.





## Chapter 30

# AGILE REQUIREMENTS METHODS

### Key Points

- The purpose of the software development method is to mitigate the risks inherent in the project.
- The purpose of the requirements management method is to mitigate requirements-related risks on the project.
- No one method fits all projects, therefore the method must be tailored to the particular project.
- Three requirements methods (extreme, agile, and robust) are presented.

## MITIGATING REQUIREMENTS RISK WITH EFFECTIVE REQUIREMENTS PRACTICES

So far in this book, we have described a comprehensive set of practices intended to help teams more effectively manage software requirements imposed on a system under development. Since the systems that teams are building today can be exceedingly complex, often comprising hundreds of thousands or even millions of lines of code and tens to hundreds of person-years in development time, it makes sense that requirements themselves are also likely to be exceedingly complex. Therefore, a significant variety of techniques and processes—collectively a complete *requirements discipline*—are required to manage requirements effectively.

However, lest we lose sight of the purpose of software development, which is to deliver working code that solves customer problems, we must constantly remind ourselves that the entire requirements discipline within the software life-cycle exists for only one reason: *to mitigate the risk that requirements-related*

*issues will prevent a successful project outcome.* If there were no such risks, then it would be far more efficient to go straight to code and eliminate the overhead of requirements-related activities. Therefore, when your team chooses a requirements method, *it must reflect the types of risks inherent in your environment.*

#### **Three Points to Remember about Method**

1. The purpose of the software development method is to mitigate risks inherent in the project.
2. The purpose of the requirements management method is to mitigate requirements-related risks on the project.
3. No one method fits all projects; therefore, the requirements method must be tailored to the particular project.

Each of the requirements techniques we've discussed was developed solely to address one or more specific types of requirements-related risks. Table 30–1 summarizes these techniques, along with the nature and type of risks that each is intended to mitigate.

### **METHODOLOGY DESIGN GOALS**

As we have said, the purpose of requirements methodology is to address requirements-related project risks. The purpose of the overall development methodology is to address collective project risks. In his book on agile development, Cockburn [2002] identifies four major principles to apply when designing and evaluating software methodologies.

1. Interactive, face-to-face communication is the cheapest and fastest channel for exchanging information.
2. Excess methodology weight is costly.
3. Larger teams need heavier methodologies.
4. Greater ceremony is appropriate for projects with greater criticality.

Let's examine these principles briefly to see what insight we can gain into selecting the correct requirements management methodology for a particular project context.

**Table 30–1** Requirements Techniques and the Specific Project Risks They Address

Technique	Risk Addressed
Interviewing	<ul style="list-style-type: none"> <li>▪ The development team might not understand who the real stakeholders are.</li> <li>▪ The team might not understand the basic needs of one or more stakeholders.</li> </ul>
Requirements workshops	<ul style="list-style-type: none"> <li>▪ The system might not appropriately address classes of specific user needs.</li> <li>▪ Lack of consensus among key stakeholders might prevent convergence on a set of requirements.</li> </ul>
Brainstorming and idea reduction	<ul style="list-style-type: none"> <li>▪ The team might not discover key needs or prospective innovative features.</li> <li>▪ Priorities are not well established, and a plethora of features obscures the fundamental “must haves.”</li> </ul>
Storyboards	<ul style="list-style-type: none"> <li>▪ The prospective implementation misses the mark.</li> <li>▪ The approach is too hard to use or understand, or the operation’s business purpose is lost in the planned implementation.</li> </ul>
Use cases	<ul style="list-style-type: none"> <li>▪ Users might not feel they have a stake in the implementation process.</li> <li>▪ Implementation fails to fulfill basic users needs in some way because some features are missing or because of poor usability, poor error and exception handling, and so on.</li> </ul>
Vision document	<ul style="list-style-type: none"> <li>▪ The development team members do not really understand what system they are trying to build, or what user needs or industry problem it addresses.</li> <li>▪ Lack of longer-term vision causes poor planning and poor architecture and design decisions.</li> </ul>
Whole product plan	<ul style="list-style-type: none"> <li>▪ The solution might lack the commercial elements necessary for successful adoption.</li> </ul>
Scoping activities	<ul style="list-style-type: none"> <li>▪ The project scope exceeds the time and resources available.</li> </ul>
Supplementary specification	<ul style="list-style-type: none"> <li>▪ The development team might not understand nonfunctional requirements: platforms, reliability, standards, and so on.</li> </ul>
Tracing use cases to implementation	<ul style="list-style-type: none"> <li>▪ Use cases might be described but not fully implemented in the system.</li> </ul>
Tracing use cases to test cases	<ul style="list-style-type: none"> <li>▪ Some use cases might not be tested, or alternative and exception conditions might not be understood, implemented, and tested.</li> </ul>
Requirements traceability	<ul style="list-style-type: none"> <li>▪ Critical requirements might be overlooked in the implementation.</li> <li>▪ The implementation might introduce requirements or features not called for in the original requirements.</li> <li>▪ A change in requirements might impact other parts of the system in unforeseen ways.</li> </ul>
Change management	<ul style="list-style-type: none"> <li>▪ New system requirements might be introduced in an uncontrolled fashion. The team might underestimate the negative impact of a change.</li> </ul>

**Principle 1: Interactive, Face-to-Face Communication Is the Cheapest and Fastest Channel for Exchanging Information** Whether eliciting requirements information from a customer or communicating that information to a team, face-to-face discussion is the best and most efficient way to communicate. If the customer is close to the team, if the customer is directly accessible, if requirements can be explained to the team directly, if the analyst can communicate directly with the customer and the team, then less documentation is needed.<sup>1</sup> However, due to the criticality of understanding requirements for the system, some requirements must be documented. Otherwise, the team bears the risk that implicit, tacit assumptions to the effect of “we all know what we are developing here” may again become a primary risk factor in the project. But certainly, fewer documents need be produced, and necessary documents—Vision documents, use cases, supplementary specifications, and the like—can be shorter and written with less specificity.

**Principle 2: Excess Methodology Weight Is Costly** This principle translates to “Do only what you have to do to be successful.” Every unnecessary process or artifact slows the team down, adds weight to the project, and diverts time and energy from essential coding and testing activities. The team must balance the cost and weight of each requirements activity with the risks listed in Table 30–1. If a particular risk is not present or likely to occur, consider deleting the corresponding artifact or activity from your process. Alternatively, think of a way to “lighten” the artifact until it’s a better fit for the risk in your particular project. Write abbreviated use cases, apply more implicit traceability, and hold fewer reviews of requirements artifacts.

**Principle 3: Larger Teams Need Heavier Methodologies** Clearly, an appropriate requirements methodology for a team of three developers who are subject matter experts and who have ready access to a customer may be entirely different than the right methodology for a team of 800 people at five different locations who are developing an integrated product line. What works for one will not work for the other. The requirements method must be scaled to the size of the team and the size of the project. However, you must not overshoot the mark either; an over-weighted method will result in lower efficiency for a team of any size.

---

1. It is important to take this notion with a grain of salt. As Philippe Kruchten pointed out to us recently, “I write to better understand what we said.”

**Principle 4: Greater Ceremony Is Appropriate for Projects with Greater Criticality** The criticality of the project may be the greatest factor in determining methodology weight. For example, it may be quite feasible to develop software for a human pacemaker’s external programming device with a two- or three-person coding team. Moreover, the work would likely be done by a development team with subject matter expertise as well as ready access to clinical experts who can describe exactly what algorithms must be implemented and why and how. However, on such a project, the cost of even a small error might be quite unacceptable since it could endanger human life. Therefore, all the intermediate artifacts that specify the use cases, algorithms, and reliability requirements must be documented in exceptional detail, and they must be reviewed and rationalized as necessary to ensure that only the “right” understanding appears in the final implementation. In such cases, a small team may need a heavyweight method. The opposite case may also be true. A noncritical project of scope sufficient to require a large team may be able to apply a lighter-weight method.

## DOCUMENTATION IS A MEANS TO AN END

Most requirements artifacts, Vision documents, use cases, and so forth—and indeed software development artifacts in general, including the code—require documentation of some kind. Given that these documents divert time and attention from essential coding and testing activities, a reasonable question to ask with respect to each one is “Do we really need to write this document at all?”

You should answer “Yes” *only* if one or more of these four criteria apply.

1. The document communicates an important understanding or agreement for instances in which simpler verbal communication is either impractical (for example, a larger or more distributed team) or would create too great a project risk (for example, a pacemaker programmer device).
2. The documentation allows new team members to come up to speed more quickly and therefore renders both current and new team members more efficient.<sup>2</sup>

---

2. In our experience, this issue is often overrated, and the team may be better off focusing new members on the “live” documentation inside the requirements, analysis and design tools, and so forth.

3. Investment in the document has an obvious long-term payoff because it will evolve, be maintained, and persist as an ongoing part of the development, testing, or maintenance activity. Examples include use case and test case artifacts, which can be used repeatedly for regression testing of future releases.
4. Some company, customer, or regulatory standard imposes a requirement for the document.

Before including a specific artifact in your requirements method, your team should ask and answer the following two questions (and no, you needn't document the answers!).

1. Does this document meet one or more of the four criteria above? If not, then skip it.
2. What is the appropriate level of specificity that can be used to satisfy the need?

With this perspective in hand, let's move on to defining a few requirements approaches that can be effective in particular project contexts. We know, of course, that projects are not all the same style and that even individual projects are not homogenous throughout. A single project might have a set of extremely critical requirements or critical subsystems interspersed with a larger number of noncritical requirements or subsystems. Each element would require a different set of methods to manage the incumbent risk. Therefore, a bit of mixing and matching will be required in almost any case, but we can still provide guidelines for choosing among a few key approaches.

## **AN EXTREME REQUIREMENTS METHOD**

In the last few years, the notion of *Extreme Programming* (XP) as originally espoused by Beck [2000] has achieved some popularity (along with a significant amount of notoriety and controversy). One can guess at what has motivated this trend. Perhaps it's a reaction to the inevitable and increasing time pressures of an increasingly efficient marketplace, or a reaction to the overzealous application of otherwise effective methodologies. Alternatively, perhaps it's a reaction to the wishes of software teams to be left alone to do what they think they do best: write code. In any case, there can be no doubt of the "buzz" that Extreme Programming has created in software circles and that the related Agile Methods movement is now creating as it attempts to add balance and practicality to the extreme approach.

Before examining how we might define an extreme requirements method, let's look at some of the key characteristics of XP.

- The scope of the application or component permits coding by a team of three to ten programmers working at one location.
- One or more customers are on site to provide constant requirements input.
- Development occurs in frequent builds or iterations, each of which is releasable and delivers incremental user functionality.
- The unit of requirements gathering is the *user story*, a chunk of functionality that provides value to the user. User stories are written by the customers on site.
- Programmers work in pairs and follow strict coding standards. They do their own unit testing and are supposed to routinely re-factor the code to keep the design simple.
- Since little attempt is made to understand or document future requirements, the code is constantly refactored (redesigned) to address changing user needs.

Let's assume you have a project scope that can be achieved by a small team working at one location. Further, let's assume that it's practical to have a customer on site during the majority of the development (an arrangement that is admittedly *not* very practical in most project contexts we've witnessed). Now, let's look at XP from the standpoint of requirements methods.

A key tenet of any effective requirements method is early and continuous user feedback. From this perspective, perhaps XP doesn't seem so extreme after all. Table 30-2 illustrates how some key tenets of XP can be used to mitigate requirements risks we've identified so far.

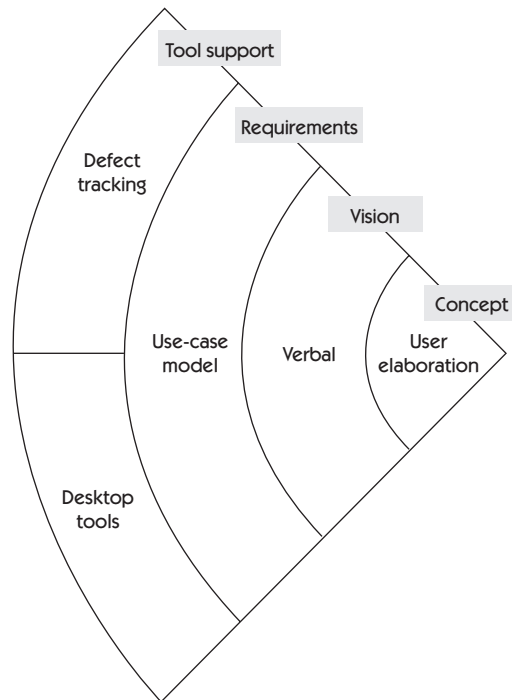
With this background, let's see if we can derive a simple, explicit requirements model that would reflect or support an XP process. Perhaps it would look like Figure 30-1 and have the characteristics described briefly below.

**Concept.** At the heart of any requirements process lives the product concept. In this case, the concept is communicated directly from the customer to the project team—verbally, frequently, and repeatedly as personnel come and go on the team.

**Vision.** The vision carries the product concept, both short-term and long-term. A Delta Vision document typically describes the new features and use

**Table 30–2** Applying Extreme Programming Principles to Requirements Risk Mitigation

Extreme Programming Principle	Mitigated Requirements Risk
Application or component scope is such that three to ten programmers at one location can do the coding.	Constant informal communication can minimize or eliminate much requirements documentation.
One or more customers are on site to provide constant requirements input.	Constant customer input and feedback dramatically reduces requirements-related risk.
Development occurs in frequent builds or iterations, each of which is releasable and delivers incremental user functionality.	Customer value feedback is almost immediate; this ship can't go too far off course.
The unit of requirements gathering is the user story, a bite of functionality that provides value to the user. Customers on site write user stories.	A use case describes sequences of events that deliver value to a user, as written by the developer with user input. User stories are often short descriptions of a path or scenario of a use case. Each captures the same basic intent—how the user interacts with the system to get something done.

**Figure 30–1** An extreme requirements method

cases to be implemented in a specific release. In XP, this document may not exist. We are dependent on the customer's ability to tell us what the product needs to do now and what it needs to do later, and we are dependent on the development team to make the right architectural decisions now—for both now and later. Whether or not this can be made to work in practice depends on a number of project factors and the relative risk the team is willing to take; you can't say for certain that it couldn't work, at least for some project scenarios.<sup>3</sup> Therefore, we'll leave this artifact out of our extreme requirements method.

**Requirements.** Another principal tenet of our text is that the use-case model carries the majority of functional requirements. It describes who uses the system and how they use it to accomplish their objectives. XP recommends the use of simple “stories” that are not unlike use cases but are typically shorter (they appear to be more like a use-case scenario) and at a higher level of abstraction. However, we recommend that there *always* be a use-case model, even if it's a simple, nongraphical summary of the key user stories that are implemented and what class of user implements them. We'd insist on this use-case model, even for our extreme requirements method.

**Supplementary Specification/Nonfunctional Requirements.** XP has no obvious placeholder for these items, perhaps because there are not very many, or perhaps the thinking is that they can be assumed or understood without mention. On the other hand, perhaps customers communicate these requirements directly to programmers whose work is affected by them. Seems a bit risky, but if that's not where the risk lies in your project, so be it; we'll leave this artifact out of our extreme requirements method.

**Tooling.** The tools of XP are whiteboards and desktop tools, such as spreadsheets with itemized user stories, priorities, and so forth. However, defects will naturally occur, and although XP is quiet on the tooling subject, let's assume we can add a tracking database of some kind to keep track of all these stories—perhaps their status as well as defects that will naturally occur and must be traded off with future enhancements.

---

3. As we said, the method is not without its critics. One reviewer noted the big drawback of the “one user story at a time” approach is the total lack of architectural work. If your initial assumption is wrong, you have to refactor the architecture one user story at a time. You build a whole system, and the nth story is, “OK, this is fine for one user. Now, let's make it work for 3,000.”

With these simple documents, practices, and tools, we've defined an extreme requirements method that can work in appropriate, albeit somewhat extreme, circumstances.

## AN AGILE REQUIREMENTS METHOD

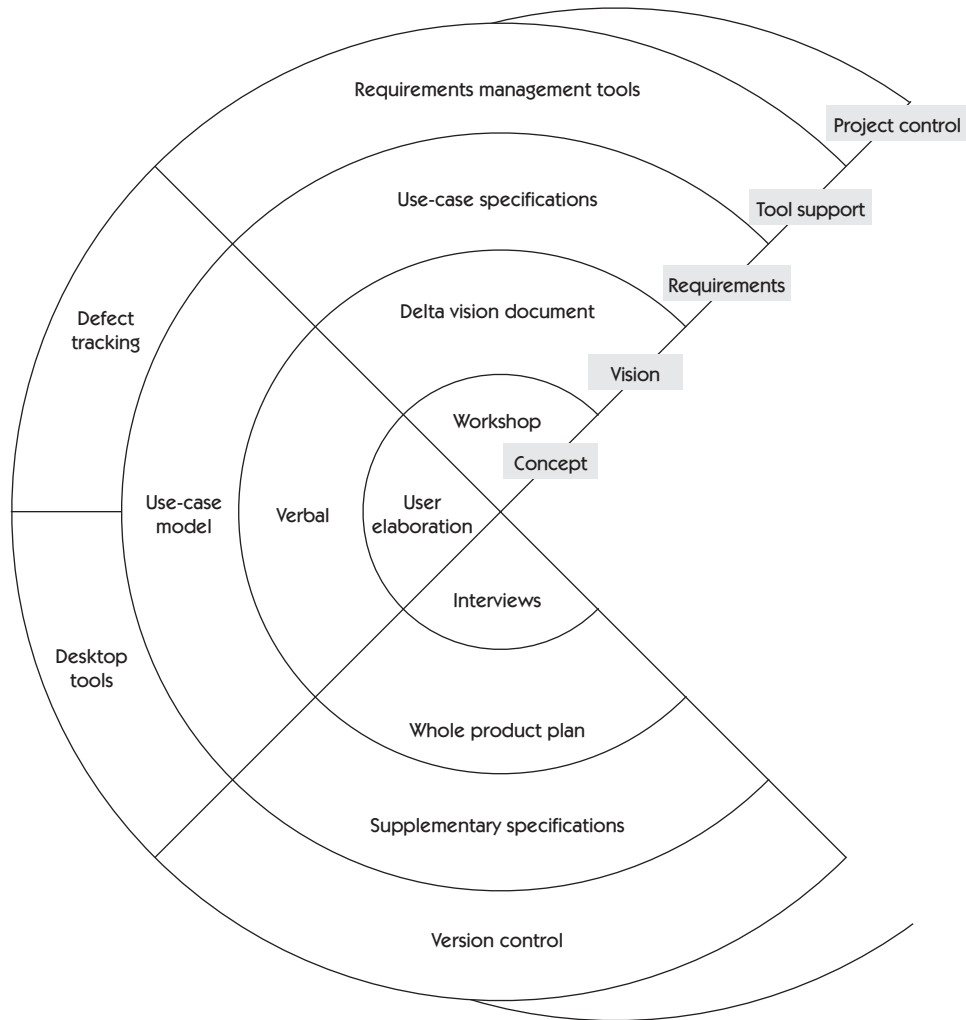
But what if your customer can't be located on site? What if you are developing a new class of products for which no current customers exist? What if the concepts are so innovative that customers can't envision what stories they would fulfill? What if your system has to be integrated with either new systems or other existing systems? What if more than three to ten people are required? What if your system is so complex that it must be considered as a "system of systems"—with each system imposing requirements on others? What if some of your team members work from remote sites? What if a few potential failure modes are economically unacceptable? What then?

Then you will need a heavier method, one that can address the additional risks in your project context. You will need a method that looks more like the agile requirements method depicted in Figure 30–2. Its characteristics are described briefly below.

**Concept.** In the agile requirements method, the root of the project is still the concept, but that concept is tested and elaborated by a number of means, including requirements workshops or interviews with prospective customers.

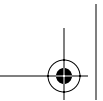
**Vision.** The vision is no longer only verbal; it is defined incrementally in the Delta Vision document, which describes the new features to be implemented in a specific release. The whole product plan describes the other elements of your successful solution: the commercial and support factors, licensing requirements, and other factors that are keys to success.

**Requirements.** The use-case model diagram defines the use cases at the highest level of abstraction. In addition, in this more robust method, each use case has a specification that elaborates the sequence of events, the pre- and post-conditions, and the exceptions and alternative flows. The use-case specifications will likely be written at different levels of detail. Some areas are more critical than others are; other areas are more innovative and require further definition before coding begins. Still other areas are straightforward extensions to known or existing features and need little additional specification.



**Figure 30-2** An agile requirements method

**Supplementary Specification/Nonfunctional Requirements.** Your application may run on multiple operating systems, support multiple databases, integrate with a customer application, or have specific requirements for security or user access. Perhaps external standards are imposed on it, or perhaps a host of performance requirements must be individually identified, discussed, agreed to, and tested. If so, the supplementary specification contains



this information, and it is an integral artifact to an agile requirements management method.

**Tooling.** As the project complexity grows, so do the tooling requirements, and the team may find it beneficial to add a requirements tool for capturing and prioritizing the information or automatically creating a use-case summary from the developed use cases. The more people working on the project, and the more locations they work from, the more important version control becomes, both for the code itself and for the use cases and other requirements artifacts that define the system being built.

With some practical and modest extensions to our extreme requirements method, we've now defined a practical agile requirements method, one that is already well proven in a number of real-world projects.

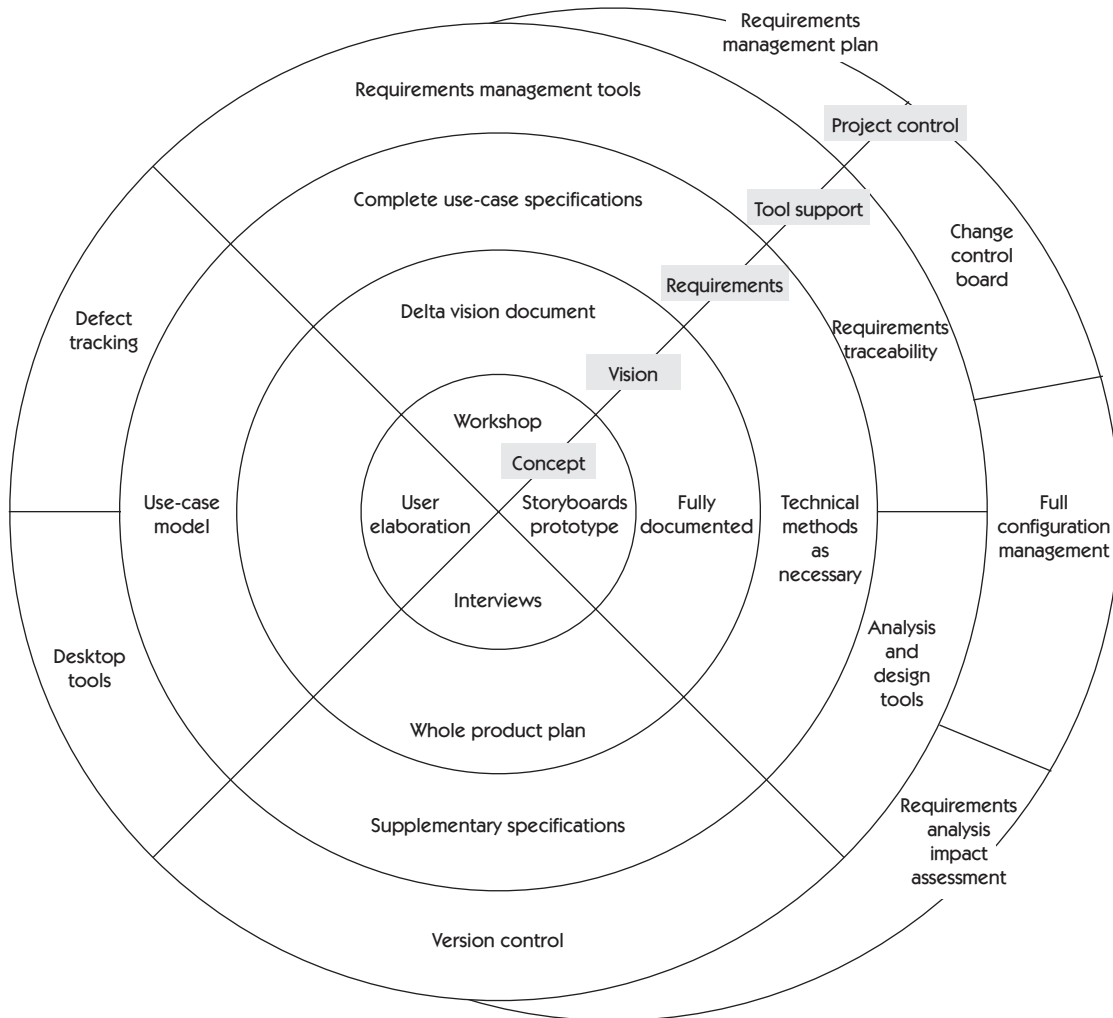
## A ROBUST REQUIREMENTS METHOD

But what if you are developing the pacemaker programmer we described above? What if your teams are developing six integrated products for a product family that is synchronized and released twice a year? You employ 800 developers in six locations worldwide, and yet your products must work together. Alternatively, what if you are a telecommunications company and the success of your company will be determined by the success of a third-generation digital switching system that will be based on the efforts of 1,000 programmers spanning a time measured in years? What then?

*Then you will need a truly robust requirements method.* One that scales to the challenge at hand. One that can be tailored to deliver extremely reliable products in critical areas. One that allows developers in other countries to understand the requirements imposed on the subsystem they are building. One that can help assure you that your system satisfies the hundreds of use cases and thousands of functional and nonfunctional requirements necessary for your application to work with other systems and applications—seamlessly, reliably, and flawlessly.

So now we come full circle to the robust requirements management method expressed in Figure 30–3. Its characteristics are briefly explored below.

**Concept.** Given the complexity of the application itself and the likelihood that few, if any, features can actually be implemented and released before a significant amount of architectural underpinnings are developed and implemented,



**Figure 30-3** A robust requirements method

we add a range of concept validation techniques, storyboards, prototypes, architectural models, and the like. Each will bring us closer to our goal of understanding the intended behavior of the system we are about to build.

**Vision.** In order to assure understanding among a large number of stakeholders, developers, and testers, the vision, both short-term and long-term, must be documented. It must be sufficiently long-range for the architects

and designers to design and implement the right architecture to support current and future features and use cases. The whole product plan should be extended to describe potential variations in purchase configurations and likely customer deployment options. The plan should also define supported revision levels of compatible applications.

**Requirements.** The use cases are elaborated as necessary so that prospective users can validate the implementation concepts. This ensures that all critical requirements will be implemented in a way that helps assure their utility and fitness. Because some elements of the application are critical, all alternative sequences of events are discussed and described. Pre- and post-conditions are specified as clearly and unambiguously as possible. Additional, technical specification methods (analysis models, activity diagrams, message sequence diagrams) are used to describe more clearly how the system does what it does and when it does it.

**Supplementary Specification/Nonfunctional Requirements.** The supplementary specification is as complete as possible. All platforms; application compatibility issues; applicable standards; branding and copyright requirements; and performance, usability, reliability, and supporting requirements are defined.

**Tooling.** Larger, more distributed teams require industrial-strength software tooling. Analysis and design tools further specific system behavior, both internal and external. Multisite configuration management systems are employed. Requirements tools both support requirements traceability from features through use cases and into test cases and track changes and change history. The defect tracking system extends to support users, including customers, from any location.

**Project Control.** Larger projects require higher levels of project support and control. Requirements dashboards are built so that teams can monitor and synchronize interdependent use-case implementations. A change control board is created to weigh and decide on possible requirements additions and defect fixes. Requirements analysis and impact assessment activities are performed to help understand the impact of proposed changes and additions.

Taken together, these techniques and activities in our robust requirements method help assure that this new system—in which many tens or hundreds of person-years have been invested and which will touch the lives of thousands of users across the globe—is accurate, reliable, safe, and well suited for its intended purpose.

## SUMMARY

In this chapter, we reinforced the concept that the project methodology is designed solely to assure that we mitigate the risks present in our project environment. If in our projects we focus too much on methodology, we add overhead and burden the team with unnecessary activities. If we aren't careful, we'll become slow, expensive, and eventually uncompetitive. Some other team will get the next project, or some other company will get our next customer. If we focus too little on methodology, we assume too much risk on the part of our company or our customers, with perhaps even more severe consequences.

To manage this risk, we looked at three prototypical requirements methods: an *extreme requirements method*, an *agile requirements method*, and a *robust requirements method*, each of which is suitable for a particular project context. Yet we recognize that every project is unique, and every customer and every application is different; therefore, your optimal requirements method will likely be none of the above. Perhaps it will be some obvious hybrid, or perhaps a variant we did not explore. In any case, your team's job is to select the right requirements method for your next project while keeping the project as agile as possible.

Finally, with this understanding behind us, we can now move on to creating that specific requirements prescription you've all been asking for.

