Agile Requirements Methods

by **Dean Leffingwell**
Software Entrepreneur and
Former Rational Executive

To ensure that their software teams build the right software the right way, many companies turn to standard processes such as Rational Software's Rational Unified Process® (RUP®), a comprehensive set of industry best practices that provide proven methods and guidelines for developing software applications. Through the application of use cases and other requirements techniques, the RUP helps development teams build the right software by helping them understand what user needs their products must fulfill. Moreover, the RUP and many other contemporary software processes prescribe a software lifecycle method that is iterative and incremental, as this method helps teams address the risk inherent in a new development effort more effectively than did earlier, more rigid "waterfall" process approaches. Risk can originate from a variety of sources: technology and scale, deficient people skills, unachievable scope or timeline issues, potential health or safety hazards defects, and so on. Experience has proved repeatedly that addressing these risks early in the lifecycle is a key factor in producing successful project outcomes, and requirements management is one very effective way to accomplish this.

Mitigating Requirements Risk with Effective Requirements Practices

In our book *Managing Software Requirements: A Unified Approach*,[1] Don Widrig and I described a comprehensive set of practices intended to help teams more effectively manage software requirements imposed on a system under development. As the systems 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.

But 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 lifecycle 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*. Each of the requirements techniques we describe in our book, as well as those recommended in the RUP, was developed solely to address one or more specific types of requirements-related risks. Table 1 summarizes these techniques, along with the nature and type of risks that each is intended to mitigate.

Table 1: Requirements Techniques Address Specific Project Risks

Technique	Risk Addressed
Interviewing	- The development team might not understand who the real stakeholders are. - The team might not understand the basic needs of one or more stakeholders.
Requirements Workshops	- The system might not appropriately address classes of specific user needs. - Lack of consensus among key stakeholders might prevent convergence on a set of requirements.
Brainstorming and Idea Reduction	- The team might not discover key needs or prospective innovative features. - Priorities are not well established, and a plethora of features obscures the fundamental "must haves."
Storyboards	- The prospective implementation misses the mark. - The approach is too hard to use or understand, or the operation's business purpose is lost in the planned implementation.
Use Cases	- Users might not feel they have a stake in the implementation process. - Implementation fails to fulfill basic user needs in some way because some features are missing or because of poor usability or error and exception handling, etc.
Vision Document	- The development team does not really understand what system they are trying to build, or what user needs or industry problem it addresses. - Lack of longer term vision causes poor planning and poor architecture and design decisions.
Whole Product Plan	- The solution might lack commercial elements necessary for successful adoption.
Scoping Activities	- The project scope exceeds the time and resources available.
Supplementary Specification	- The development team might not understand non-functional requirements: platforms, reliability, standards, and so on.

Trace Use Cases to Implementation	- Use cases might be described but not fully implemented in the system.
Trace Use Cases to Test Cases	- Some use cases might not be tested, or alternative and exception conditions might not be understood, implemented, and tested.
Requirements Traceability	- Critical requirements might be overlooked in the implementation. - The implementation might introduce requirements or features not called for in the original requirements. - A change in requirements might impact other parts of the system in unforeseen ways.
Change Management	- New system requirements might be introduced in an uncontrolled fashion. - The team might underestimate the negative impact of a change.

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, Alistair Cockburn identifies four major principles to apply when designing and evaluating 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.[2]

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.

Principle #1: Interactive, Face-to-Face Communication Is the Cheapest and Fastest Channel for Exchanging Information

Whether eliciting requirements information from a customer or user, or communicating that information to a team, face-to-face is the best and most efficient way to communicate. If the customer is close to the team and directly accessible, if the customer can explain requirements directly to the team, and if the *analyst* can communicate directly with the customer and the team, then less documentation is needed[3] -- although critical requirements must still be documented. Otherwise, there is a danger that the tacit assumption "We all know what we are developing here" may become a primary risk factor for the project team. But certainly the team can get by with fewer, highly necessary documents -- Vision documents, use cases, supplementary specs, and the like -- and these can be shorter and less detailed.

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 requirement activity with the risks listed in Table 1. If a particular risk is not present or likely, then 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, as an over-weighted method will result in lower efficiency for a team of any size.

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 some subject matter expertise as well as ready access to clinical experts who can describe exactly what algorithms must be implemented. However, on such a project, the cost of even a small error might be quite unacceptable, and even entail loss of 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 vetted as necessary to ensure that only the "right" understanding appears in the final implementation. In such cases, therefore, a small team would need a heavyweight method. And conversely, a non-critical application with sufficient scope to require a larger team might very well be able to use a lighter method.

Documentation Is a Means to an End

Most requirements process artifacts, Vision documents, use cases, and so forth -- and indeed most software development artifacts in general, require non-code 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 (larger or more distributed team) or would create too great a project risk (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.[4]
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 again and again for regression testing of future releases.
4. A requirement for the document is imposed by some company, customer, or regulatory standard.

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!).

- Does this document meet one or more of the four criteria above? If not, then skip it.
- What is the minimum level of specificity that can be used to satisfy the need? If you do not need the level the project calls for, then either do not use it, or use an abbreviated version.

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, or 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 non-critical requirements or subsystems. Each element would require a different set of methods to manage the incumbent risk. So 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 as originally espoused by Beck[5] 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. Or 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 methods have

created in software circles, and that the "agile methods" movement is now creating, as it attempts to add balance and practicality to the extreme approach. Let's look at some of the key characteristics of XP and then examine how we might define an Extreme Requirements Method that would be compatible with this approach.

Three Points to Remember About Method

- The purpose of the software development method is to mitigate 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 requirements method must be tailored to the particular project.

1. The scope of the application or component permits coding by a team of three to ten programmers working at one location.
2. One or more customers are on site to provide constant requirements input.
3. Development occurs in frequent builds, or iterations, each of which is releasable and delivers incremental user functionality.
4. The unit of requirements gathering is the "User Story," a chunk of functionality that provides value to the user. User stories are written by customers on site.
5. Programmers work in pairs and follow strict coding standards. They do their own unit testing and are supposed to provide constant refactoring of the code to keep the design simple.
6. Since little attempt is made to understand or document future requirements, the code is constantly re-factored (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 admittedy *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. When looked at from this perspective, perhaps XP doesn't seem so extreme after all. Table 2 illustrates how some key tenets of XP can be used to mitigate requirements risks we've identified so far.

Table 2: Applying XP Principles to Requirements Risk Mitigation

XP Principle	Mitigated Requirements Risk
Application or component scope is such that the coding can be done by three to ten programmers at one location.	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 chunk of functionality that provides value to the user. User stories are written by customers on site.	A use case is "a sequence of events that delivers value to a user." Can user stories and use cases be all that different? If users contribute to both of them, then how far apart can they be?

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 1 and have the following characteristics.

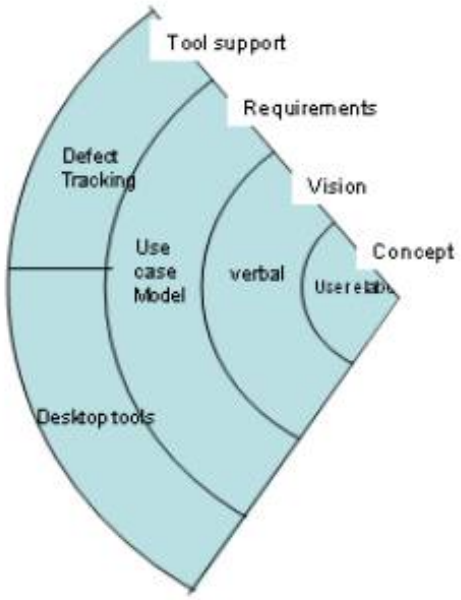

Figure 1: Extreme Programming Requirements Model

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 change.

Vision. As explained in *Managing Software Requirements*[6] and in the RUP, the *Vision* carries the product concept, both short term and long term. A "Delta Vision document" typically describes the new features and use 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 risks the team is willing to take; you can't say for certain that it couldn't work, at least for some project scenarios.[7] So we'll leave this artifact out of our extreme requirements method.

Requirements. Another principal tenet of our text and the RUP 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 perhaps shorter and at a higher level of abstraction. However, we recommend that there *always* be a use-case model, even if it's a simple, non-graphical 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 method.

Supplementary Spec/Non-Functional Requirements. XP has no obvious placeholder for these items, perhaps because there are not very many, or the thinking is that they can be assumed or understood without mention. Or 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 on your project, so be it; we'll leave this artifact out of our extreme method.

Tooling. The tools of XP are whiteboards and desktop tools, such as spreadsheets with itemized user stories and 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 occur and must be traded off with future enhancements.

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 ten to twenty 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 more robust method. One that can address the additional risks in your project context. Then you will need a method that looks more like the agile method depicted in Figure 2.

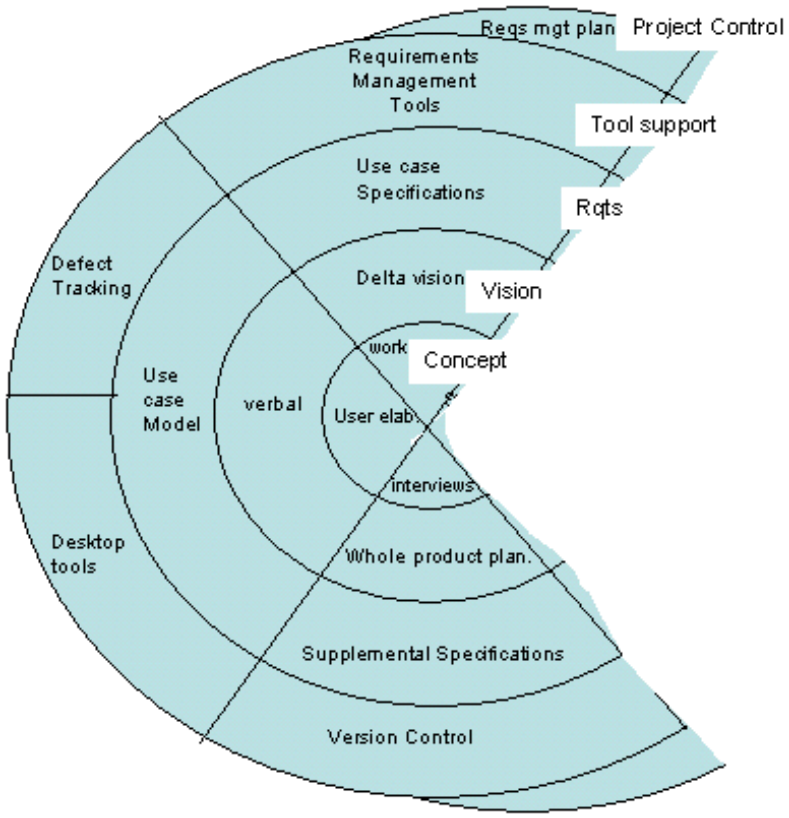

Figure 2: An Agile Requirements Approach

Concept. In the agile 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 and use cases 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 differing levels of detail. Some areas are more critical than others; 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.

Supplementary Spec/Non-Functional 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 upon it, or a host of performance requirements that must be individually identified, discussed, agreed to, and tested. If so, then the supplementary specification contains this information, and it is an integral artifact to an

agile software 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. And the more people that work 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.

Well now, with some practical and modest extensions to our extreme method, we've now defined a practical and *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. Or 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 thousands of 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 that are 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 3.

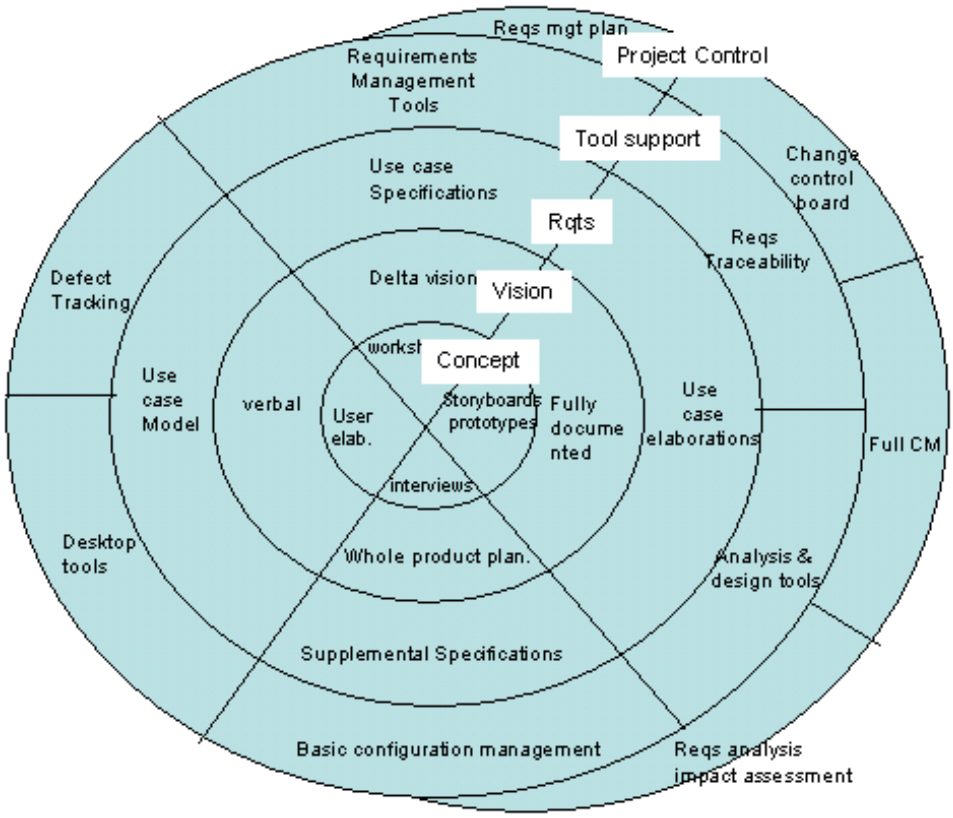

Figure 3: A Robust Requirements Management Method

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, we want to add a range of concept validation techniques. 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 amongst a large number of stakeholders, developers, and testers, the Vision, both near term and longer 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 the application is critical, all alternative sequences of events are discussed and described. Pre-and post-conditions are specified, and are as clear and unambiguous as possible. Additional, more formal techniques -- 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 Spec/Non-Functional 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. Multi-site configuration management systems are employed. Requirements tools support requirements traceability from features through use cases and into test cases. The defect tracking system extends to support users 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 constituted to weigh and take decisions upon 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 management method help assure that this new system -- in which many tens or hundreds of man 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 article, we've reinforced the concept that the project methodology is designed solely to assure that we mitigate the risks present in our project environment. Too much methodology and 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. Too little methodology, and 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've 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. And 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 itwill be some obvious hybrid, or perhaps a variant we did not explore. But if you are properly prepared, then you can select the right requirements method for your next project.

References

Rational Unified Process 2001. Rational Software Corporation, 2001.

Dean Leffingwell and Don Widrig, *Managing Software Requirements: A Unified Approach*. Addison-Wesley, 1999.

Kent Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

Alistair Cockburn, *Agile Software Development*. Addison-Wesley, 2002.

Notes

[1] Dean Leffingwell and Don Widrig, *Managing Software Requirements: A Unified Approach*. Addison-Wesley, 1999.

[2] Alistair Cockburn, *Agile Software Development*. Addison Wesley, 2002, pp. 149-153.

[3] It is important to take this notion with a grain of salt. As Philippe Kruchten points out, "I write to better understand what we said."

[4] 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.

[5] Kent Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[6] Leffingwell and Widrig, *Op.Cit*.

[7] As we said, the method is not without its critics. One reviewer noted the big drawback of the "one user story at a time," is the total lack of architectural work. If your initial assumption is wrong, you have to re-factor architecture one user story at a time. You build a whole system, and the n-1th story is, "OK, this is fine for one user. Now, let us make it work for 3,000."

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!