Chapter 7

Requirements Specification and Agile Methodologies

Introduction to Agile Methodologies*

Agile methodologies are a family of nontraditional software development strategies that have captured the imagination of many who are leery of traditional, process-laden approaches. Agile methodologies are characterized by their lack of rigid process, although this fact does not mean that agile methodologies, when correctly employed, are not rigorous or suitable for industrial applications. They are. What is characteristically missing from agile approaches, however, are "cookbook" solutions (e.g., those prescribed in the Project Management Body of Knowledge [PMBOK Guide]) and are therefore sometimes called "lightweight" approaches.

Agile methodologies are traditionally applied to software engineering. There are elements of agile methodologies that can be applied to the engineering of systems (in particular, the human considerations), however, agile is not typically used in nonsoftware systems. This is so because agile methodologies depend on a series of rapid, nonthrowaway prototypes, an approach that is not practical in hardware-based systems. In any case, the nonsoftware engineer can still benefit from this chapter because agile methodologies are increasingly being employed, and because the mindset of the agile software engineer includes some healthy perspectives.

* Some of this section has been excerpted from Phillip A. Laplante, *What Every Engineer Needs to Know about Software Engineering*, Boca Raton, FL: CRC/Taylor & Francis, 2006, with permission.

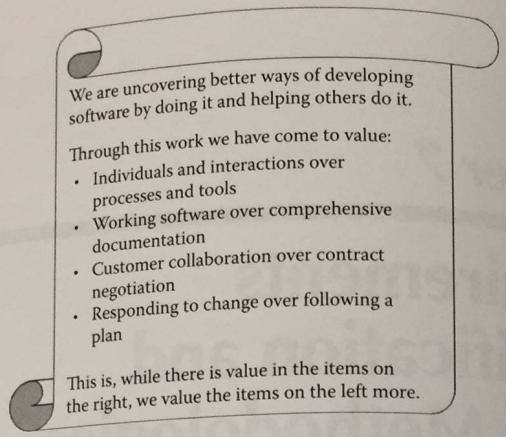

Figure 7.1 Manifesto for agile software development. (Adapted from K. Beck, *Extreme Programming Explained: Embrace Change*. Chicago: Longman Higher Education, 2000.)

In order to fully understand the nature of agile methodologies, we need to examine a document called the Agile Manifesto and the principles behind it. The Agile Manifesto was introduced by a number of leading proponents of agile methodologies in order to explain their philosophy (see Figure 7.1).

Signatories to the Agile Manifesto include many luminaries of modern software engineering practice such as Kent Beck, Mike Beedle, Alistair Cockburn, Ward Cunningham, Martin Fowler, Jim Highsmith, Ron Jeffries, Brian Marick, Robert Martin, Steve Mellor, and Ken Schwaber. Underlying the Agile Manifesto is a set of principles. Look at the principles below, noting the emphasis on those aspects that focus on requirements engineering, which we set in italics.

Principles behind Agile Manifesto

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity—the art of maximizing the amount of work not done—is essential. "Do the simplest thing that could possibly work."

The best architectures, requirements, and designs emerge from self-organizing teams. (Beck 2000)

Notice how the principles acknowledge and embrace the notion that requirements change throughout the process. Also, the agile principles emphasize frequent personal communication (this feature is beneficial in the engineering of nonsoftware systems too). The highlighted features of requirements engineering in agile process models differ from the "traditional" waterfall and more modern models such as iterative, evolutionary, or spiral development. These other models favor a great deal of up-front work on the requirements engineering process and the production of often voluminous requirements specifications documents.

Agile software development methods are a subset of iterative methods* that focus on embracing change and stress collaboration and early product delivery, while maintaining quality. Working code is considered the true artifact of the development process. Models, plans, and documentation are important and have their value, but exist only to support the development of working software, in contrast with the other approaches already discussed. However, this does not mean that an agile development approach is a free-for-all. There are very clear practices and principles that agile methodologists must embrace.

Agile methods are adaptive rather than predictive. This approach differs significantly from those models previously discussed, models that emphasize planning the software in great detail over a long period of time and for which significant changes in the software requirements specification can be problematic. Agile methods are a response to the common problem of constantly changing requirements that can bog down the more "ceremonial" up-front design approaches, which focus heavily on documentation at the start.

Agile methods are also "people-oriented" rather than process-oriented. This means they explicitly make a point of trying to make development "fun."

* Most people define agile methodologies as being incremental. But incremental development implies that the features and schedule of each delivered version are planned. In some cases, agile methodologies tend to lead to versions with feature sets and delivery dates that are almost always not as planned.

Presumably, this is because writing software requirements specifications and software design descriptions is onerous and hence, to be minimized.

Agile methodologies often have strange names like Crystal, Extreme Programming (XP), and Scrum. Other agile methods include Dynamic Systems Development Method (DSDM), Feature-Driven Development, and adaptive programming, and there are many more. We look more closely at two of these, XP and Scrum.

Extreme Programming (XP)

Extreme Programming* is one of the most widely used agile methodologies. XP is traditionally targeted toward smaller development teams and requires relatively few detailed artifacts. XP takes an iterative approach to its development cycles. We can visualize the difference in process between a traditional waterfall model, iterative models, and XP (Figure 7.2). Whereas an evolutionary or iterative method may still have distinct requirements analysis, design, implementation, and testing phases similar to the waterfall method, XP treats these activities as being interrelated and continuous.

XP promotes a set of 12 core practices that help developers respond to and embrace inevitable change. The practices can be grouped according to four practice areas:

Planning
Coding
Designing
Testing

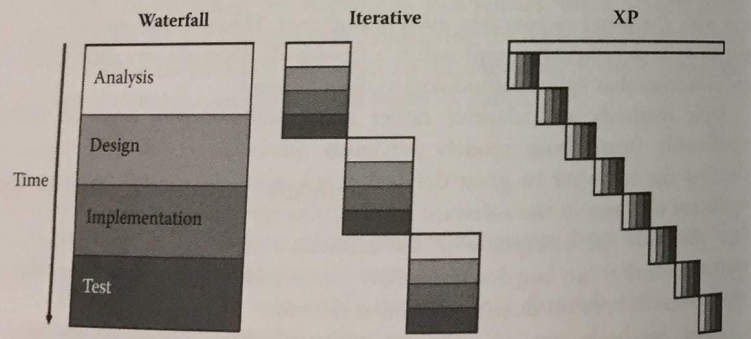

Figure 7.2 Comparison of waterfall, iterative, and XP development cycles. (Adapted from K. Beck, *Extreme Programming Explained: Embrace Change*, Chicago: Longman Higher Education, 2000.)

* Extreme Programming is sometimes written "eXtreme Programming" to highlight the "XP."

Some of the distinctive planning features of XP include holding daily stand-up meetings,* making frequent small releases, and moving people around. Coding practices include having the customer constantly available, coding the unit test cases first, and employing pair-programming (a unique coding strategy where two developers work on the same code together). Removal of the territorial ownership of any code unit is another feature of XP.

Design practices include looking for the simplest solutions first, avoiding too much planning for future growth (speculative generality), and refactoring the code (improving its structure) continuously.

Testing practices include creating new test cases whenever a bug is found and having unit tests for all code, possibly using such frameworks as XUnit.

Scrum

Scrum, which is named after a particularly contentious point in a rugby match, enables self-organizing teams by encouraging verbal communication across all team members and across all stakeholders. The fundamental principle of Scrum is "empirical challenges cannot be addressed successfully in a traditional 'process control' manner," (Schwaber and Beedle 2001) meaning that the problem cannot be fully understood or defined. Scrum encourages self-organization by fostering high-quality communication among all stakeholders. In this case it is implicit that the problem cannot be fully understood or defined (it may be a wicked problem). And the focus in Scrum is on maximizing the team's ability to respond in an agile manner to emerging challenges.

Scrum features a living backlog of prioritized work to be done. Completion of a largely fixed set of backlog items occurs in a series of short (approximately 30-day) iterations or sprints. Each day, a brief (e.g., 15-minute) meeting or Scrum is held in which progress is explained, upcoming work is described, and impediments are raised. A brief planning session occurs at the start of each sprint to define the backlog items to be completed. A brief post mortem or heartbeat retrospective occurs at the end of the sprint (Figure 7.3).

A "ScrumMaster" removes obstacles or impediments to each sprint. The ScrumMaster is not the leader of the team (as they are self-organizing) but acts as a productivity buffer between the team and any destabilizing influences. In some organizations the role of the ScrumMaster can cause confusion. For example, if two members of a Scrum team are not working well together, it might be expected by a senior manager that the ScrumMaster "fix" the problem. Fixing team dysfunction is not the role of the ScrumMaster. Personnel problems need to be resolved by the

* Stand-up meetings are 10 minutes or less long where participants literally stand up and give an oral report of the previous day's activities and plans for that day.

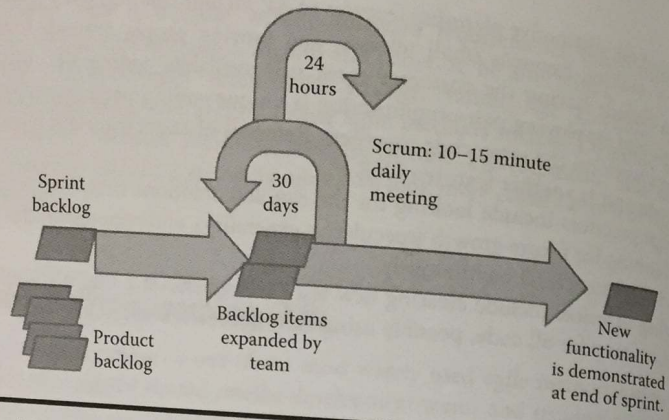

Figure 7.3 The Scrum development process. (From B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide to the Perplexed*, Reading, MA: Addison-Wesley, 2003, which itself was adapted from K. Schwaber and M. Beedle, *Agile Software Development with SCRUM*, Upper Saddle River, NJ: Prentice Hall, 2001.)

line managers to whom the involved parties report. This scenario illustrates the need for institutionwide education about agile methodologies when such approaches are going to be employed.

Requirements Engineering for Agile Methodologies

A major difference between traditional (here we mean, waterfall, evolutionary, iterative, spiral, etc.) and agile methodologies is in the gathering of requirements. In fact, some proponents of agile methodologies use the supposed advantages of agile requirements engineering practices as a selling point for agile methods. Requirements engineering approaches for agile methodologies tend to be much more informal.

Another difference is in the timing of the requirements engineering activities. In traditional systems and software engineering, requirements are gathered, analyzed, refined, and so on at the front end of the process. In agile methods, requirements engineering is an ongoing activity; that is, requirements are refined and discovered with each system build. Even in spiral methodologies, where prototyping is used for requirements refinement, requirements engineering occurs much less so downstream in the development process.

Customers are constantly involved in requirements discovery and refinement in agile methods. All systems developers are involved in the requirements engineering activity and each can and should have regular interaction with customers.

In traditional approaches, the customer has less involvement once the requirements specification has been written and approved, and typically, the involvement is often not with the systems developers.

Presumably, the agile approach to requirements engineering is much more invulnerable to changes throughout the process (remember, "embrace change") than in traditional software engineering.

General Practices in Agile Methodologies

Cao and Ramesh (2008) studied 16 software development organizations that were using either XP or Scrum (or both) and uncovered seven requirements engineering practices that were "agile" in nature.

1. Face-to-face communications (over written specifications)
2. Iterative requirements engineering
3. Extreme prioritization (ongoing prioritization rather than once at the start, and prioritization is based primarily on business value)
4. Constant planning
5. Prototyping
6. Test-driven development
7. Reviews and tests

Some of these practices can be found in nonagile development (e.g., test-driven development and prototyping), but these seven practices were consistent across all of the organizations studied.

As opposed to the fundamental software requirements specification, the fundamental artifact in agile methods is a stack of constantly evolving and refining requirements. These requirements are generally in the form of user stories. In any case, these requirements are generated by the customer and prioritized: the higher the level of detail in the requirement, the higher the priority is. As new requirements are discovered, they are added to the stack and the stack is reshuffled in order to preserve prioritization.

There are no proscriptions on adding, changing, or removing requirements from the list at any time (which gives the customer tremendous freedom). Of course, once the system is built, or likely while it is being built, the stack of user stories can be converted to a conventional software requirements specification for system maintenance and other conventional purposes.

Example Application of Agile Software Development

Consider the pet store point of sale system. Suppose that we wanted to use an agile software development methodology, specifically, Scrum, to develop this system. Let's look at what the requirements engineering process might look like.

Suppose that we had a five-person development team, including the ScrumMaster. For simplicity, let's assume that the only stakeholders are the pet store owner (who is really the customer, because he is paying for the system), store customers, cashiers, and accountants. We select class representatives for the store customers, cashiers, and accountants. Collectively, let's call them the "stakeholder panel." Now consider the following activities:

(A) The ScrumMaster organizes a set of meetings between the development team and the customer (the store owner) to get a basic understanding of the system requirements, constraints, and ground rules. Based on these discussions, the ScrumMaster organizes a set of additional elicitation activities, possibly including structured interviews, card sorting, and focus groups with the other stakeholders. A QFD study is used as a validation activity (because there are likely many other POS systems that can be compared). The goal of these activities is to produce a set of prioritized user stories (let's say around 100), including effort estimates for each. But the customer has told us that he wants a "cutting edge" system, so we know that as the system goes through development, the customer and other stakeholders are going to want new or different functionality.

(B) The development team analyzes the user stories and selects a subset of these; let's say 10, for the first sprint. These 10 would be the most "user-facing" stories, leaving as much back-end processing for later as possible.

(C) The development team consults with the stakeholders again to get further clarity for these 10 user stories before beginning development.

(D) Each day starts with a Scrum. Developers use test-driven design to derive the system design organically as well as unit test cases. The ScrumMaster facilitates communications between the development team and the various stakeholders so that stakeholder facing questions can be answered quickly. After approximately 30 days, the first sprint is completed.

(E) The stakeholder panel is presented with the system built during the sprint for feedback. This feedback will include changes to the current system increment, new features to be added, and possibly, features to be omitted.

(F) Based on this feedback, the development team creates new user stories and adds them to the backlog, modifies other user stories as needed, and reshuffles the backlog.

(G) The development team selects a new set of user stories (say 10) from the backlog for the next 30-day sprint, and process steps (C) through (F) are repeated until the backlog is exhausted.

Additional acceptance testing may be conducted, depending on the terms of the contract with the customer. This testing would be based on a set of criteria developed during the contract negotiation. Such acceptance testing based on a requirements specification is usual for conventional development, but atypical for agile development.

There were 100 user stories originally, but even if we schedule 10 user stories per sprint (month), the total development time is likely to be longer than 10 months. This is because new user stories are being added, and old ones are being changed. Of course, the ScrumMaster is tracking the project velocity metrics (i.e., the number of user stories implemented each week, sprint, and over the course of the project) during each sprint in order to refine the project completion estimates.

When Is Agile Recommended?*

In Marinelli's (2008) survey less than 2% of respondents reported using agile methodologies. Based on other findings of the 2008 survey, however, such as the increased use of user stories and use cases, it seems that companies are adopting certain agile practices if not the agile methodologies. Since then the use of agile methodologies seems to be increasing. For example, in a Forrester/Dr. Dobbs developer survey, 35% of respondents reported using some kind of agile methodology (West et al. 2010). In addition, open source software, which is pervasively used in many enterprises, is often developed in a manner analogous to agile development.

But the question arises, when should agile development methodologies be used? Boehm and Turner (2003) suggest that the way to answer the question is to look at the project along a continuum of five dimensions: size (in terms of number of personnel involved), system criticality, personnel skill level, dynamism (anticipated number of system changes over some time interval), and organizational culture (whether the organization thrives on chaos or order). In Figure 7.4 as project characteristics tend away from the center of the diagram, the likelihood of succeeding using agile methodologies decreases.

Therefore, projects assessed in the innermost circle are likely candidates for agile approaches, those in the second circle (but not in the inner circle) are marginal, and those outside the second circle are not good candidates for agile approaches.

Agile Requirements Best Practices

Scott Ambler (2007) suggests the following best practices for requirements engineering using agile methods. Many of the practices follow directly from the principles behind the Agile Manifesto:

- Have active stakeholder participation
- Use inclusive (stakeholder) models
- Take a breadth-first approach
- Model "storm" details (highly volatile requirements) just in time

* This section is adapted from Laplante (2006) with permission.

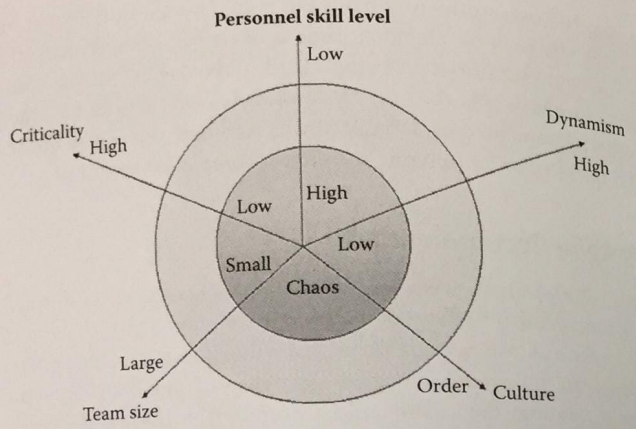

Figure 7.4 Balancing agility and discipline. (Adapted from B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide to the Perplexed*, Reading, MA: Addison-Wesley, 2003.)

- Implement requirements; do not document them
- Create platform-independent requirements to a point
- Remember that "smaller is better"
- Question traceability
- Explain the techniques
- Adopt stakeholder terminology
- Keep it fun
- Obtain management support
- Turn stakeholders into developers
- Treat requirements like a prioritized stack
- Have frequent personal interaction
- Make frequent delivery of software
- Express requirements as features

Ambler (2007) also suggests using such artifacts as CRCs, acceptance tests, business rule definitions, change cases, dataflow diagrams, user interfaces, use cases, prototypes, features and scenarios, use case diagrams, and user stories to model requirements. These elements can be added to the software requirements specification document along with the user stories.

For requirements elicitation he suggests using interviews (both in-person and electronic), focus groups, JAD, legacy code analysis, ethnographic observation, domain analysis, and having the customer on site at all times (Ambler 2007). For the remainder of this chapter our discussion focuses on the use of user stories to model requirements.

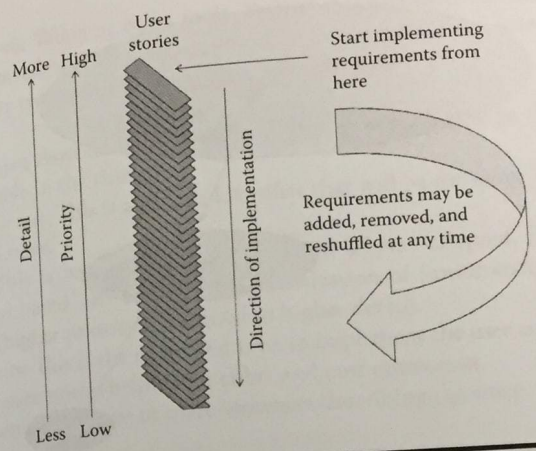

Figure 7.5 Agile requirements change management process. (Adapted from S. Ambler, *Agile Requirements Change Management*, http://www.agilemodeling.com/essays/changeManagement.htm, 2007.)

Requirements Engineering in XP

Requirements engineering in XP follows the model shown in Figure 7.5 where the stack of requirements in Ambler's model refers to user stories. And in XP, user stories are managed and implemented as code via the "planning" game.

The planning game in XP takes two forms: release and iteration planning. Release planning takes place after an initial set of user stories has been written. This set of stories is used to develop the overall project plan and plan for iterations. The set is also used to decide the approximate schedule for each user story and overall project.

Iteration planning is a period of time in which a set of user stories and fixes to failed tests from previous iterations are implemented. Each iteration is one to three weeks in duration. Tracking the rate of implementation of user stories from previous iterations helps to refine the development schedule.

Because requirements are constantly evolving during these processes, XP creator Kent Beck says that "In XP, requirements are a dialog, not a document" (Beck et al. 2001), although it is typical to convert the stack of user stories into a software requirements specification. This conversion can take place as user stories are built out or at the end when all user stories have been implemented.

Requirements Engineering in Scrum

In Scrum, the requirements stack shown in the model of Figure 7.4 is, as in XP, the evolving backlog of user stories. And as in XP, these requirements are frozen at each iteration for development stability. In Scrum, each iteration takes about a month.

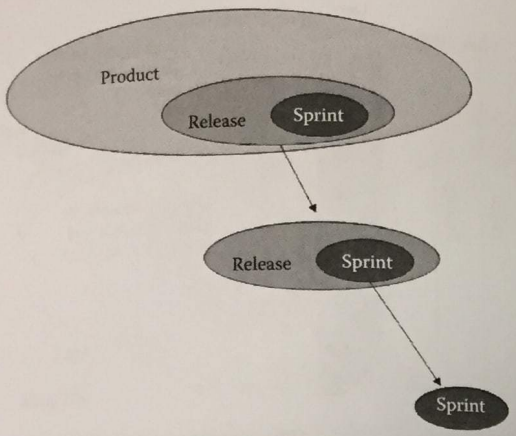

Figure 7.6 Backlog relationship among product, releases, and sprints.

To manage the changes in the stack, one person is given final authority for requirement prioritization (usually the product sponsor).

In Scrum the requirements backlog is organized into three types: product, release, and sprint. The product backlog contains the release backlogs, and each release contains the sprint backlogs. Figure 7.6 is a Venn diagram showing the containment relationship of the backlog items.

The product backlog acts as a repository for requirements targeted for release at some point. The requirements in the product backlog include low-, medium-, and high-level requirements.

The release backlog is a prioritized set of items drawn from the product backlog. The requirements in the release backlog may evolve so that they contain more details and low-level estimates.

Finally, the sprint backlog list is a set of release requirements that the team will complete (fully coded, tested, and documented) at the end of the sprint. These requirements have evolved to a very high level of detail, and hence, their priority is high.

Scrum has been adopted in several major corporations, with notable success. Some of the author's students also use Scrum in courses. In these cases it proves highly effective when there is little time for long requirements discovery processes.

Writing User Stories

User stories are the most basic unit of requirements in most agile methodologies. Each user story represents a feature desired by the customer. User stories (a term coined by Kent Beck) are written by the customer on index cards, although the process can

be automated via Wikis or other tools. Formal requirements, use cases, and other artifacts are derived from the user stories by the software engineering team as needed.

A user story consists of the following components:

Title: This is a short handle for the story. A present tense verb in the active voice is desirable in the title.
Acceptance test: This is a unique identifier that will be the name of a method to test the story.
Priority: This is based on the prioritization scheme adopted. Priority can be assigned based on "traditional" prioritization of importance or on level of detail (higher priority is assigned to higher detail).
Story points: This is the estimated time to implement the user story. This aspect makes user stories helpful for effort and cost estimation.
Description: This is one to three sentences describing the story.

A sample layout for these elements on an index card is shown in Table 7.1. Initial user stories are usually gathered in small offsite meetings. Stories can be generated either through goal-oriented approaches (e.g., "Let's discuss how a customer makes a purchase") or through interactive (stream-of-consciousness) approaches. Developing user stories is an "iterative and interactive" process. The development team also manages the size of stories for uniformity (e.g., too large—split, too small—combine).

An example user story for a customer returning items in the pet store POS system is shown in Table 7.2. User stories should be understandable to the customers and each story should add value.

Table 7.1 User Story Layout

Title		
Acceptance Test	*Priority*	*Story Points*
Description		

Table 7.2 User Story: Pet Store POS System

Title: Customer Returns Items		
Acceptance Test: custRetItem	*Priority: 1*	*Story Points: 2*
When a customer returns an item, the purchase should be authenticated. If the purchase was authentic then the customer's account should be credited or the purchase amount returned. The inventory should be updated accordingly.		

Table 7.3 User Story: Baggage Handling

Title: Detect Security Threat		
Acceptance Test: detSecThrt	*Priority: 1*	*Story Points: 3*
When a scanned bag has been determined to contain an instance of a banned item, the bag shall be diverted to the security checkpoint conveyor. The security manager shall be sent an e-mail stating that a potential threat has been detected.		

Developers do not write user stories; users do. But stories need to be small enough that several can be completed per iteration. Stories should be independent (as much as possible); that is, a story should not refer back and forth to other stories. Finally, stories must be testable. As with any requirement, if it cannot be tested, it's not a requirement! Testability of each story is considered by the development team.

Table 7.3 depicts another example user story describing a security threat detection in the airport baggage handling system.

Finally, it is worth noting that there is a significant difference between use cases and user stories. User stories come from the customer perspective and are simple and avoid implementation details. Use cases are more complex, and may include implementation details (e.g., fabricated objects). Customers don't usually write use cases (and if they do, beware, because now the customer is engaging in "software engineering"). Finally, it's hard to say what the equivalence is for the number of use cases per user story. One user story could equal one or more than 20 use cases. For example, for the customer return user story in Table 7.2, you can imagine that it will take many more use cases to deal with the various situations that can arise in a customer return. In agile methodologies user stories are much preferred to use cases.

Agile Requirements Engineering

We need to make a distinction between requirements engineering for agile methodologies and "agile requirements engineering." Agile requirements engineering means, generally, any ad hoc requirements engineering approach purported to be more flexible than "traditional" requirements engineering. This definition is not to be confused with specific practices for requirements engineering in agile methodologies as we just discussed (Sillitti and Succi 2006).

A number of agile requirements engineering approaches have been introduced in the past few years, many of them not much more than sloppy requirements engineering. But some of the recent work in this area has been good too. However, for any "legitimate" agile requirements engineering methodologies that you may

encounter, most of the practices can be traced to the Agile Manifesto. For example, Vlaanderen et al. (2011) showed how certain requirements engineering practices from Scrum could also be for software product management (after delivery). In particular they identified sprint cycles, backlog administration, daily meetings, and early and frequent collaboration as necessary practices.

Story-Test-Driven Development

To illustrate an "agile methodology," we describe one notable example. In this methodology called "story-test-driven development" (SDD), many of the usual customer-facing features of agile methodologies are incorporated along with short bursts of iterative development à la XP or Scrum. One difference in SDD, however, is that instead of the conventional user stories, customers write or review "story-tests." Story-tests are nontechnical descriptions of behavior along with tests that verify that the behavior is correct. The story-tests help "discover a lot of missing pieces or inconsistencies in a story" (Mugridge 2008).

A nice feature of the story-test is that it uses the Fit framework to allow users to build test cases into the stories in a tabular fashion (see Chapter 8). Hence, the users intuitively specify behavior and the tests for that behavior in the requirements.

For example, for a typical payroll system for a business (such as the pet store), a story-test describing how to calculate ordinary and overtime pay for an employee based on various clocked hours is given in Figure 7.7.

Within

WageProcessing

Calculate

Hours	Max ordinary	Ordinary	Overtime
9,9,9,9,9	9	45	0
9,7	9	16	0
9,0	9	9	0
0	9	0	0
11,7	9	16	2
9,9,9,9,9	8	40	5
10,11,12,13,14	9	45	15

Figure 7.7 Using story-test to show how a company calculates pay for ordinary and overtime hours. (Adapted from R. Mugridge, *Software*, 25(1): 68–75, 2008.)

The table in Figure 7.7 shows, for instance, that if an employee works five consecutive nine-hour workdays, then she is considered to have 45 hours in ordinary and zero hours in overtime pay due. If a worker clocks 10-, 11-, 12-, 13-, and 14-hour days in a week, then she is entitled to 45 hours of regular and 15 hours of overtime pay. But the Fit framework is an interactive specification: plugging in different values into the table that are incorrect will show up as invalid (see Chapter 8).

Story-test-driven development is considered to be complementary to test-driven development (TDD) in that the former is applied to overall system development (Mugridge 2008).

Challenges for Requirements Engineering in Agile Methodologies

Of course, there are challenges in any new technology, and there are some in using agile methodologies, particularly with respect to requirements engineering. Williams (2004) discusses some of these shortcomings.

For example, agile methodologies do not always deal well with nonfunctional requirements. Why should this be so? One reason is that they are not always apparent when dealing with requirements functionality only (through user stories). Williams suggests dealing with this challenge by augmenting the user stories with appropriate nonfunctional requirements written in standard form.

Another shortcoming is that with agile methodologies customer interaction is strong, but mostly through prototyping. As we have seen, there are other ways to elicit nuanced requirements and understand stakeholder needs; for example, using various interviewing techniques would be desirable.

Furthermore, with agile methodologies, validation ("Is the system the right one?") is strong through testing and prototyping, but verification ("Is the system correct?") is not so strong. Williams (2004) suggests that using formal methods could strengthen agile requirements engineering.

Rubin and Rubin (2011) identified documentation constructs that are often missed in agile software development and they suggested ways to alleviate this problem. In particular, they noted that domain knowledge and the emergent system architecture and design are not properly captured by traditional agile requirements practices. Their findings imply that agile requirements documents should be annotated with appropriate domain knowledge, and the corresponding agile architecture and design documents be annotated with the final system architecture and design, respectively.

Finally, requirements management is built into the process (e.g., XP and Scrum), but it is mostly focused on the code level. Williams suggests that requirements management could be strengthened by adding more "standard" configuration management practices.

Exercises

7.1 How do you fit SRS documentation into an agile framework?

7.2 Is it possible to use agile methodologies when the customer is not on site? If so, how?

7.3 Why are agile methodologies generally not suitable for hardware-based projects as opposed to software projects?

7.4 Why can it be difficult for agile methodologies to cover nonfunctional requirements?

7.5 Are there any problems in encapsulating requirements into user stories?

7.6 For the pet store POS system, generate a user story for customer purchases.

7.7 For the pet store POS system, generate use cases for various customer purchases.

7.8 For the airport baggage handling system, generate a user story for dealing with baggage that is to be diverted to another flight.

7.9 For the airport baggage handling system, generate use cases for dealing with baggage that is to be diverted to another flight.

*7.10 There are a number of software tools, both commercial and open source, that can be used to manage the requirements activities of an agile project. Research these tools and prepare a product comparison matrix.

References

Ambler, S. (2007). *Agile Requirements Change Management*, http://www.agilemodeling.com/essays/changeManagement.htm, last accessed March 2013.

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Chicago: Longman Higher Education.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Agile Manifesto and Principles Behind the Agile Manifesto. Online at http://agilemanifesto.org/, last accessed March 2013.

Boehm, B. and Turner, R. (2003). *Balancing Agility and Discipline: A Guide to the Perplexed*, Reading, MA: Addison-Wesley.

Cao, L. and Ramesh, B. (2008). Agile requirements engineering practices: An empirical study, *Software*, 25(1): 60–67.

Eberlein, A. and Sampaio do Prado Leite, J.C. (2002). Agile requirements. In *International Workshop on Time-Constrained Requirements Engineering* (TCRE'02), Essen, Germany, September.

Laplante, P.A. (2006). *What Every Engineer Needs to Know about Software Engineering*. Boca Raton, FL: CRC/Taylor & Francis.

* This exercise is suitable for a small research assignment.

Marinelli, V. (2008). *An Analysis of Current Trends in Requirements Engineering Practice*, Master of Software Engineering Professional Paper, Penn State University, Great Valley School of Graduate Professional Studies, Malvern, PA.

Mugridge, R. (2008). Managing agile project requirements with story-test-driven development, *Software*, 25(1): 68–75.

Rubin, E. and Rubin, H. (2011). Supporting agile software development through active documentation, *Requirements Engineering*, 16(2): 117–132.

Schwaber, K. and Beedle, M. (2001). *Agile Software Development with SCRUM*. Upper Saddle River, NJ: Prentice Hall.

Sillitti, A. and Succi. G. (2006). Requirements engineering for agile methods. In A. Aurum and C. Wohlin (Eds.), *Engineering and Managing Software Requirements*. New York: Springer, pp. 309–326.

Vlaanderen, K., Jansen, S., Brinkkemper, S., and Jaspers, E. (2011). The agile requirements refinery: Applying SCRUM principles to software product management. *Information and Software Technology*, 53(1): 58–70.

West, D., Grant, T., Gerush, M., and D'Silva, D. (2010). Agile development: Mainstream adoption has changed agility. Forrester Research.

Williams, L. (2004). *Agile Requirements Elicitation*. Online at http://agile.csc.ncsu.edu/SEMaterials/AgileRE.pdf, last accessed March 2013. Unpublished manuscript.