If the Shoe Doesn't Fit...

AGILE REQUIREMENTS FOR STEPSISTER PROJECTS

By Jennitta Andrea

To the uninitiated, Agile software methods make radical recommendations for gathering and documenting functional requirements. For example, in Extreme Programming (XP) we replace up-front requirements analysis with incremental requirements definition, replace document-based communication with face-to-face communication between developers and subject matter experts (SMEs), and record requirements as automated functional tests.

This is not a fairy tale: The resulting requirements process is both efficient and effective. Communication is direct, clear, unambiguous, and verifiable. The "catch" is that this approach has the best chance of succeeding when the project meets all of the following critical success factors:

- A team is small enough to fit within a collocated space—ideally less than ten people but no more than twenty.
- SMEs are a permanent part of the team.
- SMEs have a clear vision for the system requirements and can effectively communicate this to developers.
- SMEs can express the requirements in the form of functional tests.
- Either the problem domain has a short learning curve or the developers have deep experience in a more complex domain.

A project that fits these criteria perfectly—like Cinderella fit into the glass slipper—is an ideal candidate for this Agile requirements process. However, not all projects are Cinderella projects. What if your team is large? What if there are language barriers? What if the SMEs have tunnel vision and don't understand the big picture? What if the physical office space cannot be reconfigured? What if the developers are new to a complex or critical domain? What should you do with these stepsister projects? It's important to have an answer because stepsister projects are far more common than Cinderella projects.

If the Shoe Doesn't Fit...

Act 1, Scene 1: A team of ten skilled developers is brought together to work on Bravo, a mission critical application for a complex business domain. This is the first time the team members have worked together, and none of them has worked in this domain or on a previous Agile project. Although a highly qualified SME is assigned to the project full time, this is her first involvement with a software development project and she lacks a clear vision of the end product.

Enter an Agile expert to conduct a readiness assessment for Bravo to determine if it is a good candidate for a particular Agile requirements process. It is readily apparent that the glass slipper does not fit. At this point the expert can lead Bravo down one of three alternative story lines.

The *naïve expert story line* continues with the expert guiding the Bravo team through the Agile requirements process

even though it is a stepsister project. Risk is added to the project because this requirements process is too light for the project factors.

The *purist expert story line* continues with the expert walking away, insisting the Bravo team is not worthy of using an Agile requirements process. The team adopts a more familiar heavyweight, up-front requirements process instead. Waste is added to the project because this requirements process is too rigid and bulky for the project factors.

The *pragmatic expert story line* continues with the expert suggesting that if a particular glass slipper does not fit, the Bravo team can mold an Agile requirements process to fit its unique project footprint. The requirements process is as efficient and effective as possible, given the circumstances.

This article explores the pragmatic story line in more depth and offers thinking tools for understanding the distinctive characteristics of a project and strategies for shaping an Agile requirements process that is a snug fit. Our target is to minimize the effort in specifying requirements without introducing risk or waste.

A Brief Deconstruction

Today's systems are difficult to build and difficult to comprehend. We need a requirements approach that views the problem from more than one perspective. Functional requirements for a system are specified by a set of interrelated textual and graphical notations that describe the essentials: behavior, domain vocabulary, state transitions, and workflow.

A functional requirements process not only prescribes which techniques and notations to use but also includes refinements, such as whether to create a physical artifact, and if so, how much detail is necessary, how formal should it be, the lifetime of the artifact, who needs to be involved, and how the work is chunked. To ensure that all of the separate pieces are balanced and consistent with fundamental guiding principles, each isolated decision must always be made with an eye to the whole. A pragmatic requirements process ensures that each choice is appropriate for a project's particular characteristics: team size, physical proximity, domain experience, domain complexity, and system criticality.

Artifacts

There are a staggering number of requirements artifacts to choose from, including user stories, functional tests, use cases, activity diagrams, sequence diagrams, state transition diagrams, class diagrams, storyboards, prototypes, etc. Many of these artifacts have both a graphical and a textual format. The artifacts cover one or more of the following system perspectives: vocabulary, behavior, state transition, and workflow.

To help decide which artifacts are necessary, consider these project factors:

- *System type*: Include only the perspectives that are appropriate for the type of system being built. For example, a highly integrated business critical application likely will need a workflow perspective, while a simple data entry application will not.
- *Domain complexity*: As the complexity of the problem increases, it becomes more difficult to describe from a single perspective. Segmenting the details across many diverse perspectives is one strategy for managing complexity.
- *Target audience*: A variety of players have an interest in the system (e.g., customers, users, developers, testers, auditors, and marketers). Survey each role to determine what information each needs and what format best conveys the information.

Detail

We can specify an artifact at various levels of detail. By giving an artifact an intent-revealing name, we can communicate some basic information about the requirement. An outline of the requirement gives the reader a general overview (e.g., a sketch of a functional test without specific data values or a paragraph describing a user goal and the outcomes of a use case). Full detail removes ambiguity by including all of the essential information (e.g., the preconditions, action, and validation sections of a functional test; or a use case with the steps for the main scenario and all alternative scenarios).

To help decide the necessary level of detail, consider these project factors:

- *Domain knowledge*: As the team's knowledge and experience with a domain increases, less detail is required.
- *System criticality*: As the impact of the system on human safety and/or business stability increases, more detail is needed to reduce risk associated with ambiguity.
- *Domain complexity*: As the complexity of the problem increases, more detail is required to accurately describe it.
- *Time*: As the project progresses through stages (e.g., scoping, prioritizing, estimating, developing, accepting, and delivering), the detail of the artifacts may vary based on need.

Formality

The notation used to express a concept, the tool used to record the concept, and the mechanism for delivering the message (channel) are indicators of the formality of functional requirements. We can mix and match the formality of each of these components. For example, consider a requirements specification in which the SME sketches a UML diagram on a whiteboard during a face-to-face meeting with several developers. This blends together a formal notation, an informal tool, and an informal communication channel. By increasing or decreasing the formality of each of these components independently, many other effective combinations are possible.

To help decide how formal the notation, tool, and channel should be, consider these project factors:

- *System criticality*: As the safety or business criticality of a system increases, so too does the need to be auditable. The formality of the notation will likely increase to a formal industry standard.
- *Team size and/or proximity*: As the team size and/or distance increases, it becomes more difficult to use face-to-face communication effectively. To ensure the message is delivered consistently to everyone on the team, the channel likely will have to increase in formality (e.g., semi-formal email messages or formal documents, diagrams, or presentations).

- *Team experience*: Regardless of whether a notation is informal/ad-hoc or formal/industry standard, the effort to understand it is based upon how well the notation maps to the domain and how much of a learning curve is required to understand it.

Lifecycle

In life we are taught that the "early bird gets the worm," so getting an early start on requirements seems like the responsible thing to do. However, if requirements are uncertain or are likely to change, specifying them too early may cause rework. On the other hand, starting work on an artifact too late may create a bottleneck for the waiting consumer.

Just as we must be concerned with how late in the project we should wait before creating an artifact, we must be concerned with how early we should retire it. Requirements artifacts often are considered untrustworthy because they get out of synch with the real system so quickly. We must make conscious decisions about how long handcrafted artifacts should be maintained. Once an artifact has served its purpose, it is a candidate for retirement, either by deleting it or by marking it as depreciated. Carefully consider how permanent artifacts will be kept in synch with the actual system. Ideally they are either reverse engineered or are an integral part of the development process (e.g., automated functional tests).

To help decide the lifecycle of an artifact, consider these project factors:

- *Requirement priority*: Prioritize the requirements for a system based upon value, risk, and opportunity for insight. Start the highest priority items early.
- *Target audience*: Adjust the start date of an artifact so that it is ready just in time for the consumer.
- *Requirement stability*: Delay work on unstable requirements for as long as is practical.
- *Artifact purpose*: Once you understand who needs each artifact and why he needs it, decide when to retire it.

Guiding Principles

While defining or refining a requirements process is not for the faint of heart, it is possible—with knowledge, experience, and perhaps the help of a Fairy Godmother. The obvious first step in refining an Agile requirements process is to review your project factors and, where feasible, adjust the factors until they are as close to the ideal as possible.

When defining the elements of an Agile requirements process, the target is to minimize the effort spent specifying requirements without introducing risk or waste. For each requirement we want to minimize the number of artifacts produced so we are not saying the same thing multiple times, minimize the detail contained in the artifacts to avoid telling someone what they already know, minimize the formality of the artifacts so we are not doing more work than necessary, develop the artifact as late as possible to avoid rework or unused work, and retire the artifacts as early as possible so that we minimize the maintenance effort.

Visualizing a Process

With so many interdependent choices, it is hard to know where to begin when refining a requirements process. One approach is to layer the combined choices together in a visual thinking tool called an Artifact Density Graph (ADG). Each artifact is represented in its own row. Each decision about an artifact has a different effect on the appearance of the row. Detail is recorded through the height of the bar—the shorter the bar, the less detail the artifact contains. Formality is recorded through the color of the bar—the lighter the color, the less formal the artifact. Lifecycle is recorded through the width of the bar—the right and left edges can be adjusted to shorten the artifact's lifetime.

Once the ADG has been completely filled in, we can see the silhouette of the process. The general qualities of the requirements process become apparent. Overall effort is indicated by the amount of color in the graph. Artifact duplication is detected by multiple rows covering the same perspective during the same time period.

The ADG of a Cinderella XP requirements process is shown in Figure 1. This process is based on two key artifacts: user stories and functional tests. They serve distinctly different purposes and have distinctly different silhouettes in the graph.

Figure 1: Cinderella XP Artifact Density Graph

On a Cinderella project, a system is defined in terms of small pieces of useful functionality, called user stories. The prioritization and iterative development of user stories is key to a project's being able to adapt quickly to change. Because user stories are not very detailed, containing just enough information to facilitate credible estimates, the bar is not very high. User stories are informal (light color of the bar), typically handwritten text on index cards. Their lifecycles are very short (narrow width of the bar) because their primary purpose is to contain just enough information to plan and guide an iteration. The full details of the requirements are elaborated during the iteration through face-to-face conversations and ultimately take the form of functional tests. When these tests are automated, they are fully detailed (full height of the bar) and formal (dark color). The functional tests are a permanent requirements artifact (full width of the bar).

Visualizing Project Factors

As we have seen, project factors play a significant role in a requirements process. There are many different, interdependent, and contradictory project factors. It often is difficult to know which ones are dominant. One approach to seeing the big picture is to represent the project factors in a visual thinking tool called a radar diagram

(See Figure 2.) Each project factor is represented by its own axis and scale.

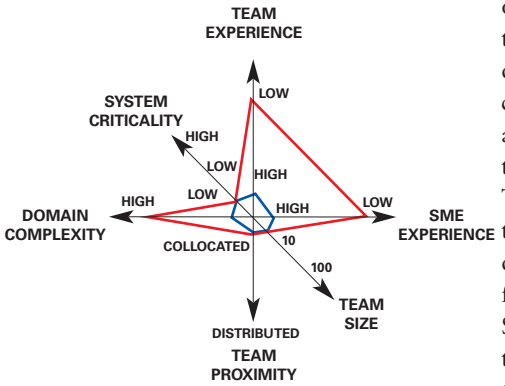

Figure 2: Radar diagram for XP and Bravo

The scales have been arranged such that the ideal factors for an XP requirements process form a tight web in the center. The blue line maps the project factors for a Cinderella XP requirements process: small, collocated team, a qualified SME, and either a simple domain or a team that is experienced in the domain. The red line maps the project factors for our fictional project, Bravo. Fortunately, Bravo's team size, team proximity, and system criticality correspond to the XP ideal. Bravo differs from the XP ideal in domain complexity, team domain experience, and the SME's inexperience at specifying software requirements.

Bravo

Our goal is to develop an Agile requirements process for Bravo, a stepsister project. Because Bravo's team size, proximity, and system criticality factors map to XP's ideal, the project can base its requirements process on this prioritized, story-driven, iterative approach. However, performing just-in-time requirements elaboration within a short iteration will present challenges. The combination of domain complexity and domain inexperience often brings productivity to a halt because the SME and developers spend more time on analysis than on development during the iteration. A common adaptation is to allocate time in a previous iteration for the SME to work with a coach to specify the complex stories for the current iteration to an outline level of detail (one level higher than is typically done for XP). This provides the team with an overview of the domain concepts before the planning of the current iteration, making the team's estimates more credible. Additional detailed and semiformal artifacts such as class diagrams and activity diagrams are added to the process to compensate for the team's lack of domain knowledge. These artifacts are semipermanent since they can be retired when the team's domain experience improves. Using low-fidelity storyboards early on helps the SME visualize the purpose and flow of the application, which helps to drive out the detailed requirements.

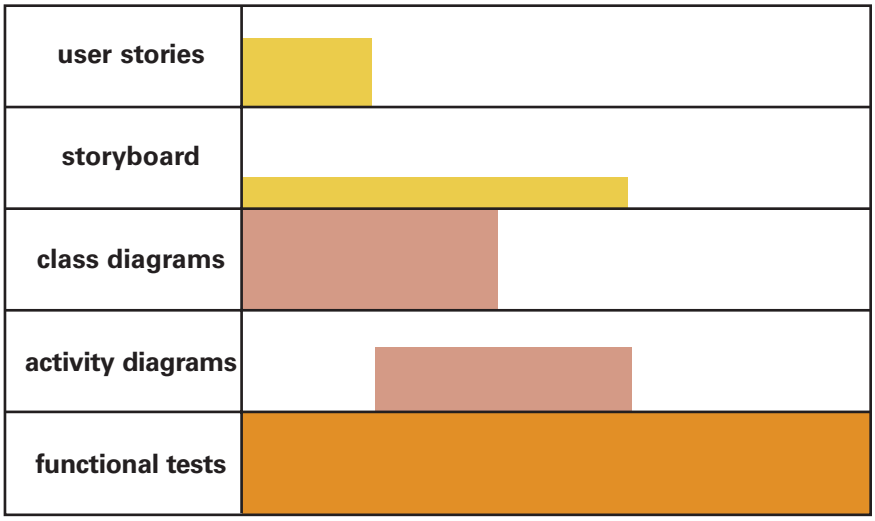

Figure 3:
Project Bravo Artifact Density Graph

Figure 3 illustrates the ADG for Bravo. But remember, Bravo is a fictional project that is used to illustrate the application of the concepts. Expect a different ADG for projects with different factors.

Happily Ever After

Stepsister projects can enjoy the benefits of an Agile requirements process that minimizes effort, increases responsiveness to change, reduces waste, and mitigates risk. Mastery of process adaptation is developed over time as the team gains experience with the interdependent and often contradictory relationships between the project's factors and the elements of a requirements process. When the team builds its requirements process on a solid foundation of Agile principles, it will be clearer what the tolerances are for decisions and how to judge the results. The safest way to refine a process is to have considerable hands-on experience with that process before you start.

A final word of caution: If the team is not watchful, the clock can strike midnight on the requirements process and things can begin to unravel. Project factors are likely to change over their lifetime: people gain more domain experience, new people join the project, the team gets relocated to a different space, new features are added, etc. Changing just one thing about a project can make a big difference in how the requirements process fits a project. Regularly assess the requirements process and project factors and make adjustments when symptoms like duplication, waiting, requirements bugs, and ambiguity are detected. If this all seems a bit daunting to tackle on your own, a Fairy Godmother (a.k.a. process coach) can make a big difference—after all, Cinderella wouldn't have gotten anywhere without one!

Acknowledgements

This article builds upon a tutorial that was co-developed with Gerard Meszaros. **{end}**

Jennitta Andrea is a coach, developer, tester, analyst, and instructor with clearStream. She worked on her first Cinderella XP project in 2000 and has worked on more than a dozen (mostly stepsister) Agile projects since then. Jennitta is regularly published, speaks at conferences, and is a board member of the Agile Alliance.