Agile Product & Project Management
Executive Update Vol. 13, No. 5

From Agile Requirements to Agile Code

by Steve Berczuk

Being agile means managing a project in an iterative fashion with incremental delivery. Iterative and incremental delivery requires a change in your approach to planning, to execution, and to basic cultural assumptions about software delivery. Changes to planning and execution are necessary for an agile approach to work. This *Executive Update* discusses how planning and execution interact to change assumptions about software delivery and also points out some approaches in engineering practices that will increase the likelihood of agile success.

AGILE BASICS

The agile software development model is simple in concept but challenging to execute. Agile methods acknowledge uncertainty and manage that uncertainty using working software as the measure of progress. By using techniques that rely on transparency, inspection, and adaptation, teams can deliver business value while also adjusting goals in response to changes in current needs or future technical or market forces.[1] Some agile methods, such as XP, focus on technical practices, while others, such as Scrum, focus on project management. To iteratively plan, deliver, review, and then adjust plans based on experience requires changes on both the planning and technical sides, and change is always difficult.

Good planning makes it easier to execute in an agile way. But agile plans can only be effective if the team follows engineering practices that provide the feedback needed to inspect and evaluate as well as those that produce a code base that allows you to adapt. If you are open to an iterative approach to agile adoption, you can make small steps that will help product management and development teams work together better.

AGILE PLANNING

Before diving more deeply into the interactions between planning and execution, it's important to clarify that agile planning means "enough" planning to move forward and measure progress.[2] You need to strike a balance between (1) vague and imprecise requirements that are difficult to implement and track and (2) elaborate use-case analyses for features that don't need them. By defining things with just enough detail, you can get feedback based on seeing software in action.

Agile requirements are focused on delivering functionality to users and start with three things: a user (who has a business need), a feature (what the system will do), and a goal (the reason the user wants to do the task).[3] While many teams can develop lists of features, the user and end goal parts of the story are often missing, but they are the most difficult and most important parts to express. Having a clear statement of product goals allows you to decide what implementation will satisfy the core need and to evaluate whether the story is even necessary to implement. Since implementing features that do not have a clear use is wasteful, removing items from the backlog can be a major efficiency gain for a team. Try to develop user stories with incremental delivery in mind.

Iterative and Incremental Planning Challenges

To deliver increments of software functional in iteration, product owners need to think in terms of incremental features and their priority. Incremental features are features that are well defined, move the product forward, and can be developed in an iteration. Because they are small, incremental features may not be as rich as a traditional requirement, but their development provides insight into implementation risks and requirements risks, allowing product owners to make adjustments to meet a delivery goal. Defining incremental features for an agile project is difficult because it requires a great deal of precision: you need to define the goal with enough detail so that you can evaluate its completeness.

Incremental requirements require a cultural shift toward making decisions in the face of uncertainty.

Agile methods work best if the team is aiming in one direction then changes direction based on feedback, rather than working off of imprecise goals. If a requirement isn't well understood, the exercise of defining a (simple) execution will be useful. For an agile project to be successful, you need to be able to set expectations about what a team will deliver in order to see how well goals are met. Deciding if a team completed a story may always have a subjective element, but try to define what "done" means if you want to be able to measure and track progress reliably.

Without a good understanding of a "complete story," the team cannot estimate accurately and thus cannot set expectations. Not being able to set expectations can cause trust between the team and the product owner to break down. Without trust, it's harder for management to accept the idea of self-organizing teams, which loses the efficiency benefits that self-organization provides.

Prioritization is also a challenge. First, product owners are sometimes reluctant to assign priorities out of a sense that only "critical" issues will get attention. This indicates a lack of trust in the team to deliver on commitments and its ablility to change direction. Also, assigning a "1 … *n*" priority to an entire list can be lots of work, but working off a list of Priority 1 issues is in effect working off of an unprioritized list. The goal of agile is to help organizations deliver the most important business value and not waste effort delivering features that will not be used.

Agile Execution Challenges

The engineering teams need to make changes to support agile planning. Traditional teams sometimes try to manage risk by doing lots of design up front and avoid writing code that may need to be changed later. In most projects, agile or not, needs and code do change. While design and good coding practices are important on all projects, an agile developer needs to write code that can adapt to the challenges of incremental and iterative delivery.

AGILE CODE

Much like agile requirements aim toward a goal, agile engineering practices help teams write code that meets the goals of an iteration and changes direction easily. To be an agile team, you need agile code. The agility of a code base is related to the architecture, development practices, and the delivery model. As working software is the main way of evaluating progress on an agile project, agile engineering practices can drive agile planning techniques when they are lacking.

Agile code is code that you can change while still being able to deliver working software on a regular basis. What makes code agile is a combination of good design and the application of practices that provide constant feedback on the state of the code so problems can be detected as soon as they occur. These practices include:

- Automated unit and integration tests, in combination with continuous integration, to provide immediate feedback on the effects of a change to the code
- Refactoring and continually improving the structure of code while maintaining functionality
- Frequent deployments to a production-like environment to identify issues early and to make the application visible to stakeholders

By maintaining code in a working state, it is possible for the team to quickly implement changes to a product backlog that a product owner might request. Since the goal of an agile project is to deliver business value, remember that technical practices are a means to an end, and items such as refactoring, design, and testing should not appear directly on the product backlog. Rather, consider how these tasks further the progress of the project. Maintaining clean agile code does have more up-front costs than not doing so, but it's important to allow for the effort to do so, since delivering code that can sustain change is essential to agile success.

AGILE ARCHITECTURE

Working software and demonstrable features are the measures of progress on an agile project. Given a narrowly focused user story, agile teams focus on developing software in vertical slices through the architecture by feature rather than in traditional architectural layers (i.e., UI, application, database). Developing end-to-end features — rather than focusing on the data model, the UI, or application tier — has its

The *Executive Update* is a publication of Cutter's Agile Product & Project Management practice. Print ISSN: 1946-7338 (*Executive Report*, *Executive Summary*, and *Executive Update*); online/electronic ISSN: 1554-706X.

advantages. Users can see the application *do* something. A data model, while important, is not easy to demonstrate, and a UI backed only by scaffolding does not help identify implementation challenges early. With an end-to-end approach, the team and product owners can understand what features are truly necessary and gain a better sense of what to defer if something is late.

Early feedback on architecture is especially valuable for managing risk. The team can validate the interactions between layers and make changes to simplify work at other layers easier, thus minimizing unnecessary rework. UI implementation can be influenced by decisions made at the data layer and vice-versa.

Developing in vertical slices does not eliminate the need for architecture and design, but it does call for a more lightweight architecture that you can evaluate as you go. Authors Jim Coplien and Gertrud Bjørnvig say much more on this subject in their book *Lean Architecture: for Agile Software Development*.[4]

The Agile Team

To be able to implement in vertical slices and be efficient, agile teams are often composed of generalizing specialists. Generalizing specialists can work on multiple aspects of the system, though they have expertise in a particular area.[5] This means that all work that touches the UI is not blocked if your UI developer is overly busy. It also allows a first-pass, end-to-end implementation by a single developer. As a generalizing specialist, you are not abandoning the idea that there are no "experts"; you are encouraging team members to learn about and work with other aspects of the code. Having such a cross-functional team not only reduces bottlenecks in the development process, but also improves code quality by increasing the number of people who work with, and thus implicitly review, code.

Delivery and Deployment

Working software is the measure of progress in an agile project. But working means more than just "can demo" or "compiles and passes automated tests." Software isn't useful, and stakeholders cannot provide useful feedback on it until it can run on the target environment. Make your application available on a target system early, and verify the deployment and installation process often. This gives you an opportunity to identify decisions, which will simplify the deployment and configuration process early, as there are differences between a development system and a production-like one that you need to address to be "done."

CONCLUSION

To be successful at agile, consider the entire product lifecycle, from planning to execution, and be aware of the challenges that the difference in approaches will present to teams coming from a different background. Agile engineering practices may encounter less resistance than the planning practices, and, as long as your organization wants to be more agile, working on the technical practices can help identify other bottlenecks to agile.

ENDNOTES

[1]"The Scrum Guide: The Official Rulebook." Scrum.org, 2011 (http://www.scrum.org/scrumguides).

[2]Berczuck, Steve. "Starting Agile Adoption: Part II — Avoiding Common Pitfalls of Planning." Cutter Consortium Agile Product & Project Management *Executive Update*, Vol. 11, No. 21, 2010.

[3]Cohn, Mike. *User Stories Applied: for Agile Software Development*. Addison-Wesley Professional, 2004.

[4]Coplien, James O., and Gertrud Bjørnvig. *Lean Architecture: for Agile Software Development*. Wiley, 2010.

[5]Berczuck, Steve. "Generalists, Specialists, and Generalizing Specialists." Cutter Consortium Agile Product & Project Management *Executive Update*, Vol. 12, No. 16, 2011.

ABOUT THE AUTHOR

Steve Berczuk is an engineer and ScrumMaster at Humedica, where he's helping to build next-generation clinical informatics applications based on SaaS. The author of *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*, he is a recognized expert in software configuration management and agile software development. Mr. Berczuk is passionate about helping teams work effectively to produce quality software. He has a master's degree in operations research from Stanford University, a bachelor's degree in electrical engineering from MIT, and is a Certified Practicing ScrumMaster. He can be reached at steve@berczuk.com.