Agile Software Development for the Entire Project

Granville Miller
Microsoft

Does an agile software development process require real organizational change or can an existing organization become more agile? How do the many traditional information technology (IT) roles such as the business analyst, architect, and tester become a more integrated part of an agile process? Some recent work [1] debunks the myths that agile processes require on-site customers, produce ad-hoc requirements and design, and cannot scale to large projects. This article furthers this work by introducing innovative techniques from a new agile process developed and used by projects within Microsoft. These techniques span the traditional IT roles such as the business analyst, project manager, architect, developer, tester, and release manager.

Many of today's more popular agile software development processes concentrate strictly on the developer and project manager. Traditional information technology (IT) roles such as business analysts, architects, and testers do not play a part in many of these agile processes. Yet, most software product and IT organizations have these roles or their equivalent. What is more, they are not ready to give up on them. On the contrary, these roles are becoming more valuable rather than less so as distributed development becomes more prevalent.

There are other practices such as the on-site customer, universal code ownership, pair programming, and stand up meetings that have proven barriers to widespread adoption of the more popular agile processes in many organizations. We have heard that it is mythical that these practices are required to be agile [1]. However, we have not been offered alternatives in a process form. This article introduces Microsoft Solutions Framework (MSF) for Agile Software Development, a context-based, agile software development process for building .NET applications [2]. This new process provides innovative techniques to extend agile software development to all of the traditional IT roles.

MSF for Agile Software Development is composed of a set of proven practices commonly used to build software at Microsoft. These practices have been collected in an agile form and used by teams both inside and outside of Microsoft. This process provides a set of practices that complement each other; that is, the sum of the practices is greater than each one used in isolation [3]. It also presents alternative practices to those commonly found in many agile processes.

The Agile Pattern

The core of any agile software development process is the way that it partitions and plans the work. Most agile processes share a similar method of planning or the *planning game* [4]. To start, a project is divided into time boxes or fixed periods in which *development* is done. These time boxes are called iterations. The iteration length is usually fixed between two to eight weeks, although really small projects have been known to set the iteration length in days or even hours.

> *"The personas describe usage patterns, knowledge, goals, motives, and concerns of a group of users. The key to good personas is that they are memorable and represent a set of typical customers."*

In each time box, we schedule work from two lists, our version of the product backlog [5]. The first is the scenario list that contains the names of scenarios (or scenario entries) that serve as placeholders for necessary functionality. The second is the quality-of-service requirement list that contains a list of requirements in areas such as performance, platform, or security. The scenarios and quality-of-service requirements in these lists are prioritized, and rough order-of-magnitude estimates are initially provided by the developers.

Scenarios and quality-of-service requirements are selected for the upcoming iteration and placed in the iteration plan (the equivalent of the sprint backlog [5]). The amount of work that is chosen is based upon the previous iteration's velocity. Once selected for iteration, more detailed scenario information is written by the business analyst. After the detailed information is provided, developers divide the scenarios into tasks and provide more detailed costs for the tasks. The costs are checked to make sure no developer is overloaded.

All of this planning occurs in a staggered manner. For example, our business analyst and project manager are working on planning iteration 1 in iteration 0. Developers spend a negligible portion of their time dividing the scenarios (and quality-of-service requirements) into tasks and choosing their tasks for the next iteration. However, most of their time is spent completing their tasks for the current iteration.

Development tasks are just one form of work breakdown that occurs. Testers and architects also create tasks as part of the iteration plan. These roles are integrally involved in ensuring that the solution is well architected and tested. They work in conjunction with the developers, business analysts, and project managers to ensure the system holds together. We will explore the nature of the architect and test roles later in this article.

Customer Collaboration Over Contract Negotiation[1]

There is no denying that the on-site customer, a customer that can work directly with the team to explain what is required of the system, is probably the best way to ensure project success. Unfortunately, most users have jobs other than guiding the delivery of a new system. It is rather ironic that the very thing that leads to a successful project is such a rare occurrence.

At Microsoft, lack of time is only one reason our users may not be able to be on-site. They may be located in a different city or even a different country. They may not be a part of our company at all in the case of commercial products. In any of these circumstances, our ability to interact with these users may be limited. When we obtain the opportunity to interact, we need to make the most of it. We also need to be able to communicate the essence of these interactions to the rest of the team.

Of course, this is exactly what the business analyst[2] is supposed to do in most organizations. In cases where travel is necessary to interact with users, they go. After all, nothing interesting happens in the office. We send these folks to meet with our customers because sending developers on frequent trips has an adverse affect on the project's velocity. However, customer knowledge should not be locked in a few people's heads. Instead, it should be shared with the entire team.

Sharing details of a customer visit is commonly performed in most organizations with trip reports. However, trip reports are an inadequate vehicle for providing anything more than a cursory insight into the customers. Instead, Microsoft utilizes a technique called *personas* as a basis for bringing the spirit of the customer to the entire development team [7]. Personas are respectful, fictitious people that represent groups of users. The personas describe usage patterns, knowledge, goals, motives, and concerns of a group of users. The key to good personas is that they are memorable and represent a set of typical customers.

Personas can also be compared to actors in use-case models [8]. An actor is an entity that interacts with the system. Human actors are instances of a role. The actor often contains very little information other than this role name. Therefore, while an on-site customer can usually provide us with better insight, an actor provides fewer details about the user community than a persona. In fact, actors make the assumption that all of the people that play a given role interact with the system in the same way.

Personas allow all of the members of the development team to obtain a deeper understanding of the user community. Design, development, and test decisions can often be made purely on the basis of a good persona. This allows the team to maintain velocity even when the business analyst is *on the road*. Personas must be constantly refined as new information is learned through interactions with the users. Posters of the personas can be found on the walls in the halls of the Microsoft campus.

In addition to writing the personas, the business analyst also generates the scenario entries in the scenario list. Once a scenario is scheduled for iteration, the business analyst writes up the details of that scenario. Personas are used in these scenario descriptions to show how a user would interact with the system. This provides the development team with an even deeper insight into the user community through understanding how the personas interact with the system.

Finally, there is no substitute for reviews of system functionality after key iterations with the customer. There are many vehicles for these reviews from actual working systems to storyboards with screen captures in cases where it is impossible to simulate the deployment environment in the area where the review is held. Experience at Microsoft has shown that using personas in conjunction with scenarios leads to fewer changes resulting from these reviews than when personas were not used. Ultimately, a certain amount of change occurs when reviewing newly built systems even when an on-site customer is present.

"As larger, agile projects require teams of teams, communication between the teams becomes especially important. Representing the needs of the solution as a whole is the architects' responsibility."

Working Software Over Documentation[3]

The goal of each iteration is to produce working software. The agile community believes that those activities that do not contribute to this working software are considered lower priority, if not detracting. Unfortunately, there is also a general belief that many of the traditional architectural activities fall into this category.

To be clear, the agile philosophy does not hold a belief that architecture is unbeneficial. Instead, it is a reaction to some of the large design efforts that were performed at the beginnings of projects and later found to be flawed. This form of design is known as *big design up front* (BDUF). The objection that the agile community has to BDUF is that without working software, these efforts have no feedback mechanism. Therefore, quite a bit of time can go into these efforts without an understanding of whether they are constructive or not. Many projects found that their implementation technology did not support these designs and a considerable amount of time had been wasted.

Architecture is an important part of any project, agile or otherwise. It is especially important in the larger agile projects [9]. However, architecture must lead as well as reflect the structure and logic of the working code. Disconnected architectural efforts are often greeted with skepticism by the developers who are building the pieces of the system. However, understanding every detail of a system, especially a larger one is beyond most people's capability. Architects have their hands full just keeping abreast of the changes for iteration. Therefore, keeping the architecture synchronized should be as simple as a whiteboard drawing and as equally expressive [10].

Architects must therefore take a broad view of the system in addition to understanding a certain depth. This breadth is important on larger, more complex projects. When a project spans multiple teams, it is important to communicate responsibility and overall system structure. As larger, agile projects require *teams of teams*, communication between the teams becomes especially important. Representing the needs of the solution as a whole is the architects' responsibility.

To create an agile architecture, MSF utilizes *shadowing*. A shadow is architecture for the functionality to be completed in the iteration. The shadow leads the working code at the beginning of iteration as the architects get out in front of the development for the iteration. During this time, the architecture and the working code are not in sync. This shadow communicates any re-architecting or redesign that needs to occur to keep the code base from becoming a stove pipe, spaghetti code, or one of the many other architectural anti-patterns [11].

As the pieces of the leading shadow are implemented, the architecture begins to reflect the working code base. The original parts of the system that were

architected but not implemented now become implemented. When the architecture represents the working code, we call the shadow a trailing shadow. As the sun sets on the iteration, the leading shadow should be gone and replaced strictly by trailing shadow. The trailing shadow is an accumulation of the architectures over all the iterations.

To keep architecture from becoming too detailed, we recommend that it be focused at the component and deployment levels. For example, a smart client system for generating budget information may consist of a Windows client and a number of Web Services [12]. Each of these Web services, the underlying database server, and the client itself would be components in this model. Remaining at the component level keeps architects from becoming the police of low-level design, although it never hurts to get tips from a more experienced developer.

The Microsoft terminology for one of these deployable components such as a Web service or database server is an *application*. One of the chief tools for the MSF architect is the application diagram, the equivalent of the component diagram in the Unified Modeling Language. Since the application diagram focuses on more concrete entities such as a Windows application, ASP.NET Web service, or external database, more system-level detail can be provided.

Shadowing is applied at the component or applications level. A shadow application initially communicates a desired change in the component-level behavior of a system. Shadow applications become invaluable when multiple teams are trying to coordinate work across multiple components. Changes can be made without affecting the code base until the architecture is ready to be implemented. Next, the code is generated[4] or written for the shadow and the leading shadow is removed and replaced with a trailing shadow.

The planning process for creating shadow applications is similar to the agile pattern used to partition and plan the development work for the system. New architecture tasks are created at the beginning of the iteration when any structural changes need to be made to the architecture to accommodate the new scenarios or quality-of-service requirements. Architecture tasks are like the development or coding tasks that are used to divide the scenarios into the lower-level pieces that can be assigned to a single developer. However, they pertain to the architectural functions that must be performed to keep the system from entropy[5].

As a result of these tasks, the architect will add the endpoints or interfaces to the shadow applications to reflect the needs of the new requirements. These endpoints can be validated to ensure that the components such as Web services will work together properly in the context of the deployment environment. The endpoints of these applications can be connected to show how the components interact. Each application may be distributed on a separate machine or clustered to work together on a single machine.

As the development team becomes ready to implement the scenarios, the endpoints are deleted from the shadow applications and added to the application that represents working code. Unit tests are created for each side of the component to ensure that the proper functionality is provided and unit-tested. Finally, working code is written for these new endpoints.

At the end of the iteration, all of the proxy or unimplemented endpoints should be gone. In other words, all of the architecture should be translated into working code. The architectural model is not divorced from the working system, but rather is a reflection of it. This makes the documentation for the component model match the working system. Unit tests should be in place to make sure that the interfaces continue to work as new functionality is added.

Shadow applications provide many advantages. They keep the high-level design of the components in the system consistent with the code base. They allow larger teams to define responsibilities in the context of an agile architecture. Shadow applications are used to track the building of functionality across component boundaries. In this way, they allow MSF for Agile Software Development to scale to larger, more complex projects.

Individuals and Interactions Over Processes and Tools[6]

The idea of valuing people over processes and tools is not an indication to move away from the use of today's advanced tools. In fact, one of the roots of the agile revolution is the advance of the compiler technology provided by our software development tools. These compilers have made it easier for us to build systems incrementally. If compilation times took hours, as they did in the past, instead of seconds or minutes, can you imagine performing one refactoring at a time? Can you imagine running a unit test first to see it purposely fail after waiting two hours for it to compile?

As our development tools have advanced, so has our capability to take advantage of these advances in our development processes. However, developing software is ultimately an activity for knowledge workers. The static nature of tools and processes is no match for the adaptation that people can provide to deal with the ever-changing nature of our project and our industry.

Each project operates under a different climate and set of working conditions. The factors that influence a project include size, criticality, deadline, and required quality. There is a general perception that you always need to change the process to deal with these project differences. Creating agile processes for each of these types of projects would mean that there would be hundreds of new agile processes. Instead, we can understand how these factors affect our process and utilize a context-based approach.

A context-based process allows us to tune the process to the context of our project. The quality criteria used for release are often driven by the project type. Context-driven testing bases the test approach on the factors of each project as well. The idea behind context-driven testing is that the successful approach to testing one type of application may be criminal on another type. Test thresholds, metrics for determining the shipping quality, may be used to govern the test and release approach.

The test thresholds are determined by the project team and recorded by the test team. Only one test threshold is required of an MSF project. This is code coverage for unit tests, a metric that measures the percentage of code that is tested by a set of unit tests. Like many of the other agile processes, MSF for Agile Software Development requires unit testing as part of its development activities.

However, the effectiveness of this safety net is measured in MSF. Normal test-driven development can account for 50 percent to 70 percent code coverage on many projects, but to achieve higher levels of code coverage requires more complex techniques such as mock objects [13]. Some projects, like a data converter for a one-time use, may be fine with a lower unit testing code coverage threshold for this safety net. A critical system such as an automatic pilot system probably requires a greater level of unit testing.

These metrics may be extended to

govern the project as well. For example, maximum bug debt, the maximum number of bugs that a developer may have, can be used to determine when an iteration devoted to fixing bugs (called bug allotment iteration) should be scheduled. When the number of bugs exceeds this threshold, this is an indication for the project manager to provide a whole or part of iteration for fixing bugs.

Responding to Change Over Following a Plan[7]

Wouldn't it be nice if you knew exactly what had to be done at the beginning of a project? How about if there were absolutely no surprises during the project? There are a few very small, straightforward projects that enjoy this nirvana. When the rest of us try to achieve this ideal condition, we find ourselves faking a rational design process or behaving as if change does not happen [14].

However, in the real world of software development, requirements change. We may also discover an aspect of the technology that we are using that we did not previously know. We learn about the system that we are building in the process of building it. The fact is, we can talk about these fairy-tale projects where change does not occur, but reality has a nasty habit of creeping in.

So why not plan for reality rather than trying to aspire to a mythical standard that is unattainable? In fact, we can do even better; we can use reality to develop more optimal software development processes. While our business analysts are gathering the requirements, what are our developers doing? How about our project managers? While our project managers are planning, what are our developers doing? How about our testers?

The answer is that they should all be working in parallel. While our business analysts understand the requirements, our project managers are planning, our developers are developing, and our testers are testing. How can we do this? We accomplish this through staggering the work, setting up coordination points, and providing only what is needed in a just-in-time fashion. For example, we only write the scenarios for the upcoming iteration, we plan one iteration at a time, architect only the necessary pieces, develop the functionality in the iteration plan only for this iteration, and write test cases for functionality planned in the current iteration.

Conclusion

Personas, shadow applications, and test thresholds are part of Microsoft's new agile software development process, MSF for Agile Software Development. These techniques provide alternate ways to satisfy the value statements of the Agile Alliance. They have been proven through their repeated use in delivering Microsoft software development projects.

Becoming agile is as much about changing your state of mind as it is the adoption of new practices. This article shows some new techniques to introduce agile software development to many of the roles that have not been included in many of the agile processes. By using these techniques in an agile way, we can extend agile software development processes to the entire organization.◆

References

1. McMahon, Paul E. "Extending Agile Methods: A Distributed Project and Organizational Improvement Perspective." CrossTalk May 2005 <www.stsc.hill.af.mil/crosstalk/2005/05/0505McMahon.html>.
2. Microsoft Developer Network. The MSF for Agile Software Development Workbench. Microsoft Corporation, 18 Sept. 2005 <http://lab.msdn.microsoft.com/teamsystem/workshop/msfagile/default.aspx>.
3. Miller, Granville. "Want a Better Software Development Process? Complement It." IEEE IT Professional 5.5 (Sept./Oct. 2003): 49-51.
4. Beck, Kent, and Martin Fowler. Planning Extreme Programming. Addison-Wesley, 2000.
5. Schwaber, Ken. Agile Project Management With Scrum. Microsoft Press, 2004.
6. Beck, Kent, et al. "Manifesto for Agile Software Development." Agile Alliance, Feb. 2001 <www.agilemanifesto.org>.
7. Cooper, Alan. The Inmates Are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity. 2nd ed. Sams, 2004.
8. Armour, Frank, and Granville Miller. Advanced Use Case Modeling: Software Systems. Addison-Wesley, 2000.
9. Eckstein, Jutta. Agile Software Development in the Large: Diving Into the Deep. Dorset House Publishing, 2004.
10. Ambler, Scott. Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process. Wiley, 2002.
11. Brown, William J., Raphael C. Malveau, Hays W. "Skip" McCormick (III), Thomas J. Mowbray. Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, 1998.
12. Boulter, Mark. Smart Client Architecture and Design Guide. Microsoft Press, 2004.
13. Astels, David. Test Driven Development: A Practical Guide. Prentice Hall, 2003.
14. Parnas, David Lorge, and Paul C. Clements. "A Rational Design Process and How to Fake It." IEEE Transactions on Software Engineering 12.2 (Feb. 1986): 251-257.

Notes

1. This is the third value statement from the agile manifesto [6].
2. There are many different names for this role depending on whether the project is created for commercial or internal use. The role name is not as important as the function that it performs.
3. This is the second value statement from the agile manifesto [6].
4. Class or method structure for the high-level components can be generated from a shadow.
5. Entropy is the tendency for software to become brittle and difficult to add to or change over time.
6. This is the first value statement from the agile manifesto [6].
7. This is the fourth value statement from the agile manifesto [6].

About the Author

Granville "Randy" Miller is the architect of Microsoft's agile software development process, Microsoft Solutions Framework for Agile Software Development. He has two decades of experience in the commercial software industry and has spoken at many international events, including XP200x, Conference On Object Oriented Programming Systems, Languages and Applications, Web Services Edge, Software Development West, Microsoft TechEd, and others. His interests include software development technology and agile software development. Miller is author of "Advanced Use Case Modeling" and "A Practical Guide to Extreme Programming."

Microsoft
randymil@microsoft.com