IT Solutions Consulting, Inc.

Agile Project Management

Using an Agile Methodology in a Custom Development Environment

Table of Contents

Introduction

IT Solutions is a full service IT consulting firm and Microsoft Gold Certified Partner located in Fort Washington, Pennsylvania, offering both network services and custom application development. The application development team is made up of developers, project managers, and client service/quality assurance personnel.

While our team has generally developed workgroup level solutions for small businesses, educational organizations or departments within larger companies, a number projects in the past ten years have exceeded 2,000 hours. Management of any project can be a challenge, but complex projects that may span a full calendar year can be particularly difficult to manage.

For many years we followed a traditional waterfall approach to project management using a flat-fee model for cost estimates. For the most part, this approach worked well. Some projects went over budget and others came in under. There were a variety of reasons why projects went over budget, some of which were our fault as developers and project managers. Sometimes we under-estimated the time it would take to complete tasks or we didn't properly tease out enough information during planning meetings to fully understand the scope of the projects. In other cases, over budget projects were the client's fault for not disclosing enough information at the beginning of the project or changing requirements late in the process.

Our reliance on a rigid development methodology, the only one we knew well, worked for the most part but was often limiting. We continued to try to get better using this methodology, but no matter how hard we tried, we never really knew how successful we would be from project to project. We decided to look at some of the more flexible methodologies that have been emerging over the past decade and see how they fit into our environment. The result was a study of several Agile methodologies, and in the spirit of the Agile movement, the creation of a variation or flavor of Agile specifically designed for a rapid application development company or team specializing in custom engagements.

Agile: An Overview and Brief History

Agile software development is a relationship-oriented, lightweight, iterative process that results in the cyclical delivery of tested working software. There is no one Agile methodology, but rather a variety of development methodologies that all embrace a similar flexible approach.

Variations of what is now known more formally as Agile development have been around for years, but only recently have these variations or similar approaches been grouped together under the umbrella of Agile. Examples of existing Agile methodologies are Scrum, Extreme Programming (XP), the Unified Process (UP) and Evo, one of the original iterative development methodologies (Larman, 2004).

In 2001 a group of 17 software development professionals met to find a common ground between various methodologies that had been emerging over the previous decade. The result of the summit was the composition of the *Manifesto for Agile Software Development* that defines agreed-upon values of the group (Larman, 2004). The manifesto is published on the Internet at www.agilemanifesto.org. Also included on the manifesto website is the *Principles behind the Agile Manifesto*, which is a list of 12 overall principles that point directly back to the manifesto, a history of how the manifesto came to be and a list of signatories, or software development professionals, who agree with the values and principles behind the manifesto (Cunningham, 2001).

This original group formed a non-profit organization called the Agile Alliance (Larman, 2004). The group offers individual and corporate memberships and its purpose is to support Agile software projects or help start new ones. The Agile Alliance web site (www.agilealliance.org) has become a repository for articles, news, events and other resources regarding Agile software development. Members are encouraged to submit content to the web site.

The Agile methodology that we studied in greatest detail was Scrum. The approach was first used to describe product development by Takeuchi and Nonaka in 1986. The term Scrum refers to a rugby strategy where the ball is passed back and forth between team members as they advance up the field. In 1994, Jeff Sutherland began using this approach for software teams at Easel Corporation. Ken Scwaber had also been using the approach for development at Advanced Development Methods and later the two formalized the process. Scrum was first tested and refined at Individual, Inc. in 1996. (Schwaber & Beedle, 2002)

Scrum is a team-based approach to software development that focuses on meeting business needs, fostering cooperation, and building complex sophisticated systems in a simpler and more fulfilling way. Systems are developed incrementally, based on the team's skills and knowledge, not formal complex system plans (Schwaber & Beedle, 2002).

After taking an in-depth look at Scrum and the other Agile methodologies we realized that they were primarily being used in enterprise-level development environments, where one or more teams of up to seven developers would be working on a single system over several months or even years. Our projects are typically much smaller and the entire development team can, in some cases, be just one or two developers. We knew that the principles we had been studying made sense, but needed to figure out a

way to apply them in a a rapid application development context. While some of the principles could be directly applied to our projects, many others would need to be modified, tested and possibly modified again to fit. In June 2006 we began to develop a new Agile development flavor which is specifically designed for the IT Solutions development environment. That process is outlined in detail in a later section of this paper.

The Way Things Were

The traditional, or waterfall, development approach we were using consisted of five phases: discovery, design, development, testing, and release. Our goal was to complete each phase before the next phase would begin. For example, we tried to complete all discovery meetings and gather all information needed for a project before we started the design phase. Following is a description of how we approached each phase of the process along with many of the challenges we faced along the way.

Projects began with a sales meeting that typically consisted of our sales staff and/or development supervisor who met with clients to get an idea of what they had in mind. The goal of the meeting was to get a 10,000-foot view of the client's business processes, review any existing solutions and discuss how those solutions would or should relate to new development.

Scope meetings were then scheduled with the primary contact of the system and any other potential users that were available. The number and length of scope meetings depended on several factors: the assumed size of the project, how much the client was willing to pay for the scope process, and the availability of the client's staff. Unfortunately these factors were more critical in determining the length and frequency of scope meetings than the actual goal, which was to collect all necessary knowledge about the business processes and system requirements so that a project estimate could be generated and a flat-fee price could be offered to the client.

We used an interrogation-style method for scope meetings where one of our project managers and/or developers, if one was available, would sit with the client and ask as many questions as possible about the workflow and emerging system requirements. The discussions would often follow an outline that we had created based on the 10,000-foot view from the initial sales meeting. Questions would lead to more questions that branched off from the main outline. The detail of each branch of the conversation depended on how well we asked the questions, how well informed the present user was, how well the present user could answer those questions, and whether or not we allowed too many assumptions into the conversation. We sometimes discovered later that we simply didn't have enough information about a specific topic, or that our original assumptions were wrong. The key to remember is that *we* controlled the discussions, not the client. After all, *we* were the professional software developers. The client may have never been through a process like this and therefore depended on our guidance to decide when we had enough information.

After the completion of the scope meetings, we compiled all of our notes and began working on a specification document. This document contained our understanding of the business process and workflow of the organization, a draft of an entity relationship diagram (ERD) based on what we knew at

the time, and a list of tasks that we felt we would need to accomplish to develop the solution. These documents could end up being fairly lengthy, depending on the size and the scope of the project.

At this stage of the process, it was often unclear whether or not we were still in the discovery phase or if we had moved into the design phase. To some degree, design decisions had to be made by this time to at least get a grasp of the architecture. But if a client was only paying for scope to ultimately decide whether or not to move forward with the project, we didn't want to spend too much time designing. We designed what we had to, but left many questions unanswered so that we could deliver the estimate and get the green light for the project.

In the meantime, the documentation was submitted to the client to verify its validity. To what degree clients actually reviewed the documentation was unknown. A 20-page technical document that was reviewed by a non-technical administrator probably received a skim at best. When the document was approved, we submitted an estimate of hours multiplied by our hourly rate to determine the project price.

If the estimate was agreed upon, the project would begin with a more elaborate design phase. At this point, the client usually became a little bit less available and more assumptions were made about the specifics of the system. We often based those assumptions on the specification document, which was in many cases already full of assumptions we had made during scope. Our thinking was that since we developed software for a living, we knew the *right* way systems should be programmed, and an overarching assumption was that the client would inevitably agree with us.

During the development phase, the specification document, which had been modified or enhanced by this time during the design phase, was handed off to the actual developer. Developers that ended up developing solutions were not always available during the preliminary stages of discovery and design. The specification document was the primary vehicle for knowledge transfer, along with overview meetings between all internal participating parties. Unfortunately, this transfer of knowledge took valuable project hours to complete and since everybody was busy with other things, they were often rushed and not given as much attention as they warranted. How can a developer acquire as much knowledge in an afternoon or two as someone who had spent days or weeks gathering requirements, designing the documents and generating an estimate?

Once development began, a project manager, or in some cases the developer, would schedule milestone meetings with the client to discuss progress. The meetings were often held remotely via screen shares and conference calls. The purpose of these meetings was to show progress of the system and clarify any questions that had come up along the way. These meetings were all show-and-tell from the developer or project manager. The culture of the project had already been established. We were going to go away, make the software, test it and then give it to the client when we deemed it complete. These meetings were more about showing that we were actually working on the project, rather than having the client try out a piece of working software.

Our development style was horizontal, where we developed little bits and pieces across many parts of the project rather than focusing and completing a single part. We would first convert the ERD into a

table structure, and then create all of the basic elements that we *knew* were needed in every system. We used elaborate templates that had pre-built security modules, scripted navigation with dynamic graphics and color schemes. The goal was to deliver a working piece of software at the *end* of the project, so a milestone meeting would simply be a "here's where we are" meeting. To let the client drive the system at a milestone meeting was too risky, since features were only moderately tested, and nothing was guaranteed to be complete or working.

One of the greatest challenges we faced during the development phase was the juggling of resources, both internally and with our primary contact(s) at the client. We often had a single developer working on several projects and handling maintenance for additional systems. This dramatically extended the calendar length of a project's development phase far longer than if it was being developed by a dedicated resource. Sometimes developers would be moved off of projects or brought into the middle of project cycles depending on availability and scheduling.

On the other hand we often faced resource changes at the clients. The longer a 1000-hour project takes in terms of weeks and months, the greater the risk that users will change roles or leave the organization before it is complete. When this occurred, it became very difficult to try to clarify original meeting notes or the specification document with the new client contacts. New resources had new ideas, new assumptions and sometimes completely different understandings. As a result, new decisions were made that rendered the original specification document useless.

Within this environment, change requests were a challenge. Any change that was the result of a milestone meeting or a shift in process as the result of new contacts coming on board would require a change form. If we felt that the change was dramatic enough, which was most of the time, we would present the client with a change form, hours estimate, and cost of the proposed changes. Change forms were very effective when detailed specification documents were kept up to date, but this proved very difficult due to limited project hours, especially on small- to medium-sized projects. With cooperative and understanding clients this process sometimes worked flawlessly. For the majority of cases though, even with reasonable clients, the tone of the project became adversarial.

At the time of the change form, if the state of the project did not precisely reflect the original specification documents, fault for the change was terribly difficult to prove. If we deviated from the original specifications and didn't keep them up to date, we had no way to prove that we were getting it "right" according to plan. Unless a client completely conceded to the blame for the change, we were in a position where we had to charge more money or risk losing money, which put a strain on even the best client relationships. Beyond that, change forms didn't guarantee a positive result for either party. With limited time and resources, proper scope of the change was rare. Unintended and unforeseen consequences often led to far more development time than was originally estimated. The bottom line, though, was that the client needed the change and therefore it had to happen.

At the end of formal development, we rolled out an internal beta copy for testing. During our more successful projects, we had been writing test cases throughout development, which slowed progress, but made for more thorough testing in the end. For other projects, we wrote all test cases after

development and before the official testing phase. We often contracted with an outside firm to assist in this process. Testing occurred for as long as possible in conjunction with a developer making bug fixes and changes throughout the process. One of the greatest challenges of this phase was the overwhelming amount of business process knowledge that a new tester had to consume in such a short amount of time. When a test case failed, the developer had to determine whether or not it was a native behavior that the tester was unfamiliar with, a seemingly illogical business process that only makes sense within a particular context, or a real bug that needed to be addressed.

Finally we rolled the new system out to the client for a period of beta testing. How much time would actually be needed to do beta testing was unknown. We usually allocated a certain amount of time within reason, and hopefully within budget. If we were already close to the hours estimate, beta testing would be the scariest part of the project. Oftentimes, this would be the phase that would push the project over budget. Clients had every right to be picky at this point. In many cases this was the first time they had ever touched the product. Changes were made as quickly as possible and finally the system was rolled out upon agreement from the client. This beta testing phase could take a long time, depending on the client, and could keep resources tied up for days, weeks or months, causing other projects to be delayed.

A Need for Change

While we did have some great successes using the methodology outlined above, we never felt like we were really in control of those successes. Every project carried significant risk. If we nailed the estimate, had a cooperative and flexible client, allowed for a resource to be more dedicated to a project than usual, and weren't at great risk of evolving business practices, we would more than likely have an outstanding project. Each of these factors, though, depended on outside forces to align with our goals. We needed extremely knowledgeable clients who had experience doing scope. We needed our pipeline to be stable, our maintenance clients to be relatively quiet throughout the project, and to be working with a client that had proven and stable business practices. While it did happen on occasion, varying degrees of circumstances and success were more often the case.

We knew we could get better, but how? There was an assumption within our organization that since the methodology that we followed was the one we knew best and was certainly very popular and widespread, we simply needed to get better at executing each phase. We started with discovery, since it is the foundation for the project. We spent countless hours trying to improve our techniques, from audio taping the meetings to exhaustive review of meeting notes between all participants.

Another area of focus was the idea of better protection. If our discovery was improved and the documentation was reviewed and approved by the client at every step of the project, surely we would have less trouble with change forms. We needed to make sure that documents were kept up to date throughout development, just in case.

We also realized that we had some philosophical beliefs wrapped around our development environment that seemed to be getting in our way. When we looked back on the projects that were the most successful, we found that the client had more to do with the interface design than us and that those

clients had a much better grasp on their own business processes. The more unsuccessful projects were filled with assumptions we had made about the "right" way to design an application, without taking into consideration whether or not that "right" way worked within the context of the business. After all, *we* were professional software developers and *they* were busy doing their specific jobs, so we made a lot of decisions that weren't necessarily wrong, but possibly not appropriate within a particular context.

These were very difficult characteristics to face and very difficult cultural shifts to make. We started small and made some dramatic shifts in our approach to a very small, 100-hour project that was essentially an add-on to an existing solution originally developed by the primary client contact. He was more of a programming hobbyist who took the solution as far as possible until he felt that he needed some outside help.

We spent countless hours going back and forth after each scope meeting to get sign-off and approval on all meeting notes. We dotted every "I" and crossed every "T" that we could find. Although the system's interface was unattractive and fairly unorganized (according to us), we developed the new modules with a consistent look and feel. We purposely took the back seat to all design decisions, having the client mock up every layout so that we would deliver exactly what was documented.

At first we felt like we hit a home run with the project. We came in under budget and delivered a well-tested solution to the client that seemed to fit the specifications exactly. We were very surprised to find out a few weeks after rollout that the system was not being used. The end users didn't like it, found it to be cumbersome, and resisted it. While we knew that we delivered a system that fit the original specifications document, our interpretation of those specifications and the assumptions that we made along the way did not necessarily match all of the expectations of the client, particularly the end users. The specifications had not been shared with the rest of the staff, some of which would be using the system daily and some of whom had come on board in the middle of the development process.

Two problems became evident to us as the result of this project: the ability to communicate requirements from client to consultant and capture all of them in a specification document is impossible; and overprotecting ourselves from change does not foster strong, lasting relationships with our clients.

Communication that is received by a person is actually the interpretation of an external signal coming from an external source. There is no real way to determine whether or not the message that is received is the same as the message that was sent. We can never completely specify a requirement or a design; therefore we must effectively manage imperfect communication (Cockburn, 2002). The methodology we had used to this point required all communication to happen at the beginning of the project with little room for adjustments along the way, no matter how imperfect the original communication.

Although this project was a success from our standpoint, it was not a success from the client perspective. We used extensive documentation and sign-off to protect our interests just in case the final product was deemed incorrect. If the client had complained, we knew that we could match the final system with the original specification document to avoid any liability. As a result, though, the client was very unhappy and not likely to contact us for future business.

Up to this point, we had consistently maintained *Us-vs.-Them* relationships with our clients. Regardless of the success level of the project, we had always taken the position that we were on two separate teams. It came naturally as we were accountable to different entities. We were not members of a single organization; we were part of two separate organizations with different goals. Maybe we could never have prevented certain projects from failing, but we had to believe that another way of developing software might have given us a better chance. Maybe there was a way that not only helped to develop good working software, but also helped to develop good working relationships with our clients.

It was very difficult to face the fact that we might not have all of the answers and that if we wanted to do things better, we would need to change some of our core practices. The realization struck that it might not be a case of getting better at the waterfall methodology that we had worked so hard on fine-tuning over the years, but maybe there was simply a better methodology out there.

Agile: The Process

We looked at several Agile methodologies and chose Scrum, one of the more well-known practices, to study in greater detail. We realized almost immediately that these methodologies were developed for and were being used for large projects with development teams that far exceeded what we typically used on our projects. As a result, we found that we could integrate many of the concepts, but they would need to be adjusted to fit our particular development environment.

The Agile methodology that we have been developing over the past few years is based on creating strong relationships with clients that include ongoing honest communication. We employ iterative development to deliver working software at regular time intervals. Change is not only inevitable, but welcome. All requirements are written from the user perspective, rather than the developer perspective. Testing is ongoing.

Different aspects of the project will be discussed in greater detail below. Here is a summary of the entire process:

- Preliminary meetings are held to collect a list of user-oriented features. Each item on the list should describe what the user will be able to do, as opposed to what the developer needs to do to make it happen. For example, “the user can click a button and generate a report that shows total sales by quarter” as opposed to “develop a summary report based on the invoices table, summarized by quarter and attach a print script to a button on the reports layout.” This shift in terminology and perspective is critical in keeping the users in the loop and understanding what is happening. Users may have never heard of a query or summary report, but should fully understand the need to generate a sales report.

- This shift also changes the way we develop. If our goal is to satisfy the feature, we only develop what is necessary to fulfill it. There are no assumptions. Development is completely driven by the feature and the feature represents something that a user will be able to do with the system.

- These preliminary meetings are lightweight scope meetings. The purpose is to collect a list of user features, not go into any great detail *about* the features or the development of the features. Each of the features is then assigned a high-level estimate time range based on knowledge and previous experiences. This type of estimating can be very challenging at first, but becomes much easier after comparing original estimates to actual development time later. From these ranges, an estimate range for the project can be deduced. These ranges are not in stone as we haven't collected detailed requirements yet. Project cost options are discussed in detail in the next section.

- The project timeline is then broken into releases. Releases represent the logical point in development where a system can be frozen and presented to users for testing. In most cases, we have released a copy of the system into user testing every four to six weeks. It is critical that everything presented to users at the end of a release is in working order. While users are testing the release copy of the solution, development will continue on the original development copy.

- A release planning meeting is then held where the original user feature list is broken into the various releases, creating an outline of development. Features are broken into releases based on resource availability, logical development sequence, and the ability to complete the proposed development within the designated amount of time assigned to the release. The release outline can be prepared in advance of the meeting but should be discussed, edited, and agreed upon by the entire development team.

- Releases are further broken down into shorter time intervals called iterations. We typically use one-week iterations to establish consistent weekly meeting times. Considering the rapid application development nature of most of our engagements, one week is plenty of time to plan and develop enough of the system to bring back to the users. Items from the release feature list are assigned to the first iteration based on resources, logical development sequence, and the ability to complete the feature in the amount of time assigned to the iteration.

- Each iteration starts with a review of the development that took place during the previous iteration. The features are reviewed and accepted by the users or moved into the next iteration. Following the iteration review meeting is a detailed requirements gathering for the features that are assigned to the next iteration. Requirements gathering at this stage should be extremely detailed, but focused solely on the features assigned to the iteration. All necessary personnel are required to attend the planning meeting.

- Development of only the features assigned to the current iteration occurs throughout the week, with test cases being written as the features are completed. Before the iteration is completed, another person runs each test case and offers feedback to the developer so that each feature is fully tested before the following iteration review meeting.

Team

The development teams are made up of a developer or developers, project manager, project owner, and subject matter experts. The project manager's primary role is to remove any obstructions that come up along the way that prevent developers from completing their development tasks in the time allowed. They also work very closely with the project owner to make adjustments to the feature lists and requirements and to monitor the project's budget.

The project owner should be a different, well-organized person who is respected by the users of the system and able to make critical decisions about the direction of the project. He or she is the primary contact and representative for all current and future users of the system and is typically a non-technical person. While there may be situations that warrant more than one project owner for different sections of the system, it is helpful to have a single point of contact that can make final decisions. Project owners are ultimately responsible for the outcome of the project.

Subject matter experts are primarily end users who have an expertise in a particular area and are brought in and out of the project as needed. They can participate in any of the various meetings but should only be involved when necessary. It is not uncommon to have different subject matter experts participating in each release or iteration. They will ultimately depend on the system for their livelihood and have the unique knowledge and perspective needed to efficiently plan what the system should do.

Development teams can be configured in various ways, depending on the size of the project and arrangements between the involved parties. The methodology is not limited to client-consultant projects. In fact, due to the nature of ongoing, ever-changing in-house projects, the flexibility of the Agile approach can work very well for in-house teams. As long as the various roles are played by dedicated and committed individuals who seek to collaboratively reach the same goals, the methodology can be applied to any development project.

In client-consultant relationships, typically the project manager and developers are employed by the consultant and the project owner and subject matter experts are employed by the client. Developers employed by the client may also contribute to the team and be managed by an external project manager or may take the role of project owner, representing the users and managing subject matter experts for a project while maintaining their own systems.

For in-house projects, typically a developer acts as developer and/or project manager, while a non-technical decision maker, such as a department head or supervisor, acts as project owner. Mutual respect and accessibility is critical for the relationship between the project manager and project owner, particularly in an environment where internal politics could interfere with the success of the project.

Because this methodology depends on strong relationships between the various roles, all participants must commit to being available throughout the project for regularly scheduled review and planning meetings. This commitment can be difficult for some people but is critical for the success of the project. Constant communication between the developers, project manager and project owner drives the process and can make or break a successful outcome. Meetings are the key form of communication, but

certainly not the only form. Project owners also must commit to being available by phone or email throughout an iteration to discuss any questions that should arise or clarifications that are needed.

Meetings

Preliminary or *Feature Planning* meetings are conducted at the beginning of the project and result in a user-based feature list. The developer, project owner and project manager (if applicable) should attend these meetings. Subject matter experts should be brought in as necessary to gain a full understanding of what the proposed system should do. Keep in mind that the purpose of the meeting is to collect a list of items that users will be able to do with the system, not to collect detailed requirements or to make detailed design decisions. Only details that are pertinent to building a user-based feature list should be explored in great detail.

These meetings take the place of interrogation style scope meetings used in more traditional methodologies. They should take far less time than traditional scope meetings and involve users demonstrating the business process on old systems (in the case of a conversion or re-design) or talking through manual processes (in the case of a brand-new system). Not all user features will be uncovered during these meetings, but the resulting feature list should be a starting point. Throughout the development process, other features will be uncovered and some will be deemed unnecessary. This is expected and encouraged.

One of the more abstract parts of the process is the assignment of high-level time estimate ranges to the user-based feature list. Nothing can replace experience in determining how long a feature will take to develop, but it is important to emphasize that we only develop what is necessary to make the user-based feature work. For example, we may only need to add a few fields to a table, create one new layout and one script to make a feature work. Therefore, the time it would take to perform those simple tasks should be all that is considered when estimating each feature. We never assume the need for anything that isn't directly necessary to make a user-based feature work.

A total of the user feature estimates is used to determine the calendar length of the project. Keep in mind that each week time will be taken for iteration review and planning meetings, as well as testing. Actual resources available for development may only include time for three-and-a-half days per week.

A *Release Planning* meeting is held to break down the feature list into releases. Separating all of the features into releases is helpful to get an overall sense of the project flow. These release assignments will change throughout the project. It is impossible to know for sure how the project will unfold at this point, but at least there is a general idea of what will be developed and when. The developer, project owner and project manager (if applicable) should attend this meeting. Subject matter experts would only be needed to further clarify a particular feature. The ultimate purpose of this meeting is to get an overall idea of the order in which development will occur. This will allow the project owner to schedule subject matter experts and the project manager to schedule development resources.

During the first *Iteration Planning* meeting, features that have been assigned to the first release are further broken down into at least the first two iterations. The developer, project owner, project

manager (if applicable) and any necessary subject matter experts should attend these meetings. It is not necessary to plan any further than two iterations as the focus will be solely on the first one. If development is completed faster than expected, features from the second iteration can be moved into the first iteration, so it is helpful to have an idea what is coming next.

The features that are assigned to the first iteration are then discussed in great detail. This is the point where requirements are gathered, but only for the features assigned to the first iteration. No other features need to be discussed as there will be time to plan those at the beginning of their respective iterations. All aspects of the features should be reviewed. For example, if a feature will require a layout to be created, a mock up of the layout should be designed, a list of needed fields should be discussed, examples should be reviewed, etc. The developer should have all of the knowledge needed to fully develop and satisfy each feature at the end of the meeting. Since the meeting is limited to only the features for the upcoming iteration, a great amount of detail can be discussed in a relatively short period of time. Time is not wasted discussing features that will be developed in two months as the requirements could change by then and may change based on circumstances that are unknown at this time.

Subsequent iteration planning meetings are preceded by *Iteration Review* meetings, where all of the features of the previous iteration are reviewed. The developer, project owner, project manager (if applicable) and any necessary subject matter experts should attend these meetings. Each feature should be reviewed in great detail. This is an opportunity for the project owner and subject matter experts to challenge the feature, discuss scenarios that could break the feature, and test-drive the system. If requirements about a feature were accidentally omitted and come to light during the review meeting, then the feature is not accepted and it is moved to the next iteration. The omitted requirement can be discussed in greater detail during the following iteration planning meeting.

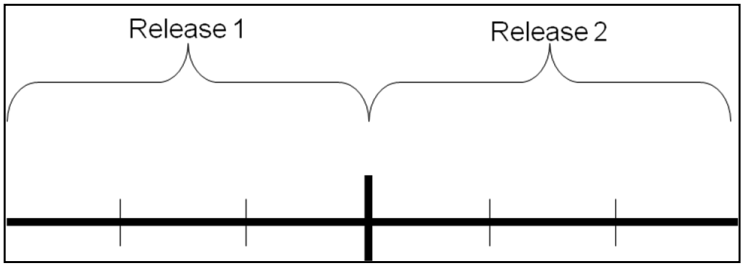
In some cases, final adjustments to a feature, i.e. position of a field on a layout, can be made on the spot during the iteration review meeting, allowing the feature to be accepted. If the feature is complete to the satisfaction of the entire development team, the feature is accepted. This does not mean that the feature is locked in stone and can never be changed. If a circumstance in the future causes more adjustments to be made, it can be added to the project as a new feature and assigned to a logical release/iteration at that time.

Following the *Iteration Review and Planning* meetings is a retrospective with the entire project team to determine if any part of the process needs to be changed or revisited. At this time it's very important to make sure that everybody is satisfied with the pace and quality of development. The process is flexible and can be tweaked per project. For example, if the project owner feels that more formal communication is necessary, a daily call can be included in the process. The key is to build strong and open relationships between all members of the team so that ownership of each phase of the project is shared.

Ongoing communication during the project is encouraged. There should be a commitment by the project owner to be available throughout the iteration to answer questions in a timely manner.

Requirements are gathered at the beginning of all iterations, but even when discussing a relatively small number of features, details can be missed. Remember, complete communication during requirements gathering cannot be guaranteed and should not be expected. Allowing for details to emerge or requirements to change when examined under a microscope will ultimately lead to far greater and more accurate delivery of the features.

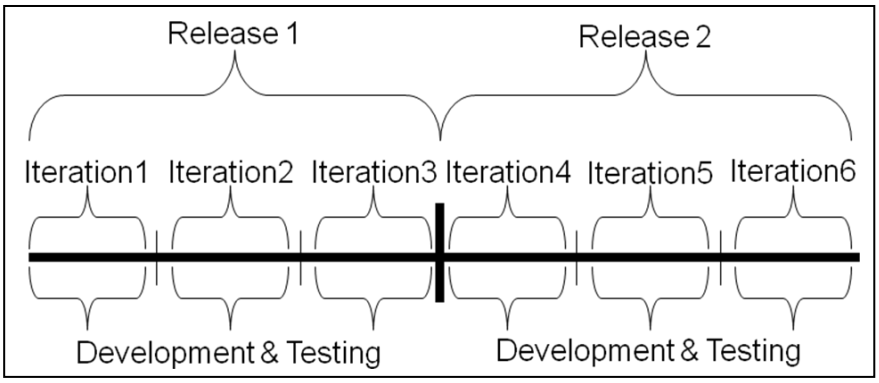
Time



Projects are first broken into *Releases* that represent a period of time in which a logical section of the solution can be fully developed and deployed into a user testing environment. In our experience, we have found it useful to develop a module or modules during a single release that will be used by a particular group of users. The focus of development for the release includes related subject matter experts who are involved with meetings for a limited time and then move directly into user testing at the end of the release. Features that are completed during early releases should be revisited during subsequent releases to ensure ongoing success and make sure that requirement changes did not affect the performance of those features.

The number of releases depends on the estimated length and complexity of the project. We have found it best to limit releases to approximately four to six weeks. Smaller projects can be developed in a single release, the result of which is a user beta testing period before deployment. Remember that iteration testing by a second developer or tester will occur at the end of all iterations, so this beta testing period will not be the first testing that will occur on the system.

The assignment of features to a release depends on the logical sequence of development and the amount of resource development time available during the designated time frame. Since a copy of the solution moves into a user beta testing environment for users, the selection of features for a release should take into consideration that all features should work properly at the end of the release and features that are available to the user during testing should not be partially developed.



Releases are further broken down into *Iterations*, which are fixed-length time boxes that make up the development cycle between requirements gathering and feature acceptance by the project owner for a given set of features. Iterations can last anywhere between one to two weeks depending on resource availability. We have

found the greatest success with two-week intervals. A full day of iteration planning and requirements gathering typically produces enough development tasks for a two-week-long iteration, considering that testing occurs by the end of the second week. If development resources are limited, the iteration can be extended to a one month time period, but shouldn't exceed one month.

Action

All development is represented by *Features* that are written from the user perspective. This can be one of the most difficult culture shifts when moving from a more traditional methodology, particularly when all previous project documentation and planning has been written from a technical perspective. This shift, though, proves extremely valuable for non-technical project owners and subject matter experts to understand what development will occur during each release or iteration.

Every action taken to develop the solution should align with a feature that describes something that a user will do. For example, rather than "add a contact table that stores name, demographic, address and communication information", a user-based feature could say "the user can add, edit or delete contact information in a form." While a project owner or user may not know what is meant by "table" and may not know how to determine whether the developer actually added one, he or she will clearly be able to tell if he or she can add, edit or delete contact information in a form.

Even more abstract tasks in development should be described to the team as user-based features. If a complex process needs to run in the middle of the night to update data in a particular table, the feature should describe the outcome of that process that the user can see or expect. For example, "the user can expect field ABC to be updated with value XYZ each day". That way, the feature can be demonstrated and the project owner or user can clearly identify if the feature is working.

Because of this change in context, no assumptions should be made about the development of the system. For example, tables should only be added to a solution when one is needed to make a feature work. It should not be assumed that tables will *definitely* be needed, even if the assumption is based on the feature list. The feature list could change before reaching that point in the project and work that was *assumed* could become unnecessary and therefore a waste of time. All development should directly satisfy a feature in the current iteration.

Features can be categorized into *Modules* to help organize various lists and help determine which features should be developed during the same releases or iterations. Modules can cross iterations and/or releases and are only used as a guide to help with organization. There are not hard and fast rules that one module needs to be completed before another is started.

During the iteration planning meeting, detailed requirements are gathered for only the features that will be developed during that iteration. The requirements should contain as much detail as possible to complete the development work. They will not end up in a specification document to be maintained for the duration of the project, therefore the amount of clarity needed should be based on the developer's understanding of the feature and how it should work.

Following the iteration planning meeting, the developer and/or project manager will write detailed development tasks for each feature. Since detailed requirements have now been collected, task estimates will be the most accurate time estimates for the iteration and will indicate whether or not it will be possible to complete the development work within the allotted time or if the development will be completed much faster than anticipated. Adjustments to the iteration feature list can be made at any time but should be communicated to the project owner through ongoing communication throughout the iteration.

If it is discovered during the task writing phase that the development will take longer than is allotted to the iteration, the developer can push back features into the following iteration. The key to remember is that whatever features are tested and reviewed at the end of the iteration should be completed and working. Partially developed features should never be presented to the project owner or subject matter expert(s) during the iteration review meeting.

If the task writing process reveals that the features will not fill out the total amount of time available during the iteration, features can be brought forward from the next iteration. If possible, requirements for those features can be collected in the middle of the iteration, remotely if necessary, but preferably in a face-to-face meeting.

Tasks should be written from the development or technical perspective and are used by the developer as a guide or recipe when actual development begins. Task writing presents an opportunity for the developer to do extensive planning and to think through each step of what it will take to make the feature a reality. By the time development begins, the developer should have a very clear sequence of well-planned steps to perform with the result being the successful completion of the user-based feature. All development should be tied directly back to making a feature work, therefore unnecessary development is avoided for maximum efficiency.

Upon completion of each feature, the developer should convert it into one or more test cases. Features can be broken into multiple test cases if they are too complex to be dealt with in one. If several different aspects of a feature could fail, it may warrant the use of multiple test cases, which would make retesting each aspect much easier. Test cases should also be written from the end user's perspective, similar to a user manual with step-by-step instructions on how to use the feature.

Testers do not necessarily have to be developers. It is a good idea that they have a sense of native database behavior because they will run each test case as if they are a user of the system. Remember, they are testing features that will explain what a user can *do* with the system; therefore the test will pass if a user can successfully perform the described action. A tester does not need to look under the hood of a solution. If the feature works, it passes; if it doesn't work, it fails.

One advantage to having developers perform the tests on each other's work for different projects is cross training. Familiarity with the business process and terminology of a solution is critical when bringing in a different resource to help finish a project or to back up the original developer in the event of unforeseen changes.

Any test case that fails should be fixed and retested before the iteration review meeting. Testing typically occurs during the last day of the iteration. Testing should begin early enough in the day to allow for fixing and retesting. A comprehensive list of test cases should be maintained to account for regression testing during later releases. So, testing occurs during iterations, during iteration review, and at the end of each release to allow for bug fixes and requirements changes throughout the development process, as opposed to limiting it to only the end of development or post-rollout when the budget has been depleted.

Closing the Deal

At IT Solutions we have employed two methods to address the billing of Agile projects: *Target Cost* and *Fixed Price, Variable Scope*. Target Cost is a more flexible approach with built-in incentives for both us and our clients. Fixed Price, Variable Scope is geared toward projects with tighter budgets that cannot be adjusted during the project.

Target Cost

The key measurement of the Target Cost approach is the target price. This number is typically determined by a rough estimate of hours on the work to be performed multiplied by the hourly project rate. For example, if the hours estimate on a project is 100 hours and the billable rate is $100 per hour, then the target price for this project is $10,000.

If the development effort is achieved in 80 hours (20 hours below the estimate) or at a price of $8,000, then the difference between this price and the target price is split between the client and the consultant; both are rewarded with $1,000. If, on the other hand, there is an overage of 20 hours or a price of $12,000, the consultant can only bill for half of the time required to complete the work, or $1,000.

Unlike a time-and-materials approach where there is little motivation from the consultant's perspective to manage costs on the project, or a flat-fee approach where there is little motivation from the client's perspective, in a Target Cost approach there is equal motivation for both parties to control costs.

The client will invariably have changes to the original requirements, especially in longer engagements. Additionally, they may see the capabilities of the custom software as it is being developed and get ideas on how to utilize the technologies they are being exposed to in other areas. This is natural and should not be discouraged. The Target Cost approach encourages theses modifications with a change-control structure that is flexible and doesn't favor or penalize one party any more than the other.

The only thing that changes the target price itself is an enhancement, a brand new feature not originally discussed during the preliminary planning meetings. So, using our example above, if a brand new 20-hour module is introduced into our $10,000 project, the target price is moved up to $12,000 to accommodate the change. This does not necessarily mean that the price of the project has increased. If the project is under-budget in other areas by a total of 20 hours then the only effect this will have is that the project will finish *on* the original budget rather than *under* the new budget. All other changes (fixes, clarifications, etc.) are included under the original target number.

Fixed Price, Variable Scope

When budgets are set in stone and cannot exceed a certain amount, custom software can still be developed using an Agile methodology. The project estimate is based on high-level estimating after a list of user-based features is gathered during the preliminary planning meetings with the client.

If the estimate exceeds the project's fixed budget, then all features are prioritized with the highest priority features at the top and lowest priority features at the bottom of the feature list. A water line is placed in the list under the feature that represents the last feature that can be developed within the fixed budget, starting from the top and working down. The bottom of the feature list, or those features located underneath the water line, should represent low priority items that do not affect the basic functionality of the software. If features that are critical to the basic performance or functionality of the system fall beneath the water line, then it may be deemed that the budget must be increased to commence development or the project should not be taken.

As a project unfolds and business requirements change or need to be adjusted new features can and should get added to the feature list. If they are high priority and need to be added above the water line, then a comparable feature from above the water line needs to be moved below it. If a new feature is not considered high priority and is more of a wish list item, it should be added in the appropriate spot on the list, even if it is below the water line.

Business requirement changes do not always necessarily mean that features will be added to the feature list. Sometimes while working through the process of planning and developing software, an organization can make its internal processes more efficient, reducing the need for certain steps in the workflow. This can reduce the number of features required in a custom solution. As features fall off of the list, lower priority items can move up, even crossing the water line into the doable time window while still keeping the project within the budget.

Regardless of whether features are added or removed from the feature list, the waterline should remain the gauge that determines features that can be completed while keeping the project within the original budget. As articulated in the Agile methodology and explanation of the Target Cost billing approach above, open and honest communication between all members of the development team is a crucial component of this approach. Decisions about priorities and the monitoring of the project should be performed by the entire development team, which includes project owners.

Bibliography

Cockburn, A. (2002). *Agile Software Development.* Boston: Addison-Wesley.

Cunningham, W. (2001). *Manifesto for Agile Software Development*. Retrieved June 8, 2007, from Manifesto for Agile Software Development: http://agilemanifesto.org/

Larman, C. (2004). *Agile & Iterative Development.* Boston: Addison-Wesley.

Schwaber, K., & Beedle, M. (2002). *Agile Software Development with Scrum.* Upper Saddle River, NJ: Prentice Hall.