RJ10476 (A1011-022) November 19, 2010
Other

IBM Research Report

Software Development as a Service: Agile Experiences

Tobin J. Lehman
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

Akhilesh Sharma
IBM Global Services
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Software Development as a Service: Agile Experiences

Tobin J. Lehman
Almaden Services Research
IBM Almaden Research Center
San Jose, California, U.S.A.
toby@almaden.ibm.com

Akhilesh Sharma
IBM Global Services
IBM Almaden Research Center
San Jose, California, U.S.A
akhi@us.ibm.com

Abstract— **At the IBM Almaden Research Center, we have been working with other divisions of IBM, offering Software Development as a Service (SDaaS). We work with our colleagues to understand their business and pain points, and then leverage that knowledge to build an application that at least meets, but ideally exceeds their needs. In our experience, applications built in the mode of Software Development as a Service have the following characteristics. There is a single client that owns the program requirements; the requirements are not fully formed; and, the software is at least partly, if not completely, custom.**

With many decades of software development experience in our group, we have used many different techniques for conducting software projects. In SDaaS, the style of interaction between the client team and the development team is a natural fit for Agile Software Development methods. Although our conditions do not fit the ideal Agile Software Development project profile, we have consistently experienced significant gains from taking the approach of iterative and incremental development. We have used the Waterfall, or plan-based, method many times for past projects, and we have consistently paid a significant price for the "Big Design Up Front" problems that go with that type of schedule.

Although agile methods have consistently outperformed other methods in the early stages of a project, they do less well as the application program grows and the project matures. Once an application is in production and has an established user base, the emphasis of the client and development teams change from fast, easy and elegant feature additions to stability, release planning and maintenance. The imposition of a fixed schedule naturally transitions the overall project management into a plan-based model. However, we came to realize that agile methods still play an essential role, even when the overall project has matured to a steady state and is managed in a plan-based fashion. When developing a new program requirement, unless it is completely understood (i.e. all aspects and expected results documented), then it must go through an agile prototyping phase in order for it to be properly developed and scheduled. That agile prototyping phase is still bound inside a plan-based schedule, but that leaves room for iteration and innovation that we would not otherwise have in a fixed schedule.

In this paper, we examine the various forces that influence both the client and development organizations that are engaged in Software Development as a Service. Using several projects from our past as examples, we show that the level of program technology and requirement understanding determines which development method will be more successful. Plan-based methods will work if all aspects of the project are known and there are minimal variables. Otherwise, our experience shows that an Agile Development model is likely to be the best choice for the beginning of a project. Once a project reaches steady state, with production releases, then a hybrid model works best, where agile methods are used to prototype new features, but bounded inside of a plan-based schedule.

Keywords: Agile Development, Plan-based Development, Hybrid Development,

I. INTRODUCTION

In the Services Research Department at the IBM Almaden Research Center, we look for ways to make the IBM Service Divisions more productive and more effective. For the past four years, we have been working with IBM's Global Technology Services (GTS) and Global Business Services (GBS) and building a suite of tools under an umbrella project called Solution Definition Manager (SDM). SDM comprises tools for engagement solution design, costing and pricing, as well as an information repository with analytics capability.

The term we apply to our software development infrastructure is "Software Development as a Service". We work with a single client per project, which means, there is one division that has as its representative a panel of subject matter experts (SMEs). The SMEs describe the overall mission statement, they offer problem statements and they provide feature suggestions. We take all of that information and use it to guide the design and prototype of the application.

Being a research group, we examine all aspects of our projects in order to optimize our effectiveness. Besides diving deep into the details of our company's services business to build the best possible services-oriented software, we also examine the software development process itself. In recent years there has been a shift from classical plan-based methods (sometimes referred to as "Waterfall" [1]), to a technique called "Agile Methods" [2]. Knowing the challenges of plan-based project management, when the SDM project started, we were eager to try new techniques

that might improve the traditional process of software development. This paper describes our experiences and observations during that journey.

1) Different Strokes for Different Folks

In software development, there is a spectrum of application requirement specification. The endpoints of the spectrum are "Nothing Defined" and "Everything Defined". Rarely does a software project qualify for either end; it falls somewhere in the spectrum of completeness (Figure 1.1). The more the requirements are defined, the less investigation, research and exploration that will be needed, and hence, the more accurately we can define the project size and schedule. However, the newer (and perhaps more interesting) the project, the less well defined the requirements.

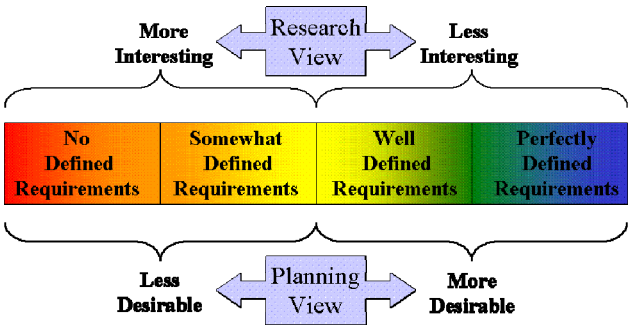

Fig 1.1: The Requirements Spectrum

The first component of the SDM project, the pricing tool, had many goals, but very few well-defined requirements. As a result, it was necessary to build the project in "prototype mode", which meant constant customer interaction, weekly design iteration and weekly demos and reviews. In short, it required the use of Agile Development Methods. The result was a substantial tool, built in record time, which met the client's needs. At first, the SDM users were skeptical of the lesser known agile process, but over time, they became convinced of the value of iterative development and they appreciated the ability to introduce changes throughout the development process, even late in the schedule. Riding on the success of the pricing tool, the SDM team went on to build the costing tool and the gross profit estimator tool.

The development of the pricing tool fit the ideal conditions for Agile Methods: a small group of relatively senior people, a small group of experts representing a single client, and a completely new software application. Iterations were held weekly (demo, followed by review), so progress was swift.

With the stunning success of the pricing tool, we declared agile methods to be the preferred development method. It became the default mode for the subsequent projects. Matching our own (initial) experiences with some of the published literature on Agile Methods [2], we felt reassured that "Agile = Good" and "Waterfall = Bad".

We were unprepared for what happened next. Excited by the prospect of working with a new process, we consulted the literature that extolled the virtues of Agile Methods, mostly as they apply to new projects [3], [4], [5]. We missed the more balanced articles, like "Get Ready for Agile Methods, with Care," by Barry Boehm [6], which discussed the more realistic side of Agile Development. To summarize, Agile Methods are great for some programming environments, but not for all. For projects that involve large teams, well-defined requirements, clients needing high assurance and large code-bases, the plan-oriented (i.e. Classical Waterfall) project profile not only works but also is often necessary. Therefore, early in a project, when the team is small, the requirements are not yet well defined, the project code base is small and the customer is interested in seeing significant progress, agile methods generate the best results. However, as a software project transitions from a small prototype to a large stable system with a large team, with promises to keep and dates to meet, then agile methods alone do not suffice; some additional mechanism is needed.

This paper discusses our experiences with various software projects, looking at the various factors that made them appropriate for Agile Development methods, for plan-based methods or a mixture of the two. This paper is organized as follows: Section 2 discusses some of the basic points of Agile Development Methods and Plan-Based Methods; Section 3 presents our experiences with several software projects, where we examine the characteristics of those projects in light of the strengths and weaknesses of agile and plan-based methods. Section 4 provides an analysis of our data and gives some observations on software development projects. Section 5 presents our new proposed method for mature software development, a hybrid approach that includes aspects of plan based methods and agile methods. Section 6 concludes the paper.

II. Characteristics of Agile and Waterfall Projects

A. Waterfall, or Plan-based projects

The traditional (i.e. old) software development model follows a modified version of the waterfall development model, which has its origins in the manufacturing and construction industries. In the classic waterfall model, each process phase (requirements, design, implementation, verification and maintenance) flows sequentially and cascades downward to the next phase. Unlike manufacturing projects, the typical software project does not have sufficient requirements specified early on in the project to follow this model. As a result, the requirements phase not only takes longer than expected, but it also is typically rushed to an early, unfinished end, which then negatively impacts the remainder of the project.

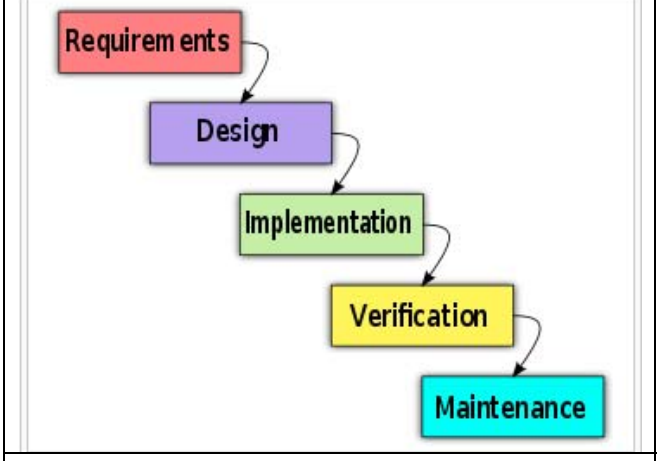

Fig 2.1: The Classic Waterfall Schedule [1]

With a Plan-Based Method, the focus is on the plan: the deliverables, the dates and the end date. The main problem with a plan-based project schedule is that it does not handle variables and contingency well. Unfortunately, variability is at the heart of almost every software project. Therefore, issues often arise in the requirements phase, then that triggers problems in the design phase, which then result in conflicts and defects in the implementation phase. The schedule slips, confidence lowers, moral suffers and the project becomes a death march [7] . There are obvious flaws in using a plan-based schedule for most software development, and yet the practice continues [1], [8].

B. The Agile Software Development Process

Agile development methods differ from plan-based development methods in one fundamental way: whereas plan-based methods make the plan itself central figure, agile methods focus on the customer. Introduced in 2001, the Manifesto for Agile Software Development [4] has the following values:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

The authors note that while there is value in the items on the right, they value the items on the left more.

As shown in Figure 2.2, Agile Software Development includes the notion of iterative cycles, where all of the phases are interconnected, each phase being a feedback mechanism for the others [9]. Essentially, it is accepted that no phase is ever finished; all phases keep evolving. This is in stark contrast to the plan-based schedule, which assumes that each phase reaches a logical conclusion before the next phase even starts.

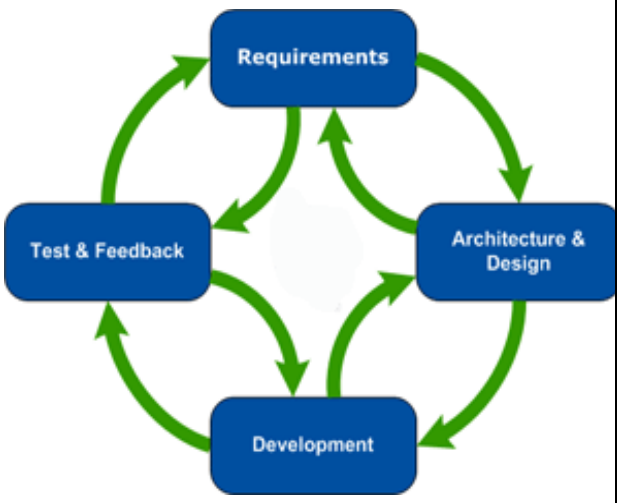

Figure 2.2: Agile Development Process Interaction [9]

The Agile Manifesto includes twelve principles:

1. Customer satisfaction by rapid, continuous delivery of useful software
2. Welcome changing requirements, even late in development
3. Working software is delivered frequently (weeks rather than months)
4. Close, daily cooperation between business people and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (co-location)
7. Working software is the principal measure of progress
8. Agile processes promote sustainable development
9. Continuous attention to technical excellence and good design
10. Simplicity
11. Self-organizing teams
12. Regular adaptation to changing circumstances

C. Agile and Plan-based Methods; Strengths and Weaknesses

A plan-based schedule represents reality; there is a start date, an end date, and finite set of resources. At a rudimentary level, it would appear that a plan-based project offers the best visibility and best understanding for funding organizations and on-lookers. Everything is documented and accounted for. Problems arise, however, when unforeseen variables are inserted into the plan, which break the schedule and ruin confidence in the team. Also, the primary issue with plan-based schedules, as stated by developers, is the "Big Design Up Front" (BDUF) problem [10], which assumes that not only is everything known up front, but that it is possible to create the finished architecture and design of the system before coding starts. Our experience indicates

that a complete design, prior to any prototyping, is rarely complete.

Agile development methods were born from the frustrations experienced by developers and customers alike when dealing with the considerable process-based obstacles that have evolved over many years of plan-based software development. Agile development methods, using customer feedback rather than planning as their primary control mechanism, have proven themselves as a more effective approach to starting a new software project; iterative development, with frequent customer interactions, greatly improves both project direction and customer morale.

Agile development places the focus on the customer. By putting the customer first in the priority list, an agile approach ensures that when the development is done, the customer receives an application that suits their needs. On the other hand, plan-based project development places the focus on the project schedule. Usually, there's a fixed project budget and an overall schedule. A plan-based project helps the funding organization track the project progress.

D. Reality Inserts Itself

When operating in the mode of Software Development as a Service, we have two types of customers: the paying customer (the funding agency) and the users. Agile software development is oriented toward the users, making sure that they get the best possible application. However, it is not oriented toward the paying customer, as the time and money spent is variable based on the iterative development and interaction by the users. We cannot satisfy just one customer, and we certainly cannot alienate the customer that pays for the project. We must satisfy both sets of customers.

Because we have both types of customers, and hence, a need for both Agile Methods and Plan-based methods, we have to figure out a way to make them work together. Looking back over the many software projects we have been involved in, we have seen many different project characteristics and many different project management styles. In this next section, Section 3, we review four projects, exemplars of different categories of projects from our past.

III. FOUR ARCHETYPAL PROJECT EXPERIENCES

Our collective experiences cover 50 years of software development projects. We describe experiences with four projects that match the four archetypical project classifications. Depending on the level of understanding, the structure and schedule of the project and the specifics of the program function (the requirements), any software project lies in one of four sections of the program requirement spectrum, ranging from one end of the spectrum, where everything is known, to the other end, where practically nothing is known.

A. Case 1: Zero Requirements

On one end of the spectrum, Case 1 is an example of a software project that is lacking requirements and having numerous other variables, such as unspecified development language and application platform. One project having this characteristic was the creation of a financial application for one of our internal clients. This was a case where the client was using a set of tools (some commercial products, such as MS Word and MS Excel, some internally developed Visual Basic applications) for computing risk, inflation, markup and other factors for service contracts. The clumsy interaction between their tools was creating unnecessary delay and aggravation in a process that seemingly should have been straight-forward. In an effort to show how our group could be of service, one of our team members created a simple web application to show how the process could be done differently – and much more easily. The web application generated much interest, but it partly missed the mark because the client needed network-disconnected client operation. However, that initial interest was enough to start an agile project where a fairly vague set of goals were increasingly refined into requirements through weekly (and sometimes daily) iteration.

With regular iterations, requirements were introduced based on previous pain points, first at a high level and then at a low level. Eventually, the 17 original pain points evolved into 17 requirements, but during agile development, expanded to 43 requirements. The web application was discarded and replaced with a Java application based on the Eclipse Rich Client Platform. In the end, the client was delighted with the result.

Observation: As we assess the experience, it's clear that no plan-based schedule could have worked, because the project was in the mode of self-discovery during much of the development process.

B. Case 2: Partial Requirements

Case 2 is an example of a project where the requirements are partially defined. This project was at a large industrial manufacturing company to implement electronic distribution and management of the software packages to the final product. High level project goals were defined at the beginning of the year. There were about 2-3 releases planned for the year. At the beginning cycle for each of the releases, detailed requirements for the production release were defined. The project assumed that there will be a few goals that need prototyping, without committing this work towards a specific production release. For a development cycle for a release, it had 2 parallel efforts, one for the production level feature development and second one for the prototype development.

For the production level features, the first part of the release cycle was dedicated towards requirements definition, UI screen mock ups, requirement reviews etc. to finalize the release scope, release date and effort/staffing levels. The development was done for about 10-12 weeks duration with 2-3 iterations during this period. During this iterative development cycle, there was no pressure on doing demos to the customer. The development team did the demo or revised the use case/UI discussion if any changes were discovered requiring customer input. The development team had about 1 week of time at the end of development cycle for overall integrated testing prior to the start of the formal functional

testing by test team on a test environment. It was followed by usual User Acceptance Test, Pre-production deployment and production deployment. If the users had some suggestions to improve the tool or had any additional feedback, they were reviewed back with users than with the business SMEs for evaluation to incorporate these suggestions in future enhancements. It was agreed that these changes can not be incorporated in the current release.

The project execution was relatively much smoother due to the following:

There was only one business owner who made the critical decisions on functionality, scope, schedule and details.

- There were only 2-3 key business leads that made the critical decisions on functionality, scope, schedule and details.
- Close cooperation between the business and the development team.
- A dedicated business analyst who was a critical bridge between the business owner and the development team.
- Business leads and development lead had high degree of trust and respect for each other. This resulted in a lot of frank discussion between the business owner and the development team, open exchange of ideas, suggestions, evaluation of various technology options and feasibility or challenges of accepting any late requirements.
- Upfront acknowledgement by business and development if any functionality or the technology options were not clear, were targeted for the prototype work The prototype development was not for production release, rather for either proof of technology concept or the prototype functionality to solidify requirement for the next production release.

Observation: The project had a somewhat flexible schedule and scope. It was important for a successful delivery that both business and technical teams worked very closely on the scope definition for a release, while defining specific prototype work to help plan subsequent releases. This was a true trusting partnership resulting in higher degree of customer satisfaction, better quality delivery on time.

C. Case 3: Good Requirements

Case 3 is an example of a project with well-defined requirements, which was a case involving a biotechnology company that is part of the highly regulated health industry. Due to FDA regulations, requirements were supposed to be well-defined, reviewed and sign-off before the development can start. This was a large project and it was a challenge to completely define all of the requirements. To overcome this challenge the project was divided in 4 phases. Overall project schedule was defined for the project including final phase for integration testing, regulatory approval and the project launch. Even though the schedule to complete the requirement for each phase was defined but there were challenges in getting all of the business SMEs to agree on the requirement and system features. Once the requirements for the first phase were completed, development started. Then, while requirements for phase 2 were being defined, developers were busy doing the development for phase 1 and at the same time, lead developers/architects were also participating in requirement discussions and reviews for phase 2. As the requirement definition schedule fell behind to keep the final delivery dates, business worked with the development team to reduce the project scope, increased development and test staff. Project was completed as scheduled, however with a much smaller feature set/scope than originally planned.

Observation: The project had a fixed schedule, however for a successful delivery it required the scope and staffing level to be continuously adjusted to keep the schedule.

D. Case 4: Great Requirements

The fewer the variables in a project, the less doubt there is about what the software program will do. Unfortunately, except for those applications defined in a classroom environment, real software projects rarely have perfectly defined requirements. However, for every rule, there is an exception, and our exception is Case 4, which was a software project for a telecommunication client. The project was to replace the existing backend infrastructure to support the cell phone web interface with IBM hardware. It required some associated custom software development. Since there was an existing system already working in production, the requirements were perfectly well-defined.

Implementation of this project was divided in two phases. Phase 1 covered some hardware related changes, introduction of some new components, and base implementation. Phase 2 covered performance related changes and remaining feature development. Given quality requirements in telecommunication, a thorough documentation of requirement and user interface wireframes were done in early part of the project phase. This was truly a waterfall model where design and development were done after the requirements were completed. The team did face challenges with the original timeline planned and due to various reasons the timeline was adjusted a few times.

The project development followed traditional waterfall methodology, even though there were some overlaps between various development phases. There was some prototype work for an early validation of some of the new technologies and techniques and not for any requirement validation. User interface wireframe details and elaborate requirement were very useful in avoiding any scope creep. As the customer realized new requirements or changes to existing requirements, the changes were obvious and appropriate change management process was followed.

Observation: For any project to have such well defined requirements and so few variables is rare. However, even in this case, where the requirements were stable and only a few variables existed (from the integration), the plan-based schedule had to undergo adjustment to compensate. It seems unlikely that except for the most trivial projects, any project that is run with plan-based methods will have to adjust its scheduled dates to adjust for each significant variable in the project.

IV. Observations, Analysis and Lessons Learned

Based on observations of our four archetypal projects, and the sum of other projects we have seen, we can conclude that no single development method is the obvious choice for all projects. The choice of development method is based first on the number of variables in the project and second on the flexibility of the client. If there are many variables in a project, then a plan-based project simply cannot work. Trying to layout program requirements for a program that is not well understood is much like playing a chess match and trying to think ahead 100 moves. Even worse, to continue the analogy, some project changes would be similar to changing the game board, changing the rules or even changing the pieces in the game. Considering all of the possible outcomes and inter-play between the features is too complex, especially if there are many new or unknown features. The so-called "Big Design Up Front" (BDUF) approach does not add value in a highly variable environment; in fact, it is a detriment. Not only does the initial design have to undergo constant changes, but other project decisions that were dependent on the initial design were believed to be stable (e.g. the application data model, user interface design) must be revisited and reworked each time the overall design changes. The main (declared) benefit of BDUF, which is to remove mistakes found during design time rather than during implementation time, is negated and additional project time is lost to design and documentation reworking. When there are significant unknowns, then agile development methods, using iteration to adapt to change and to enhance both the design and project understanding, provide a better balance of results to effort.

Plan-based principles and Agile-based principles both have a certain amount of unrealistic and idealistic assumptions. Plan-based methods have the assumption that enough of the project can be known in advance that a realistic schedule can be built. Agile-based methods have the assumption that iterative programming can happen continuously and independently of other activities in the project. Our experience with Agile methods showed that as our application got more complex, defect fixes were often not self-contained; the updated code sometimes crossed many component boundaries and affected multiple features.

A. The Effects of Reality on a Plan-based Project

Except for a very few rare cases, software development projects will have some number of unknowns, or variables, that will require an exploratory phase to resolve. Although the exploration itself introduces a certain amount of variability into the schedule, that can be controlled and included, provided that the variability concept is part of the plan.

B. The Effects of Reality on an Agile-based Project

Agile-based projects have a strong assumption that multiple teams conducting iterative development can continue independently on the same codebase. That does not fit with our experience. Once a program is released to customers, then the team must maintain a separate version of the code (i.e. a stream, in Eclipse terms) on which the support team can introduce fixes for customer defects.

There is a point in the project when the initial agile development phase must end. The real world inserts itself into projects in the form of ship dates, user requested feature sets and finite budgets. That's not to say that Agile Methods are completely removed from the project, but they are used in specific instances, with bounded time constraints.

C. Reality Creates Common Ground

For agile-based projects and plan-based projects, the effects of reality cause significant changes in how the projects should be conducted. Plan-based projects typically cannot afford the "big design up front" phase, because they will either run out of money or their funding source will grow impatient, watching a stream of money go out, but seeing only a late design document come in. Often, the design gets cut short and emergency prototyping follows. Hence, out of necessity, a certain amount of agile development is used to keep the executives from cancelling the project.

On the other side of the coin, agile-based projects typically cannot ignore the schedule that is imposed on them by the user base and the funding source, once there is a production release. Hence, out of necessity, certain amount of plan-based scheduling is used to align the project with the outside world.

D. Testing Plays a Huge Role in the Schedule

As an application grows in size, the complexity also increases. We found that new additions and bug fixes often had unexpected and unwanted effects on the existing functions. So, like any other project in this situation, we placed a heavier emphasis on regression testing, to verify and validate that the existing function continued to work as code changes were added to the system. The unexpected effect of extra regression testing was that it took "test cycles" away from the test team, thus making their Function Verification Test phase longer, which then changed the overall project release schedule.

Our group uses four types of testing, and these types are consistent with the software test literature [11]. There are unit tests, which validate the correct function of an individual class or component (i.e. unit). There are function tests, usually conducted by the test team, that verify that the individual user-visible features are performing correctly. There is the system test process, where the test team batters the tool, simulating heavy use, to see if the tool behaves correctly (no crashes, no bad output). Finally, there are regression tests, which verify that the tool continues to give the expected output, even after changes have been made to the system.

Part of the agile credo is the idea that changes to the project are welcome, even late in the development cycle. However, it is important to note that project or requirement changes are welcome **only** during the actual development (coding) phase, not to the entire development schedule, which includes the testing phases. The tests are there to verify function, stability and sometimes performance, and

when the system passes all of the test phases; it is considered ready for release. If any significant development is performed on the source code during the test phase, the tests must be rerun with the newly updated code. In the earlier phases of our project we had not yet automated function and regression testing11, so there was a sizable manual component to testing. A reset to the full test cycle (because of development changes, or any other reason) had a huge impact on the overall schedule and often forced a change in the release date.

E. Development Roles:

A software development project requires a surprisingly large number of roles. Figure 4.1 shows the roles that we eventually identified in our project, despite the fact that the first component of the SDM project started with three people. In the early stages of a project, each project member must play multiple roles, and some roles start out unfilled. However, as the project ramps up and the teams grow, many of the roles become full time jobs. One of the lessons we've learned from previous large plan-based projects is that the schedule for the scale up of the roles greatly affects the success of the project.

As long as the teams stay small (and agile), then it is expected that the team members play multiple roles [12], [13]. However, in our experience, as the overall team grows in size, so do the activities for each of the roles, thereby turning some (or many) of the roles into full time jobs.

- Leadership Team
  - Department Manager
  - Project/Release Manager
  - Project Lead
  - Project Architect
  - Scrum Master
- Development Team
  - Lead Developer
  - Developer
  - QA Specialist
  - NLS Specialist
- Build Team
  - Build Team Lead
  - Build Manager
  - Installation/Packaging Architect
- Test Team
  - Test Team Lead
  - FVT Tester
  - SVT Tester
  - Performance Specialist
- Operations Team
  - IT Specialist
  - Tools Specialist
  - System administrator
- Server Team
  - Database Admin
  - Database Specialist
  - WAS Specialist
  - Server Developer
- Work Practice Team
  - WPA Lead
  - WPA Analyst
- Documentation Team
  - Doc Team Lead
  - Documentation Specialist
  - Language Specialist
- UI Team
  - IBM UI Standards Specialist
  - UI Specialist
- Support Team
  - Support Specialist

Figure 4.1: The roles needed in an average SDM software project

Given the large number of roles to fill, if a project were to fill out the roles too soon, then the project burn rate would be prohibitively high. On the other hand, if a project were to fill out the roles too late (which is probably more common, and has been one of our problems), then the project members find themselves not only over burdened with too many tasks, but also having too little time for actual design and development. All too often the project developers find themselves being inundated with non-development tasks (e.g. build-oriented tasks, operation tasks, server issues, testing, documentation), which greatly diminishes the project moral and productivity.

F. The Iron Triangle

As mentioned in Scott Ambler's article, The Iron Triangle [14], in order for a project to succeed, it can fix no more than two of the three critical factors of a project (Time, Features, Resources) without sacrificing quality. Any two factors can be fixed, but the third must remain flexible. Breaking the triangle typically results in a cancelled project, a late or over budget project, a broken project (poor quality) or a reduced project (a reduced feature set). Our experiences match those of Scott's.

Project resources are usually fixed, and the belief that resources are variable often leads to the mistaken notion of a "person month", which was first brought to light in Fredrick Brooks' tale of the IBM OS 360 project, "The Mythical Man Month" [8]. Our experience bears this out – not only are programmers are not created equal (meaning, they are not interchangeable), but also, new people added to a project usually have to undergo significant training or indoctrination before they become self-sufficient and are able to make significant project contributions. In fact, Brooks coined a law "adding manpower to a late software project makes it later." [8],[15] Therefore, resources realistically change slowly, so the practical factors that are candidates for being variable are time (project schedule) and program features.

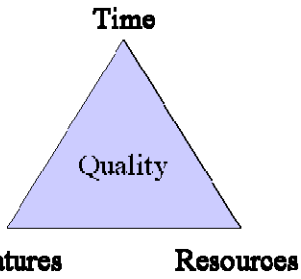

Figure 4.2: Critical project factors

In a plan-based project, the schedule and feature set are both fixed, but the reality is that deadlines are missed and feature sets shrink. In an agile-based project, approaches exist for varying either factor: using time boxing [16] to fix the schedule and vary the feature set, or fixing the feature set and letting the agile development teams vary in size, shape and schedule.

G. Lessons Learned

Agile methods help the team deal address technical challenges, with the help of the customer, in order to deliver the best possible application. Generally, the greater the number of unknowns there are in the project: application requirements, system design, computing platform, development environment, the more essential the use of

Agile Methods for eliminating the unknowns in the project. The customer involvement, the rapid development and the iterative programming are key elements in solidifying requirements and design.

Plan-based methods help the project align with the users, the funding source and the test teams, however, successful pure plan-based projects are rare because plan-based projects work well only when there is little or no variability in the schedule. To our knowledge, the only projects that have little or no variability are the ones that have no unknowns because they have been done before.

We have a separate, stand-alone comment on software development. The use of an integrated development environment such as Eclipse [17], with team support supplied by Jazz [18[, [19], [20] for feature and defect tracking, removed much of the chaos we've felt in previous projects.

V. A Proposal for a Hybrid Method

A comparison of agile methods and plan-based methods might suggest that the two methods are opposites and could not co-exist in the same project. As a result, a project manager would have to pick one method or the other for managing the software development. However, necessity is the mother of invention – we had a need for both approaches and so we found a way to make the two seemingly conflicting development methods work together.

In order to interface with a client who has to manage budgets, schedules and demanding user groups, we must publish a project schedule with specific release dates and fixed feature sets. In order to accomplish effective software development, especially when developing something new, we must employ agile methods. To keep the plan-based schedule and the flexible agile development from interfering with each other, we put them in parallel tracks.

A. Parallel Tracks, Parallel Efforts

In the Hybrid Development Method (HDM), there are two independent tracks: the release track and the prototype track (Figure 5.1). The prototype track comprises pure agile development, with short segments of development of specific features, customer demonstrations and iteration. A feature is considered mature when it has been implemented, tested and verified against a customer-supplied expected results file. The release track comprises the integration and full testing of mature, prototyped features that were completed in the previous cycle of prototype work. At the beginning of a release track cycle, the feature set is known (only mature features are eligible), and since the prototype work is done, the estimation for integration, function testing, user acceptance testing and production release work can be much more exact than when actual development is involved.

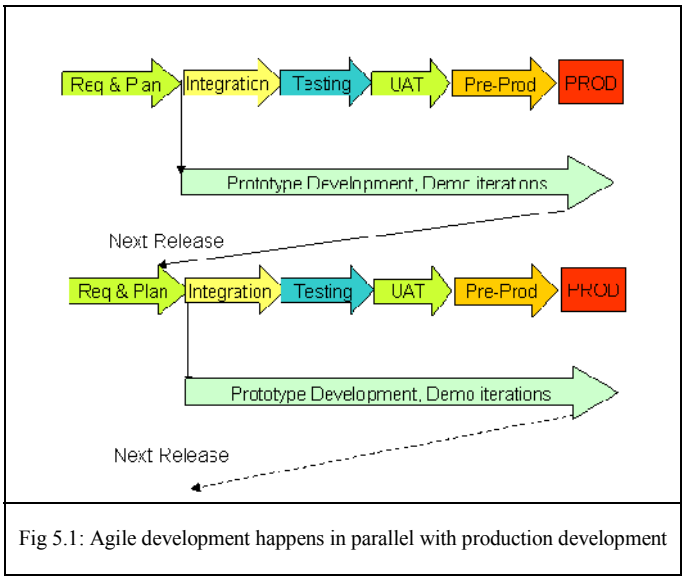

Fig 5.1: Agile development happens in parallel with production development

B. Creating the Foundation Release

Figure 5.1 shows the project tracks in a steady state; the first track – the start of the project – looks different. Figure 5.2 shows the *beginning* phase of a project. The starting phase of the project is called the "Foundation Release", and it is during this phase when all of the high level, major project decisions are made. The Foundation Release includes a combination of agile development and up front design, basically a downsized version of "Big Design Up Front". In our experience, many important and significant project decisions get made implicitly and unofficially when project development starts in prototype mode. Then, when hindsight shows the unfortunate mistakes, the project undergoes major refactoring and reorganization to address the major decisions again. Instead, these decisions should be formally recognized, addressed and documented. That way, the major project issues are formally settled early on, namely the project platform, programming language, overall system architecture, basic data model, user interface design, system logging mechanism, test methodology and infrastructure, support for national language and initial feature set.

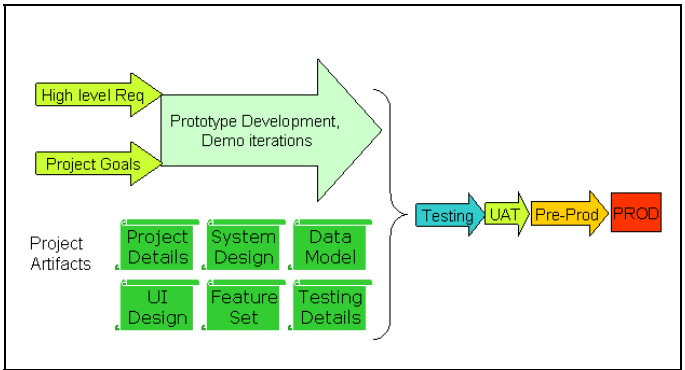

Figure 5.2: The starting phase – the "Foundation Release" of the Hybrid Development Method

A supporting argument for adopting a short plan phase comes from the User Experience (EUX) community. One of the unfortunate side effects of a project that uses agile methods is that all notions of a "Big Design Up Front" approach get pushed aside, but in fact that is a dangerous assumption when all up front User Experience design work is avoided [21]. The overall user experience, design goals and plan for achieving those goals with iterative steps must be in place before the project starts. Our experience has shown that "EUX Design by Iteration" cannot match the quality of "EUX Design by Plan".

The actual feature-based prototype work that takes place in the foundation release must be minimal. The foundation release is not meant to be the first product release for the customer – it is merely the base for the development of the first product release. It is likely that an anxious customer will push to make the foundation release an actual product, but that would impose all of the usual issues that come with fixed resources, fixed features and fixed dates – meaning, there will be early project failure.

C. Team Dynamics and the Two Tracks

The interaction between the team members and the two tracks (release and prototype) should be fluid. A developer may be working in either or both tracks at any one time. In fact it is expected and encouraged that the developer who implements a feature in the prototype track should be the one to perform the integration of that feature in the release track.

D. Final Comments

Our successful experiences using agile methods for developing new software match the claims and projections in the agile literature [2], [5], [22]. It is also clear, however, that the use of agile methods for large teams presents multiple challenges, although that is an acknowledged problem with agile methods and there are proposed solutions for splitting and downsizing teams where appropriate [12], [13], [23], [24]. However, we have not yet found in the literature a model for combining mature production releases, with fixed release dates along side flexible agile development. Our proposed Hybrid model accomplishes that.

The hybrid model gives us multiple advantages. First, it keeps the agile development model alive and well in our project; we know that agile methods are essential for developing new features, as well as experimenting with alternative ideas and designs. Second, although we get the benefits of agile methods, they do not affect the base production schedule. The requirements that are promoted to the production stream are only those requirements that have successfully graduated out of the prototype phase, where there is a well defined description; a known expected result, and a prototype implementation. Finally, and perhaps most important of all, the production development phase can be both relatively short and relatively stable. The development phase becomes more a task of integration than actual development.

Most of our software development projects start with some initial informal investigation before there's a formal project start. There is discussion, there is some proof of concept prototyping, there is some customer interaction, etc. Our notion of a Foundation Release includes the assumption that the development team has a strong idea of what they are doing. When a project is completely up in the air (similar to our "No Requirements" example), then a sizable amount of experimentation, prototyping and proof of concept work happens; it is necessary to validate the idea, the implementation and the project team. The prototype work done in the "pre-project" phase mostly comprises the minimal first feature set in the Foundation Release.

VI. CONCLUSION

With many years of experience in software development, we have experienced many different types of software project success and failure. Our experience shows that the type of project development used has a profound impact on the project; it is often the deciding factor as to its success or failure.

We have presented four archetypal projects, each representing a different position on the spectrum of project requirement definition: from almost no requirements, to very well defined requirements. Looking at the history and behavior of the projects, it appears that plan-based project schedules are inherently unrealistic, mainly because they fail to take into account the variables that occur in software projects, even those projects that appear to have zero or minimal unknowns. Project variables of any significant size or number tend to cause resets and date changes in a plan-based development project. Each change to the schedule diminishes the reputation of the program manager(s) and the development team and also damages team morale.

Unlike plan-based projects, which tend to start slow, with little results, agile-based projects start with a bang. They produce prototypes almost immediately, and win praise from customers. Agile-based projects appear to have everything figured out. However, the problems with agile appear once the project has reached a level of maturity. They tend to have a weak relationship with the fixed characteristics of the real world, and thus do not mesh well when a customer needs to know in advance exactly which features they are getting for their money

It is our thesis that neither pure agile-based projects nor pure plan-based projects are going to succeed for the majority of projects. The effects of reality are simply too powerful, thus imposing severe restrictions on those idealistic models. We propose a hybrid development method for managing larger scale projects that are expected to live for many releases. The hybrid method employs agile for all new development work. That is the best way to create new function in a timely and productive manner. However, the agile development work is kept in a separate track so that slipups and delays on features do not affect the product release dates. Each release track starts with a known set of features that will be integrated for that release, and comprises some integration work, a minor amount of development

work, and mostly testing work. This combination of plan-based methods and agile methods is, so far, the only approach that will maintain sanity in both the development groups and the customer (funding and user) groups.

ACKNOWLEDGMENT

Thanks to the entire SDM team for all of their hard work on the project.

REFERENCES

[1] "Waterfall model,", Wikipedia, http://en.wikipedia.org/wiki/Waterfall_model

[2] "Agile Software Development", Wikipedia, http://en.wikipedia.org/wiki/Agile_software_development

[3] K. Beck et al, "Manifesto for agile software development," Agile Alliance, 2001, http://agilemanifesto.org/

[4] K. Schwaber, M. Beedle, "Agile software development with Scrum," Prentice Hall, 2001

[5] "Iterative and Incremental Development," Wikipedia, http://en.wikipedia.org/wiki/Iterative_and_incremental_development

[6] B. Boehm, "Get ready for agile methods, with care," Computer, Vol 35, pp. 64-69, 2002

[7] Wikipedia, "Software Development Death March", http://en.wikipedia.org/wiki/Death_march_%28software_development%29

[8] F. P. Brooks, "The mythical man-month: essays on software engineering," Addison-Wesley, 1975.

[9] J. Rising, "Agile software devleopment process, managed mayhem,," May 6, 2009, http://www.managedmayhem.com/2009/05/06/agile-software-development-process/

[10] "Big Design Up Front", a collection of posts by various authors, http://c2.com/xp/BigDesignUpFront.html, 2006

[11] "Software testing," Wikipedia http://en.wikipedia.org/wiki/Software_testing

[12] S. Ambler, Roles on Agile Teams: From Small to Large Teams http://www.ambysoft.com/essays/agileRoles.html

[13] S. Ambler, Scott, "Supersize Me," Dr. Dobbs, March 1, 2006 http://www.drdobbs.com/184415491;jsessionid=HF2LQPI14DGBHQE1GHPSKH4ATMY32JVN?queryText=supersize+me

[14] S. Ambler, "The "broken iron triangle" software development anti-pattern, 2006 http://www.ambysoft.com/essays/brokenTriangle.html

[15] Wikipedia, "Brook's Law", http://en.wikipedia.org/wiki/Brooks%27s_law

[16] Wikipedia, "Timeboxing," http://en.wikipedia.org/wiki/Time_boxing

[17] Eclipse Software Development Home Page, http://www.eclipse.org/

[18] IBM Rational Red Paper, Bruce Powel Douglass, Mats Gothe, IBM Rational Workbench for Systems and Software Engineering

[19] Jazz Home Page, http://jazz.net/

[20] IBM, "Collaborative software development. A new approach: open commercial software development and jazz," http://www-01.ibm.com/software/info/features/collaboration/main.html

[21] S. Ambler, Introduction to Agile Usability, User Experience Activities on Agile Development Projects, Agile Modeling, 2010, http://www.agilemodeling.com/essays/agileUsability.htm

[22] H. Smits, "5 Levels of Agile Planning: From Enterprise Product Vision to Team Stand-up", Rally Software Development Corporation Whitepaper, 2006

[23] J. Rasmusson, J. McDonald, "Canadian Workshop on Scaling XP/Agile Methods," March 2003, http://martinfowler.com/articles/canScaling.html

[24] J. Eckstein, "Extreme Programming and Agile Methods — XP/Agile Universe 2002 Lecture Notes in Computer Science, 2002, Volume 2418/2002, 325-345, DOI: 10.1007/3-540-45672-4_44

[25] P. Kruchten, "Scaling down large projects to meet the agile sweet spot," University of British Columbia http://www.ibm.com/developerworks/rational/library/content/RationalEdge/aug04/5558.html