Agile Development: Overcoming a Short-Term Focus in Implementing Best Practices

Karthik Dinakar

School of Information Systems & Management, Carnegie Mellon University
923 S Aiken Avenue
Pittsburgh PA 15232
Phone: +001 781-354-5564
karthikdinakar@cmu.edu

Abstract

Agile development deemphasizes long-term planning in favor of short-term adaptiveness. This is a strength in a rapidly changing development environment. However, this short-term focus creates a temptation to neglect best practices that are essential to long-term success. This report outlines my experience as a software developer in a leading internet portal that thrives on agile development using SCRUM. It describes the problems that arose when best practices were ignored and how our team overcame them.

Categories and Subject Descriptors D.2.9 [Management]: life cycle, productivity, programming teams, time estimation, software psychology.

General Terms Management

Keywords Agile development, SCRUM, sprint planning, agile infrastructure, agile teams.

1. Introduction

That agile development has an affinity towards the short term, with minimal planning is well known. But this does not mean that planning is antagonistic to agile development. When short-term goals limit the adoption of agile best practices that prove to be highly beneficial to the development of the product in the long run, it causes problems in the team that are cumulative and often irreversible. By using the phrase short-term goals, I refer to the pressures that managers and developers face from the top management with respect to tight release deadlines and overlapping developmental cycles that leave little room for the adoption of necessary best practices.

Often, a particular best practice which can reap rich benefits in the long run is delayed either because of the immediacy of more pressing issues or because of the tendency to not look beyond the short-term. In most cases, the reason given is the former while the real reason is the latter. Agile teams that don't adapt according to their own post-release recommendations will eventually have to deal with the cumulative effects of postponing the adoption of each of those best practices.

If the developers in an agile project intend to incorporate a set of best practices that they deem necessary, but are constrained from limitations and expectations set by the upper management, there are significant interrelated implications for team morale [CM00] and the quality of the product itself.

To avoid this quagmire, it is essential that the stakeholders in an agile-driven project understand the importance of implementing best practices and recommendations that surface after a post-release or an interim review. In particular, teams embarking on a new product should invest heavily in the early adoption of agile best practices, given the constraints and limitations that they are likely to encounter if they chose to adopt them at some point in the future.

1.1 Background: Agile Principles

Agile development relies on the collaborative efforts of everyone involved in the development of the product. Working software is underlined as the most tangible yardstick of the state of the product. Agile teams are self-directed and continuously improve the team's processes based on a cycle of self-feedback.

Each agile team is expected to tailor its own processes and informal rules, and improve them iteratively after collective retrospection.

Based on the original twelve principles of the agile manifesto [AM01], I now discuss important lessons that are relevant to problems that I discuss later.

Changing requirements are an integral part of agility. Developers and testers should be cognizant of the fact that requirements can be added, removed or modified even near the end of a release cycle. Agility implies being able to respond swiftly and effectively to changing requirements.

To *deliver working software frequently*, each team conforms to a set of coding and code-review standards that ensure the robustness of the code lying in the version control system at any given time.

OOPSLA 2009, October 25-29, 2009, Orlando, FL, USA.

Developers form the core of an agile team. Without them, the product will never reach fruition. To give each developer a sense of ownership, and for product managers to understand the technical limitations of a feature, it is essential that *business people and developers work together throughout the project.* Program and product managers are often so tied to the business side of producing software that they sometimes cannot comprehend why a particular product feature needs more time for development than they would like. Developers must be involved during discussions about requirements. The rationale behind every requirement needs to be explained to them, and they must be free to suggest better alternatives. Once the developer is aware of why a certain requirement is needed, he is in a better position to anticipate the scope of the requirement. He can then design and write his code keeping in mind the larger picture of the product and the direction that is headed towards. I describe the positive effects of involving developers in product backlog discussions in section two.

At the heart of the concept of agility is *early and continuous delivery* of software. Minimal agile infrastructure, such as an automated build system that gels with version control is a must if there ought to be clean, early and continuous delivery of software to customers. An automated build system not only frees developers from making the builds manually, it aids in test-driven development, ensuring release-quality code at every major point in the development cycle. Continuous delivery of software implies the ability to have *releasable* code at every point during the development cycle. I discuss the essentiality of fundamental agile infrastructure in section three.

Product and program managers must strive to give developers the *environment* conducive for healthy development. Micromanaging of developer task estimation is indicative of a lack of trust. People doing the actual fine grain tasks must be free to estimate them in a rationalized manner, and this is particularly important for developers. When the time required to code a particular feature that is assigned to a developer is estimated by the engineering manager, it says 'I do not think you know what this task is all about. It'll take you x hours to complete it'. Developers can estimate their own tasks better, and must be encouraged to be good estimators of the features that they implement. In sections four and five, I discuss how sprint planning can reach maturity and become very effective when developers and QA are empowered to estimate their own tasks.

Communication between UI, development and QA teams are best done when they are *face-to-face*. Developers ought to understand that *working software* is the only real measure of progress in an agile-driven project.

Decision discussions and meetings must have a place for developers. A *good design enhances agility* and the developer ought to involved in this process from the word go. If a developer designs and writes code in isolation, oblivious to the larger architectural goals of the product, subsequent development cycles are likely to incur a lot of code refactoring, which can be avoided. Hence it is a good practice to involve the developer in design discussions. Work overload and team work-load distribution are managerial issues that can either have a positive or negative effect on team morale and cohesion. In section six, I describe the negative effects of sprint overloads and skewed work-load distributions on team morale and how this can be overcome.

It is also essential that the team *reflects* on how to become more effective and then tune and *adjust its behavior accordingly*. This is a vital aspect of agile development. Action based on self-feedback and retrospection is essential in ensuring that the development process and the people involved in it work synergistically to produce high quality software. This allows an agile team to concentrate on the technical and process related problems of the current release cycle that was just completed. What were issues confronting the team and what can be done to avoid them in the future? Were code reviews effective in preventing regression bugs? Why were there so many code integration problems? Why were tasks overshooting their allotted time and extending beyond a sprint cycle? Answers to such questions are very helpful – they help teams to improve iteratively after each release cycle.

Lessons drawn from the original principles of the agile manifesto give us a framework to examine the dilemmas and challenges of overcoming near-sightedness and the value gained by early induction of agile best-practices.

1.2 Project overview

I started my work as an intern in a new, startup web platform that was to allow the rapid designing and publishing of websites with minimal human intervention. The platform was to be used across the company for many of its existing properties, and utilized a lot of web-services that the company offered.

Since this was a new product which was to be created by a newly formed team, the bare essentials of agile software development infrastructure had to be established. Each team in the company was free to choose its own suite of developmental tools and define their own standards in the spirit of agility.

Fundamental software development infrastructure such as source control, coding standards, processes for code reviews and check-ins, informal rules for design discussions and team meetings were drawn out before the start of the project. Key infrastructure elements which bear special significance for agile development, particularly in the web domain, such as an automated build system, a unit test framework and automated QA sanity checks were not a part of the initial development infrastructure. Instead, it was decided that these were to be implemented in small increments alongside the development work of the platform.

As development of the platform got underway, these key infrastructure elements were not given much of a priority. In fact, they were not implemented even after the third release of the platform. The repercussions were felt heavily by developers. Task estimates began slipping and the number of regression bugs was on the rise. Repeated appeals by developers for adoption of these key elements were de-prioritized by the upper management who faced their own pressures for newer and faster releases of the product. Faced with decreasing confidence in the product that they were coding and steadily declining team morale, the developers reached a tipping point and decided that they would work overtime to implement these key elements.

A new automated build system was built using *CruiseControl*, and a unit testing framework was adopted. Adherence to coding guidelines was made a part of the automated test-case framework. Sanity tests were automated and performed at frequent intervals, thereby saving valuable time for both the development and QA teams. A case was made to the engineering and product managers to align the process of task estimation more closely with the principles of agile development. Guidelines were drawn for the length and scope of sprint meetings, and developers sought a greater involvement in design discussions and product backlog meetings. In brief, the team introduced a set of agile best practices into the building of the product.

The benefits of implementing these key changes were gradual but highly significant. Design discussions were increasingly more fruitful. Task estimations were more accurate and sprint task spillages – the moving of incomplete tasks to the next sprint cycle, were on the decline. More importantly, there was a marked decrease in the number of regression bugs, and the product was increasingly more stable. The team morale steadily improved and subsequent releases were of higher quality.

The gist of these experiences is that managers and developers of agile teams must recognize the long term benefits of adopting agile best practices. The early adoption of fundamental agile infrastructure, preferably before the developmental work gets underway, is crucial.

2. Product Backlog: Involving Developers in Decision Making

Defined in simple terms, a product backlog is a list of to-do things. They normally contain both the functional and non-functional requirements of a product. Each feature or requirement is prioritized with a view to adding value for the customer. Features that have a higher priority are listed first and described in greater detail. The product backlog forms the basis for determining the list of tasks to be performed during each sprint cycle.

In this section, I discuss the role of the product manager and his responsibility to involve developers and QA in the discussion and prioritization of the product backlog. I argue that if not allowed to participate in the building and prioritization of the product backlog, developers and QA are likely to view changing requirements less favorably.

Since changing requirements is one of the cornerstones of agile development, an agile team must position itself to supporting this vital capacity of being able to respond to changing requirements. The product backlog is an important agile artifact from this principle of agility.

2.1 Role of the product manager in fostering developer involvement

During the initial releases of the platform, the product backlog was written by a product manager in close consultation with the team's engineering manager. Prior to the injection of best-practices mentioned above, the task of formulating and refining the product backlog was an exercise done in isolation by a couple of people, to the total exclusion of the developers. The product and the engineering managers hardly consulted or involved developers during the prioritization of features. No rationale was given to developers regarding the need or urgency of a feature.

As a result, developers were put in a position where they had to code a feature because they were simply asked to. If a particular feature of the platform was urgently needed, the reason behind the urgency was not conveyed to developers. This created a perception that the act of formulating the product backlog was somehow the sole privilege of a select few and that developers were simply viewed as 'resources' that should be told what they ought to be doing.

There was a definite lack of ownership on the part of developers, which is in direct contradiction to the agile principle of building software collaboratively. This lack of ownership meant that developers were coding features without fully understanding them. It was not very clear why certain features were regarded as more important than others. As a direct result, designs of the code changed in almost every release cycle. The code was refactored in almost every release cycle. Most developers in the team considered code refactoring to be a dull and boring activity. This further added to the decrease in team morale.

Developers need to understand what they are building and why they are building a product. They should be able to say 'I was involved in the discussion of this feature. This feature is my responsibility. I own this feature. I am in charge of it'.

2.1.2 Product backlog prioritization

Another bane of the lack of visibility of the product backlog was the effect that it had on QA. The QA team was simply asked to test a feature without understanding the relative importance and customer value that a feature carried. This resulted in QA testing all the features with roughly the same priority, until they were explicitly told much later that some features were critical and needed exhaustive testing. Exhaustive testing conducted at a later point unearthed significant regression bugs that could have been caught earlier.

As a part of best-practices adoption, a case was made to involve developers and QA in the explication and prioritization of the product backlog. Meetings to discuss product requirements were initiated and an effort was made to involve both the development and QA teams in the discussions.

Developers now had a much better idea on why a particular feature was deemed important. QA could now prioritize the strength of their tests for a particular feature; critical bugs were few and caught early. There was clarity across everyone involved in building the platform as to what was important and why it was important, which underscores the importance of close collaboration across agile teams for producing high-quality software.

The lesson drawn from this experience is that product managers should be very aware that whilst they are in charge of leading the product towards a certain goal, it becomes critically important to communicate the rationale behind a feature with developers and QA folks involved in building a product.

2.2 Agility of the product backlog

There were a number of instances during my work in building the platform where the product backlog was readjusted according to new and changing customer requirements. Although most customers of the platform were internal properties within the company, requests to change or add a new feature were made both during and after sprint cycles.

These changes and new features were viewed less favorably by developers, who were distraught when they were told that a feature completed by them either had to be changed or a new one had to be coded in its place. Developers felt that they had no control over the development process.

Even the QA team was resistant to these changing requirements, and was reluctant to re-adjust their schedules and test-plans to accommodate the change. The incorporation of the change or the feature was done grudgingly. This meant that the code written or modified was done half-heartedly and within a very short period of time, with the immediate result of less than satisfying and buggy code.

This issue was discussed at length during the adoption of best-practices and it was decided that everyone involved in agile development, including developers and QA, needed to welcome changing requirements, even if they crop up at the last moment. It also ties well with the earlier point that developers need to be involved in the product-backlog prioritization. Being able to produce quality software in the face of changing requirements is one of the cornerstones of agility.

This change in perspective on the part of developers and QA was gradual, but a very positive one. Features that were added during and after sprint cycles were not met with resistance. Having participated well in the prioritization of the product backlog, developers readjusted their task estimations and QA readjusted their test-plans. The focus was now on the feature asked by a customer.

This kind of customer-fixation is essential in agile development. The lesson drawn here is that the product backlog, as with any agile artifact, must be in sync with what this fundamental principle of agility espouses.

3. Essentiality of Fundamental Agile Infrastructure

An agile approach to developing software requires a basic, minimal set of development tools and processes. A wiki to share information seamlessly between various stakeholders, choosing an appropriate version control, a unit testing framework and automated build and deploy systems are some of the bare essentials that are needed for agile development [SH05].

In this section, I discuss why unit testing frameworks and automated build systems are important in agile development.

3.1 Adherence to coding guidelines and unit testing

Prior to the adoption of best practices, the team had a coding standard that had to be followed. It included naming conventions and coding styles for functions, classes and guidelines for writing code comments. These coding standards were more or less followed for the first few releases. The only checks for these coding standards were code reviews. Given that most code reviews were one-on-one meetings between two developers, adherence to coding standards decreased over time. In fact, the same person typically reviewed a developer's code most of the time, with the result that other developers had a difficult time in getting used to a coding style that was different from the type that they were accustomed to.

Unit testing was a highly individualized effort without a formal unit testing framework. As the platform increased in its complexity, the lack of a unit testing framework meant that the number of bugs caught by QA was increasing and the bug-fixing cycles were longer and more frustrating. The number of bugs unearthed by QA had a significant impact on team morale, as I shall indicate later.

During the team's discussions on adopting best practices, it was decided that a unit testing framework ought to be implemented as soon as possible. *PHPUnit* was chosen for the frontend code and *JUnit* was adopted for the backend. Coding standards were strengthened and formalized as a part of the unit tests that each piece of code had to have. Two coding standards checking tools that had already been developed within the company were used. It was simply astounding that there were so many tools that were readily available and yet the team had not thought of using them.

The benefits of these changes were highly positive and immediate. The next release of the platform produced lower numbers of regression bugs. Most of the newly written code had a uniform coding style, irrespective of the developer who coded it. Code reviews could now be done by any developer in the team. Integrating my code with that of another developer was now far easier. This resulted in smoother integration cycles and fewer integration related bugs.

3.2 Build Automation and Automated Sanity Tests

During the first five release cycles, a build was performed after every sprint and after every bug fixing cycle. This process was manual and was performed by a developer. This added responsibility of build engineering was assigned to each developer on a round-robin basis.

The complexity of the platform was such that there were a number of frontend and backend packages, most of which were required to be deployed on seven servers according to a deployment matrix. This meant an entire day was consumed in making a build. The developer making the build had to accommodate this additional role without affecting his own development estimates for a sprint cycle. The build process was viewed with scorn and disliked by all the developers who saw it as a laborious process that was highly manual in nature.

After making the build, the build engineer of the week had to perform certain mandatory sanity tests to validate the deployment of the build, which was another long and manual process. Websites had to be published on all of the frontend servers, making use of each of the backend servers at a time. It was a task that was almost dreaded by all of the developers.

Repeated pleas for having an automated build system was largely ignored by the management. Some of the developers felt that these pleas were nothing more than 'soft-pedaling' and suggested more aggressive petitioning. Others felt that a stage would eventually be reached when the management would see the light of day and realize that a build automation system was badly needed.

As part of adopting agile best practices, all the developers agreed that there was an acute need for automating the build process. Given the increasing scope of the platform and the number of other teams within the company that were going to adopt it, the team decided to automate sanity testing and make it part of the build process. CruiseControl was adopted as the build tool. Coding standards checkers and the sanity testing suite were made a part of the build process too.

Within a few sprints, nightly builds were started. Each of the initial nightly builds proved to be very painful, in the sense that many of them failed because of a coding standard failure or a unit test execution failure. But the team resolved itself to continuing the process of making nightly builds. There was positive reinforcement that however painful this exercise might be in the short term, nightly builds were a must. This had a dramatic impact on the stability and quality of subsequent builds.

Automating the build and sanity testing processes freed up valuable developer time. The QA team was so thrilled to see a developer-driven sanity testing automation framework that they embarked on an automation tool for their set of basic test cases.

In other words, adopting one best practice inspired the adoption of another.

3.3 Stability of the code and its near-release quality

Persisting with nightly builds, even as it unearthed painful habits of developers, was a bold and necessary step that proved to be highly beneficial over the long run. Regression bugs plummeted in number and there was a renewed sense of confidence in the quality of code written by each developer. More importantly, the product was now of a near-release quality after each of the nightly builds.

The lesson drawn from the above experiences is that the set of fundamental agile infrastructure must include an automated build system. It should also automate sanity testing if possible, and integrate coding-standard checkers and unit tests within the build process. It is not enough to merely choose a version control system and define loose processes for adherence to coding standards and unit testing. Automating these steps within a tool such a CruiseControl would go a long way in ensuring that the product that emerges after every build is stable and is releasable.

Having such an automated system is an important facet of continuous integration. It encourages developers to check-in code that has been thoroughly tested. Test driven development of software is critical to an agile team, because it ensures release-quality code at every point in the development cycle.

This set of fundamental agile infrastructure is necessary if an agile team is to position itself towards responding effectively to change, be it changes in requirements, in the scope of the product or changes in the team composition.

4. Sprint Planning and its maturity

A team that uses *scrum* for software development uses sprints – usually a two to three week period producing an increment of the product in terms of new or changed functionality [Ksh04].

In my team, there were usually three sprints for every release of the product with each sprint lasting for about two and a half weeks. Items to be finished during a sprint were entered into a sprint backlog, which was closely tied with the product backlog. Tasks were assigned to developers based on their core competencies. Each developer was required to break each of his tasks into subtasks. Each subtask had to be assigned an 'effort', in terms of the number of hours it would take to complete the task.

4.1 The estimator

The usual unwritten norm in the organization was that the act of estimating a task was to be done by the developer alone. Of course, the engineering manager had a prerogative to temper any estimation, and tell the developer if certain estimations seemed too optimistic or too conservative. But the primary responsibility of estimating a task was on the developer – by estimating a task the developer was committing to delivering that functionality within that time.

Prior to the incorporation of best practices, the process of estimating a task was more of a top-down chain of command. Excessive pressures on the engineering manager from his director often forced him to estimate the tasks for many of the developers. This was against the practices of other teams in the company. The platform that my team was building was a very important one, and many other teams relied on it. This was often cited as a justification as to why micromanaging of task estimations was happening in the team. Of course, the developers did not like their tasks to be estimated by another person, although they grudgingly accepted the fact that the engineering manager was probably under a lot of pressure himself.

Repeated attempts by the development team to convince the engineering manager and the division director to let them control their own estimations were often drowned down by emphasizing the urgency of the business the need. 'We need to release these features as soon as possible. Many teams depend on our platform. This is agility. We have to focus on what is important here' was the usual response. Obviously, the suggested best practices were not viewed as very important.

During the brainstorming sessions about the incorporation of agile best practices, the developers sat down with the engineering, product and program managers to convince them that there was more benefit in allowing developers to estimate their tasks rather than doing it for them. Specific examples of sprint spillages were cited as proof.

After some persuasive discussions, it was decided that the task of estimating a task was the primary responsibility of the

developer. Each developer was now expected to view their estimations very seriously and responsibly, and that finishing a task implied that the code would be written thoughtfully and tested thoroughly within the estimated time.

This was in sync with the fundamental principles of agility. Trust is an important attribute of teamwork for an agile team. And by allowing each developer to estimate their own tasks, he or she was being trusted to deliver that task within the time promised.

Suddenly, the seriousness with which each developer approached the process of estimating a task was evident for everyone to see. Each subtask was estimated right down to the act of code reviews and code check-ins and unit testing. Subsequent sprints had lesser numbers of spillages, as most of the tasks were completed within or before the estimated time. This had a significant impact on the project schedule, which was now a lot smoother and a lot more predictable.

4.2 Granularity of task estimation

It is important to highlight a detailed approach to task estimation.

High level estimations such as "9 hours for Flickr integration to the new backend framework" were weeded out as part of the new changes within the team. Each subtask had to have estimations for every step in the process. For example, the following table gives sample estimation for an important task.

Task: Flickr Integration to the new backend architecture

Subtask	Estimation(hours)
Generic XML definition	2
Integration with moderation	2
Integration with cache	3
Unit test cases	2
Unit testing	2
Frontend testing from a published website	2

With the granularity of task estimations increasing, developers had a much better idea of the how long it might take to code a given feature and allow them to perform comprehensive testing before signing off on that particular task.

4.3 Planning for early integrations in a sprint

Another important change was to formally incorporate code integration within a sprint. Before this was achieved, the code from two or more developers was integrated just before the sprint was about to end. Integration of code from two different developers was not a formal part of the task estimations. Hence many code integrations stretched well beyond a sprint cycle and affected the sprint backlogs of subsequent sprints.

Planning and formalizing code integrations, including devoting time for comprehensive integration testing, paid rich dividends. Code integrations were now well achieved within each sprint cycle and the number of integration related bugs dropped drastically. Integration was now a more peaceful process and not performed under the duress of doing it beyond a sprint cycle.

4.4 Efficacy of online sprint tools

Sprints within the company were tracked using a sprint manager. Each developer had to put up his estimations on the sprint tool. As and when things were done, the sprint tool had to be updated to reflect the work done. These estimations were visible to everyone in the team, and to that extent, provided a good visibility of what was happening to other stakeholders and the upper management.

While providing visibility, the sprint tool often forced developers to think about how their tasks were viewed and perceived by others. The focus was now on what the sprint tool showed, rather than on the task itself.

Hence it was decided that while the tool was good to the extent that it allowed each developer to track and organize his or her task, the focus should be on the task itself and not so much on what the sprint tool showed.

It was agreed that the sprint tool was only a tool; working software was more important than what the sprint tool showed. Each developer was encouraged to use the sprint tool to refine his estimations and track his or her progress in a release cycle.

The lesson here is that these sprint tracking tools should not be treated as 'contracts' between the engineering manager and the developers. They are not agile artifacts, and are more of an aid in better planning and organizing.

5. Issues in Sprint Cycles

Each release of the platform usually had three sprints, with each sprint lasting for about two and a half weeks. Prior to each sprint, there would be a sprint planning session where the program manager would sit with the development and QA teams to form the sprint catalog of features and tasks to be done.

The importance of sprint planning can hardly be emphasized. In those cases where the sprint planning was poor or not comprehensive enough, there were invariably a considerable proportion of spillages beyond that particular sprint cycle. Sometimes, the sprint planning sessions would be held on the day when the actual sprint cycle was due to begin. This meant that the sprint was already delayed by a day, because it would take at least a few hours for the individual task estimations to happen before the actual work could get underway.

5.1 Sprint planning and when it should be done

As part of the agile best practices adoption, it was agreed that sprint planning sessions would be held in advance of the actual sprint cycle. It was after all, a planning exercise and had to be performed before the commencement of the sprint cycle itself.

It was decided that the sprint planning exercise would be conducted two days prior to the start of the sprint. This idea was initially opposed by the top management which felt that two days for planning sprints was too much and that agile development, by nature did not approve of such a degree of planning. They felt that this was encroaching on valuable development time and that it would add an unnecessary delay to the overall project schedule.

The development team though, argued that the two days for sprint planning was both reasonable and necessary. Cases where improper sprint planning resulted in spillages were referred to. It was argued that sprint planning would be a comprehensive exercise with detailed and careful attention being given to task estimations and QA build schedules. Investing heavily in sprint planning meant that sprints were likely to have less spillages and have a stabilizing affect on the entire project schedule. With great reluctance, the engineering manager and the division director agreed to allow this for the next release of the platform on an experimental basis.

The experiment reaped rich rewards though. Allowing developers to deliberate on their tasks and calibrate their task estimations, with a view to incorporate code-integrations and post-build sanity testing brought a sense of order to each of the sprint cycles. The program manager and the QA team also felt that their schedules were now less variable. There was a feeling that things were proceeding in a predictable and systematic fashion. After noticing that the number of spillages in each sprint cycle was just one or two if any, the upper management was happy to note that two days of aggressive and comprehensive sprint planning was a good thing to have done. Other teams in the company were now embracing this idea too.

Sprint planning, therefore, is a must if spillages are to be minimized or eliminated. Contrary to the notion that agile development proceeds with minimal planning, this proved that careful and detailed planning is not antagonistic to agile development. In fact, planning complements agility.

5.2 Stand-up meetings and the 15 minute rule

Scrum uses daily stand-up meetings during a sprint cycle, where each developer talks about how much work he or she has completed and what would be next on his plate, and what bottlenecks or dependencies might prevent them completing his tasks. All the developers are usually present in the meeting and it is conducted by the program manager (sometimes called the scrum master). In my team, the engineering manager was also present in these meetings and his comments often carried more weight than the program manager himself.

A daily scrum meeting usually lasts for about fifteen minutes. Given the number of design discussions and other meetings that a developer was usually involved in, it was problematic if daily scrum meetings stretched beyond half an hour. This was common in the team. Scrum meetings often extended beyond half an hour.

Several blogs and research papers advocate the removal of chairs in a room meant for scrum stand-up meetings [Yip06]. Stand-up meetings are supposed to be short, precise and concise, where the focus is only on the sprint tasks. The program manager agreed that this ought to be done immediately and that each scrum meeting would last only for about fifteen to twenty minutes.

The benefit of incorporating this concept of 'stand-up' meetings was that subsequent meetings were to-the-point and saved valuable development time for each developer. Scrum meetings that digress into design discussions and requirements analysis waste a lot of development time and indicates a lack of initial planning.

A role of a *digression detector* was assigned to a developer in each of the scrum meetings. This individual was to call out digressions during the sprint meetings. The main idea was to avoid sprint meetings from digressing into design discussions. Scrum meetings are about the tasks that are currently underway and the ones that would follow next. If scrum meeting end up being design discussions, it means that development work has started without ironing out a strong design strategy.

5.3 Necessity of prioritization of bugs after every sprint

For the first few releases, the prioritization of bugs was done only after the completion of all the three sprints and just before the bug fixing cycle. This meant that bugs which accumulated over each of the sprints were put in the same basket. Critical bugs from all three sprints were fixed only during the bug fixing cycle. Bugs that emerged from the first sprint were the ones that usually caused many of the bugs in the second and third sprint.

This process was changed to prioritize bugs after each sprint. Bugs that were prioritized as potential regression causers were to be fixed by the developer who caused it in the next sprint. This was a healthy change, as the number of regression bugs in the bug fixing cycle at the end of the three sprints decreased and the cascading effect of the earlier bugs was avoided. Each developer was now keener to avoid fixing his own bugs during a sprint and was hence more careful about checking in any code before testing it thoroughly.

5.4 Sprint cycles in the vicinity of a release

The platform was to be deployed on seven production servers. All production related tasks were performed by a dedicated operations team assigned specifically for the platform.

Planning for production related tasks such as *re-brooming* production boxes and acquiring new hardware was done only during the vicinity of the staging release.

A developer from the team would wear the hat of a release engineer and sit with the operations team to work towards acquiring the hardware and establishing the supporting environments on each of the production boxes. This was a side dish of a sort for the developer, who had to do this in addition to his or her coding tasks in the sprint. The result was that the production boxes were set up just in time, all done in a hurried and stressed fashion. Repeated calls for establishing a structured process to deal with the operations team were overlooked by the upper management.

The developers, as part of the retrospective changes, decided to form a separate process to plan and deal with the operations team well before the start of the first sprint. While this process was not very formal, early preparation and active involvement of the operations team from the beginning meant that the staging and production pushes were smooth and orderly. Many production related issues, such as obtaining permissions for deployment of packages on important production servers was

now a planned exercise. The upper management was pleasantly surprised with the systematic way in which the staging and production pushes happened for that particular release and the process was formalized for subsequent releases. Early involvement of operations personnel and planning of production related tasks are vital to ensure that the release engineering process is in sync with the agile development process as a whole – being able to cope with changes in the developmental environment such as an early staging deployment.

6. Managerial Issues and Their Ramifications On Developers

Much of the literature written about agile development often espouses the concept of 'self-directed' teams [Oxy03] where the engineering manager has a minimal role of drawing up schedules and overseeing status reports. As such, many agile evangelists believe that managers should not pursue an active role in designing, analyzing, coding and other direct tasks related to agile development. What then, are the implications of heavy and intrusive managerial involvement in these direct tasks of agile development? As discussed below, managerial issues have a significant impact on the work habits of developers which in turn have ramifications for the entire product. Although there were many instances of such micromanagement in my experience, I list three issues as examples.

6.1 Sprint overload is directly proportional to the number of painful regression bugs

During the first four releases of the platform, task estimations for developers were done by the engineering manager. While this was seen as unnecessary micromanagement, another direct implication was the often heavy loads placed on individual developers for each sprint cycle.
Developers were asked to put in extra hours of work on a routine basis, and sometimes it required an entire weekend to finish the work just in time before a weekly build on Monday.

Frequent multitasking to perform other tasks such as making builds and performing manual sanity testing added to the sense of work overload. Naturally, the code that was produced during such heavily loaded sprints was susceptible to regression bugs. In fact, the number of regressions bugs was distinctly higher during such stressful sprints.

After the team's collective retrospective act of driving in agile best practices, micromanagement of task estimations ceased. Although managerial involvement in this regard continued to be fairly high by most agile evangelist's standards, avoiding sprint overloads and allowing each developer to estimate his or her own tasks did lead to a drastic reduction in the number of regression bugs.

6.2 Improper load balancing during sprints and team fractionalization

Certain developers in the team were given the additional responsibility of build and release engineering alongside their usual development work.

While this is expected of a small agile team, it has to be recognized that the individual sprints of such developers should feature these additional tasks.

Also, frequent context-switching between coding, doing builds and resolving production-related issues often results in reducing the efficacy of each of those tasks. Before the implementation of the best practices, the developer who was to make the builds and perform sanity testing was not allowed to enter these tasks in his sprints. Sprints were viewed as comprising of strictly developmental work. Any other act during these sprints, such as a design formulation or making builds was deemed as a non-sprint task. This approach was largely responsible for the sense of overload that the development team felt.

Consequently after retrospection, such tasks were allowed to be entered as individual tasks in sprint cycles. The idea that a developer's sprint catalog was to feature only coding tasks was disbanded. This change allowed for more relaxed and detail oriented approach to performing builds.

Another particularly unhealthy aspect of disproportionate distribution of workloads in the team was the strife that it produced. Developers who had more tasks often resented the fact that they were being asked to perform more tasks than others in the team. The developers, in retrospect, collectively decided that this was highly unhealthy for the team as a whole, and that the upper management ought to be persuaded to avoid such skewed work-load distribution. Fortunately, the persuasion worked and a clear directive was laid to equally distribute workloads amongst all the developers in the team.

The lesson here is that excessive micromanagement coupled with skewed work-load distribution will pose significant risks for team morale.

6.3 Design discussions devoid the developer

Engineering managers and architects should be very aware that agility as a developmental methodology calls for building a product collaboratively. This means that developers must be involved in design discussions. Designs drawn by together by the architect and the engineering manager in isolation cannot be imposed upon the developers without making them understand the rationale behind the adoption of the particular design. It is important to get their inputs as they know the code best.

After the incorporations of the best practices, developers were invited for all the design discussions. This was highly beneficial to the extent that it was easier to code a certain feature by constantly thinking about the architecture of the platform as a whole, and about any extensions or changes that might come later. Developers were able to choose appropriate design patterns while coding, and there was a sense of perceiving the code in terms of the larger architectural goals of the platform.

7. Conclusion

In this paper, I have examined the benefits of adopting agile best practices and overcoming the short-term limitations that often prevent the adoption from happening. I have compared the state of the development process before and after the adoption of best practices. While each agile team has to adopt what best practices they deem as necessary, I have learned from my experience in this company that there is never a bad time to incorporate an agile best practice. It has led me to believe that early adoption of

best practices is a must in a newly formed agile team, because of the many short term limitations that prevent such adoptions from happening in later stages of the project.

When I left this company to pursue my graduate studies, the team had benefited enormously from the adoption of best practices and many other teams within the company were following suit with their own adoptions of best practices.

8. Acknowledgements

I want to thank Professor Jonathan Aldrich from Carnegie Mellon's School of Computer Science for his judicious, crisp and timely advice that helped me in every stage of writing this report. I also want to thank my former developer colleagues for making me a part of one of the most successful agile best practices adoptions within the company.

9. References

[CM00] Cardona, P., and Miller, P. The Art of Creating and Sustaining Winning Teams. International Graduate School of Management, 2000.

[AM01] Beedle, M., Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R., Schwaber, K., Sutherland, J., Thomas, D Agile Manifesto, 2001. [Available at http://agilemanifesto.org/principles.html]

[SH05] Subramaniam V., Hunt A. Practices of an Agile Developer, The Pragmatic Programmers, LLC, 2005, pp 6-7

[Ksh04] Schwaber,K., Agile Project Management with Scrum, Microsoft Press, 2004, Appendix B.

[Yip06] Yip, J., It's Not Just Standing Up: Patterns of Daily Stand-up Meetings, [Available at http://martinfowler.com/articles/itsNotJustStandingUp.html]

[Oxy03] Lougie, A., B. A. Glen, et al. (2003). Agile management - an oxymoron?: who needs managers anyway? Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Anaheim, CA, USA, ACM.