Customizing Agile for a Service Business

Aaron Sher, Principal Engineer

Vanteon Corporation

September 2012

Abstract: *The common agile processes such as Scrum require some customization to function in a service business, where real-time estimates, hard due dates, and hourly billing is the norm. This case study examines how Vanteon customized its agile process to fit its business model while maintaining the flexibility and transparency normally associated with agile development.*

Introduction

There are a number of well-known agile software development processes: eXtreme Programming, Scrum, Feature Driven Development, etc. All of them aim at more or less the same set of principles, and share many common practices, such as:

- Timeboxing (working in short iterations)
- Small, cross-functional teams
- Prioritizing working software over documentation and planning
- Self-organizing teams
- Face-to-face communication
- Abstract measurements of effort ("velocity")

These methods were, on the whole, created in a product-development context. That is, the intention of those who created them was that they would be used by a software development team who were implementing a product, either for sale or for use by internal stakeholders within their organization. My company, Vanteon, creates hardware and software on a contract basis, and aspects of these processes adapt poorly to this environment. Nonetheless, we have created an agile process for our own use, and applied it successfully to several projects.

This white paper will describe how this was done: which aspects of agile methodology we were able to adopt, which we were forced to adapt, and which we have had to discard entirely. It will describe the typical real-world process we use on a project, and how projects have succeeded (or, in some cases, partially failed) using this process. The hope is that others in a similar position will be able to use the lessons we learned to create their own adaptations of agile methodologies, and to avoid some of the pitfalls we encountered along the way.

The Issues

Flexibility versus Predictability

One of the core principles of agile is “don’t plan any more than you absolutely have to, because some of your plans will turn out to be wasted effort”. This is stated somewhat more pithily in the Agile Manifesto[1] as “[Value] responding to change over following a plan”. This principle is a reaction to the big-schedule waterfall processes, which insist that a schedule listing every task through the end of the project must be created and maintained throughout the project lifetime.

While in theory we entirely agree with this, in practice it’s somewhat difficult. Customers are generally inclined to want actual delivery dates, cost projections, and so forth. Convincing them to let us just dive in and start coding is difficult, so we needed a way to maintain the agile flexibility while still being able to satisfy our customers’ need for predictability.

Pair Programming

Some agile processes use a technique known as “pair programming”, where two engineers sit together and write code collectively. Since we are frequently billing the customer hourly, and each hour is associated with an engineer’s name, it was obvious from the first that this particular practice was never going to be practical for us.

Unit Testing

“Test driven development” is not technically part of the agile practices, but it’s closely related to agile and is called out as a best practice in many agile methodologies. We certainly write unit tests for our code, and on some projects we can get away with doing fairly comprehensive test suites and even writing tests first. However, it is often the case that customers look at test code as extra effort that they’re paying for but that’s not adding value to the final product. On many projects, we’re working with a large existing codebase where adding tests is not practical. In some cases, our code is executing on a small embedded device where we simply don’t have the capability to execute tests because of some limiting factor in the hardware (e.g., lack of storage space for the additional code).

Colocation

Many agile processes call for the stakeholders to be co-located with the development team. Since our stakeholders are normally members of the customer’s organization, that obviously wasn’t going to be feasible. Our development team itself is usually at least all in the same building, but given the flexibility of our team structure (necessitated by the short cycle time of many of our projects) it’s not practical to physically move all the team members into a shared space.

Applicable Projects

Vanteon doesn't try to apply the agile process to every project that comes along. In some cases it's the perfect fit, and in some cases it's just impossible. Here are the criteria we use to make the decision:

- Projects where the customer or a third party insists that we develop complete specifications up front (e.g., aerospace work that's subject to DO-178B regulatory oversight) are not good candidates.
- Projects where the requirements are known to be uncertain are a good fit for agile development.
- The agile process requires involvement from the customer. Customers who want to "throw the requirements over the wall" and then go away until the project is finished are not good candidates for an agile process.
- Projects where the customer insists on a firm fixed price and schedule are not good candidates.

The ideal agile project is one where the customer knows that their requirements are uncertain, is willing to be involved, and is willing to sign up to a run-rate style contract (i.e., we commit to a certain number of engineer-hours per month to work at the customer's direction until they tell us to stop.)

This sort of project comes up surprisingly often; unfortunately, many projects do not fit this mold. If the customer thinks they know exactly what they want, or isn't willing to participate in the process, or insists that we sign up to a hard delivery date and dollar amount, then we are forced to fall back on the old waterfall process.

Hardware Versus Software

Vanteon does both hardware and software development. I am a software engineer, so I tend to think in terms of software, but often the software we develop is running on hardware that is under concurrent development. Sometimes, we're developing hardware that doesn't have any high-level software at all, or whose software will be developed by someone else. While the agile process works well for software development, it runs into some issues dealing with hardware.

Hardware development goes in cycles, each of which consists of two main phases. In the first phase, the hardware engineers select components, design and lay out boards, and generally work with software and software artifacts (documents, schematics, etc.). The agile process works perfectly for this phase, as it does for documentation or anything else where the products are mutable.

This phase of hardware development ends when the board is actually fabricated. In the second phase, the engineers may be testing components and generally working with the actual hardware. The issue is that board spins cost a great deal of money, so suddenly the work products are no longer mutable: you can't easily go back and make a feature change to the hardware without incurring a prohibitive cost.

While it's still possible to do hardware development using the agile process, this 'cliff' has to be taken into account. In our experience, the best approach is to treat the two phases as separate efforts. The first may be agile or not, but the second is almost always run as a waterfall project. This makes it clear to the customer that there is a major discontinuity in the project at that point, and that changes after that point suddenly become far more expensive.

The Overall Process

Before adopting an agile process, we performed a review of the literature. We read about many existing agile methodologies, and looked at what they had in common and where they differed. Ultimately, we decided to start with Scrum [2], add some pieces that we liked from Feature-Driven Development (FDD) [3], and then customize as needed.

The process follows the overall pattern of Scrum: short, time-boxed iterations, with a planning meeting at the beginning of each and a stakeholder demonstration at the end. However, we made the following changes to adapt Scrum to our environment:

- We eliminated the Scrum Master position, since our teams are small enough that we can't afford additional overhead.
- The Product Owner is normally someone designated by the customer as the prime point of contact.
- We don't try to physically co-locate our team and our stakeholders; instead, we rely on heavy use of communication tools such as email and our JIRA [4] database.
- In place of Scrum's backlog of requirements, we use a "feature list" containing "features" – this is a politically more positive term, since some people see "backlog" as something that happens when you're behind schedule. "Features" is a concept we imported from FDD, which means "a small unit of testable, demonstrable, user-valued functionality".
- We use the term "iteration" rather than the more jargon-y term "sprint".
- We estimate all features as soon as possible and in real man-hours of effort (as opposed to abstract time units such as "story points"), and then refine the estimates at the end of each iteration. This allows us to project, at any moment, the delivery date and total cost based on the current feature list, with the understanding that the feature list is subject to change. This also means that any time the customer requests a change to the feature list, we can immediately

provide an estimate of the impact to calendar and cost, within the limits of our knowledge at that time.

- Our teams tend to be very small (2-4 people is typical) and in fairly constant communication. We tried the typical 15-minute daily standup meetings from Scrum, but eventually we decided that they weren't offering enough value to justify the expense. We now rely on other communications channels, plus informal meetings whenever they're needed.

Detailed Process

The following figure illustrates the overall process. The section from "iteration planning" to "demo" is further broken down below.

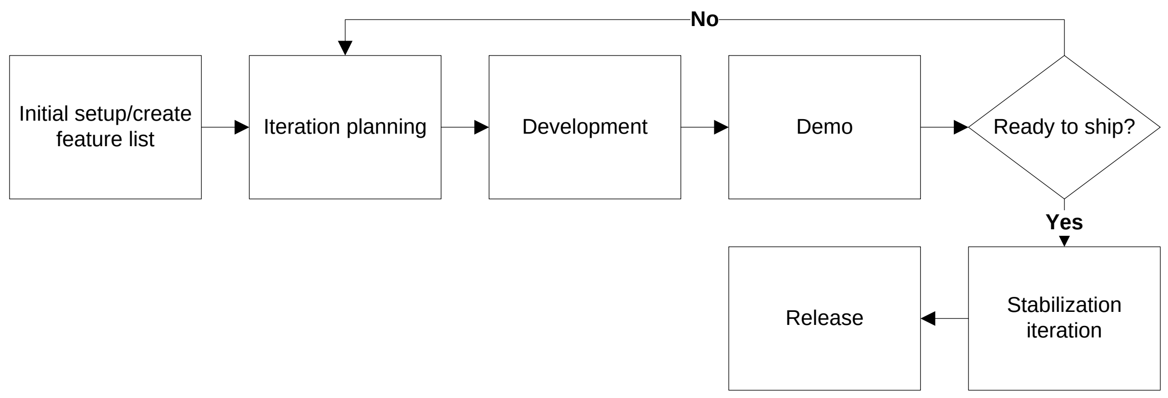

Figure 1. Overall process flowchart

Initial Setup

Prior to beginning the first iteration, the team (including the customer stakeholders) performs the following tasks:

- Familiarizes themselves with the project and its high-level requirements, insofar as they are known at that time.
- Sets up infrastructure such as the source control repository and JIRA project.
- Creates an initial feature list in JIRA. This may be very high-level, and many of the features may have few or no detailed requirements, but it serves as a starting point for refinement and prioritization.

Features in the list are phrased in the form of capabilities that the system possesses, e.g., "The software can display a list of all printers on the network." They are ideally on the order of four to eight man-hours of effort; smaller features impose too much overhead in managing them, and larger features make it difficult to track the progress of development. Initially, each feature is described in fairly general terms – as the requirements are worked out, they are added to the feature description. By the time it's implemented, it should be fully specified – if it's not, the engineer will probably discover the holes and ask the customer about them at that time.

The customer performs the following tasks:

- Reviews the initial feature list for accuracy and adds detail wherever possible.
- Prioritizes the initial feature list. This can be just a global ordering of features, or a high/medium/low priority for each, or any other similar scheme.

The engineers perform the following tasks:

- Add estimates, in man-hours, to each of the features in the initial list. Given the lack of requirements, these estimates are necessarily vague and quite likely inaccurate, but they're close enough to be a starting point.

Note that the customer tasks and the engineer tasks happen in parallel, and often iteratively – the customer's requirements may change the estimate, which might change the priority assigned by the customer or even the requirements ("I didn't realize how hard that would be, what if instead we…").

Development

Development proceeds in iterations. The following figure illustrates the process within an iteration:

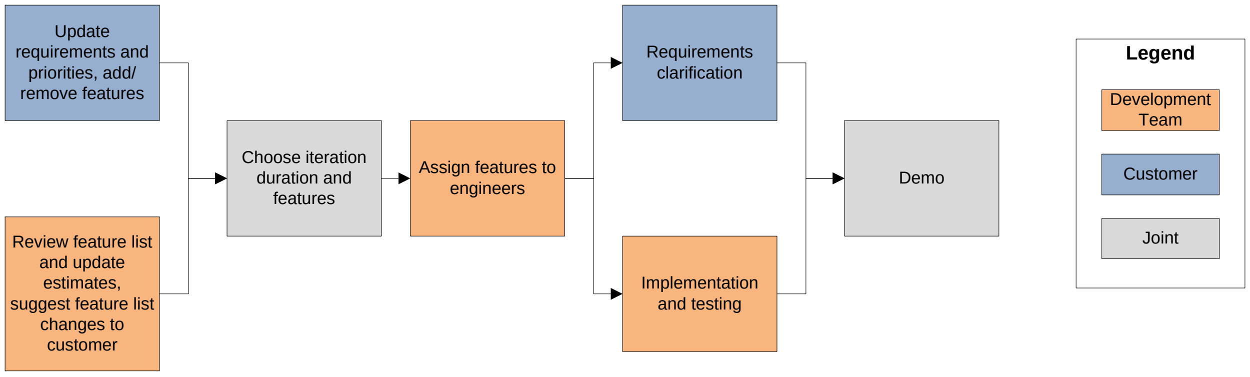
Update requirements and priorities, add/ remove features
Review feature list and update estimates, suggest feature list changes to customer
Choose iteration duration and features
Assign features to engineers
Requirements clarification
Implementation and testing
Demo
Legend
Development Team
Customer
Joint

Figure 2. Iteration flowchart

At the start of each iteration, the engineers perform the following tasks (ideally this happens *before* the iteration planning meeting):

- Review the open features remaining on the list, and update their estimates based on any new knowledge they have gained.

- Suggest feature list changes to the customer (e.g., “given the way this other feature worked out, it’s not clear that we still want this one” or “the implementation of this thing would make this new feature very easy to add”).

The customer performs the following tasks (again, ideally before the meeting):

- Reviews the open features remaining on the list, and updates their requirements and priorities.
- Adds, removes, or changes features at will, possibly based on the engineers’ suggestions.

The team then gets together, usually on a Web conference so that the customer can see the software running. During the iteration planning meeting, the team collectively performs the following tasks:

- Chooses a duration for the iteration. Each iteration is around 2-4 weeks long, but 2 weeks is the most common duration. Longer iterations reduce the amount of overhead in the process, but they also lengthen the feedback cycle and create more rework, which makes customers unhappy.
- Chooses the features to be implemented within the iteration. This is based on the number of man-hours available and the estimates for each feature. The last day or two of the iteration is reserved for testing and stabilization work. The choice of features is based on the customer’s priorities, but sometimes technical considerations (e.g., dependencies between features) dictate that lower-priority features should be implemented before higher-priority features.
- Assigns each feature to a particular engineer. This engineer doesn’t have to be the only person involved in the implementation, but he or she is the one ultimately responsible for that feature (the “feature lead”). In the vast majority of cases, each feature is implemented fully by a single engineer, so this rarely comes up.

Once all the decisions have been made, the team creates the new iteration in JIRA, and implementation begins. During implementation, each engineer tracks his own progress against the features assigned to him, and the project lead and/or project manager tracks the state of the iteration as a whole using burndown charts.

When requirements questions come up during implementation, every engineer is free to contact the customer (usually via email so that the conversation can be tracked, but phone calls work for urgent questions) to get them resolved. The answers are immediately entered into JIRA so that the feature’s requirements are kept up to date.

When a feature is completed to the engineer’s satisfaction, it is marked “resolved” in JIRA. A day or two before the end of the iteration (or sooner in some cases), a release is prepared for QA. The QA lead accepts the release and coordinates the testing efforts. Each newly-resolved feature is tested against its documented requirements, and either marked “closed” or reopened and assigned back to the engineer.

On the last day of the iteration (or the first day of the next iteration – we’re not completely consistent about that), the entire team meets with the customer stakeholders, usually via a Web conference. The QA lead demonstrates the software’s new features for the customer, and the team discusses any issues that are impacting development. Note that the demo is done by the QA lead rather than the developers – this helps to ensure objectivity, and also gives the QA lead some incentive to make sure that every feature is in fact fully tested before the demo.

Features that were not completed in the previous iteration might be simply moved to the next iteration, or in some cases the unfinished portion might be split out into a new feature. Any bugs discovered during the iteration and not fixed prior to the end of the iteration are entered as new features (e.g., “Software displays an error message without crashing when the user enters zero in the ‘denominator’ field”) and estimated and prioritized normally.

This end-of-iteration meeting is normally combined with the planning meeting for the following iteration, and the demo frequently leads to a discussion of changes or new features. These are added to JIRA and estimated and prioritized normally before the features for the next iteration are selected.

If the customer requests it, the software demonstrated at the end of any iteration can be packaged as a release for them. We don’t do this automatically because most customers don’t actually want releases that often, and because there’s a certain amount of effort involved in sending a release to a customer.

Project Completion

At some point, the customer determines that the feature set of the software is sufficient. At this point, we often do one more (possibly abbreviated) iteration consisting only of testing, debugging, and stabilization changes with no new feature implementation. The feature list selected for this iteration consists of the highest-priority bugs still open, but usually at least half of the time is left open for resolving newly-discovered issues.

Failures

Using this process, we have been highly successful (defined as “making the customer happy”) on a number of projects. However, there have been a few projects where the process did not work so well, and it would be unfair to present this case study without examining these as well.

Note that a failure of the process does not necessarily imply a failure of the project. In all cases, we have ultimately delivered a product that satisfied the customer’s requirements. However, we consider the process to have failed if we lose control of the development effort – that is, we lose the ability to predict the outcome, and devolve into a reactive mode where we are simply doing whatever the customer asks of us from moment to moment.

In general, the cases where the agile process has failed are due to a combination of factors: schedule and priority instability, requirements churn beyond what the process could easily absorb, lack of involvement from the customer, and lack of discipline in the development team.

Schedule and Priority Instability

Frequent and dramatic changes to the schedule and development priorities can wreak havoc on any process. For example, if halfway through an iteration the customer calls us up and says "we have a demo for *(some important end-user)* in three days and we absolutely need to have *(some feature that wasn't planned for this iteration, and maybe wasn't planned at all)* done by then", we have a decision to make. We can agree to the request, derail our development effort, and accept the fact that we won't finish the work planned for the current iteration, or we can tell the customer "no" and fail to meet their needs. Neither is a good choice.

Requirements Churn

Agile processes are designed to "embrace change". Vanteon's process allows the customer to make changes to the feature list at any time, with the understanding that changes to features in the current iteration will be managed as well as possible (that is, we won't go ahead and implement a feature that's been removed just because it was assigned to the current iteration), but might incur some rework. Changes to features that have already been implemented will obviously require rework, but this is known to everyone when the changes are requested.

This normally works very well. However, if the customer makes dramatic and frequent requirements changes ("it's a web service", "no, wait, we've decided it's a Java applet", "now we want it to be a native Windows app talking to a back end running on Amazon EC2"), the rework starts to overwhelm the original estimates. Even if we communicate the impact of each change as it happens, the end result of this process is usually an unhappy customer.

Lack of Customer Involvement

The process requires customers to be involved. Sometimes it's quite difficult to get customers to even look at the feature lists, much less to spend time prioritizing and detailing them. While this isn't necessarily a fatal problem, it means that the development team must spend additional time eliciting requirements and documenting them in JIRA. It also means that there tends to be more rework as the customer sees features after they are implemented and asks for changes because they don't match the image in his or her head.

Lack of Discipline

Even with significant schedule instability and requirements churn, it’s possible for the process to continue functioning, albeit with a lot of time spent in JIRA. However, it only works if somebody is willing to spend that time, to tell the customer what the impact of their changes will be, and to say “no” when the changes are just too dramatic. This requires discipline, and when things seem to be falling out of the sky, it’s very easy for the development team to just put their heads down and try to tackle the challenges as they come.

The problem with this approach is that it almost immediately renders the feature list obsolete. Once that happens, we no longer have any picture of what features are actually implemented in the software at any given moment, nor what we’re planning to implement. We can no longer say that on such-and-such a date we will have some particular set of functionality implemented, nor quote the cost of implementing any given set of changes.

While it’s possible to recover from this situation, it pretty much involves starting the project over. You have to create a new feature list, do all new estimates and prioritization, set up a new iteration from scratch, and so forth. That’s difficult and tedious, and once the team has gotten into this reactive mode, it’s very hard to convince them (and the customer) to take a break for a day or two in order to get the process back on track.

Conclusion

The process presented here is Vanteon’s adaptation of agile processes in a way that is practical within the constraints of our industry. It gives up some of the potential benefits of agile (e.g., pair programming) but requires minimal planning and oversight without allowing the project to devolve into chaos. While we haven’t had a 100% success rate using the agile process, overall it’s been significantly more successful, with significantly less overhead, than our standard waterfall process.

The lower overhead is achieved by demanding less “busywork” of our engineers and managers, which makes them happy. Customers get constant feedback and control, which makes them happy. At the end of the project, the product that we ultimately deliver is very close to what the customer wants at that time, rather than what they thought they wanted at the beginning.

Overall, our conclusion has been that agile is entirely doable within a contract-work environment such as ours, and offers significant benefits over traditional waterfall processes.

About the Author

Aaron Sher has been programming professionally for over 25 years on platforms including MS-DOS, Windows, Macintosh, Linux, and Android. He joined Vanteon (then Millennium Computer Corporation) in 1993, developing multimedia applications on Macintosh System 7. Since then, he has taken part in many dozens of projects including e-learning, image processing systems, consumer software, enterprise Web development, device drivers, and many others. He has been a developer, a software architect, a project lead, and a project manager, and is currently working as a senior software engineer and project lead.

References

[1] "Manifesto for Agile Software Development", http://agilemanifesto.org/, 9/19/2012

[2] "Scrum", http://www.scrum.org/, 9/19/2012

[3] "Feature Driven Development", http://www.featuredrivendevelopment.com/, 9/19/2012

[4] Atlassian JIRA, http://www.atlassian.com/software/jira/overview/, 10/24/2012