[M31]

Improving On Agility: Adding Usage-Centered Design to a Typical Agile Software Development Environment

Jeff Patton

Abstract

This paper describes, at a high level, the incremental development cycle typical of an agile software development environment, and how adding Usage-Centered Design will help this process run smoother. Specific points of applicability during the incremental development cycle are pointed out, along with the specific U-CD technique to apply there. The paper assumes a basic knowledge of agile software development and Usage-Centered Design.

Introduction

This paper describes, at a high level, the incremental development cycle typical of an agile software development environment, and how adding Usage-Centered Design will help this process run smoother.

Your mileage may vary

In this paper you'll find recommendations based upon the ways I've practiced Usage-Centered Design and the observations I've made while doing so. Fundamentally, agile development methodologies acknowledge the strong influence individuals and their talents have on any development process. The first of the four agile values emphasizes; "Individuals and

interactions over processes and tools." (Agile Alliance, 2001) I'll encourage you to try the specific recommendations given here. I'll also expect you to be alert for additional opportunities or challenges presented by your particular environment.

Usage-Centered Design

This paper assumes some knowledge of Constantine & Lockwood's Usage-Centered Design (Constantine & Lockwood, 1999). U-CD is a broad topic and the specific practice of U-CD can and should vary from organization to organization. For the purpose of this paper I'll summarize the U-CD workflow like this:

- Identify User Roles & Build a Role Model
- Identify Tasks Roles Perform & Build a Task Model
- Using the Task Model, Identify Interaction Contexts & Build a Navigation Map
- Write Essential Use Cases for Each Task
- Using Canonical Components, Build Abstract User Interface from Each EUC
- Draw Wireframe User Interface From Abstract User Interface

The Agile Methodology Isn't

It's important to underscore one particular point: agile software development isn't a specific methodology. The Agile Manifesto describes four basic values and 12 principles that generally describe methodologies that are agile. Use these values and principles to evaluate a specific methodology.

There are many specific methodologies that can be described as agile, most notably Extreme Programming (Beck, 1999). Other documented agile methodologies include Scrum (Schwaber and Beedle, 2000), FDD (Coad, LeFebvre & DeLuca, 1999), and Crystal (Cockburn). But, even more common than the documented methodologies are those organizations that build their own methodologies while consciously being mindful of agile principles.

When determining if a methodology is agile, I find that I have to use the pornography rule as quoted by Supreme Court justice Potter in 1964: "I can't define pornography, but I know it when I see it." The same seems to be true of agile development methodologies. Understand agile values and when looking at an agile methodology, you'll know it.

Typical Agile Incremental Development Cycle

XP has iterations and releases. Scrum has sprints. FDD has ?s. But generally speaking, agile methodologies develop software incrementally.

"There is no substitute for rapid feedback, both on the product and the development process itself. Incremental development is perfect for providing feedback points" says Alistair Cockburn in Agile Software Development (Cockburn, 2001). Whatever the agile methodology followed, documented or not, you'll generally find software development broken down into increments as short as a week and as long as several months with the "sweet spot" being about a month.

The Anatomy of an Increment:

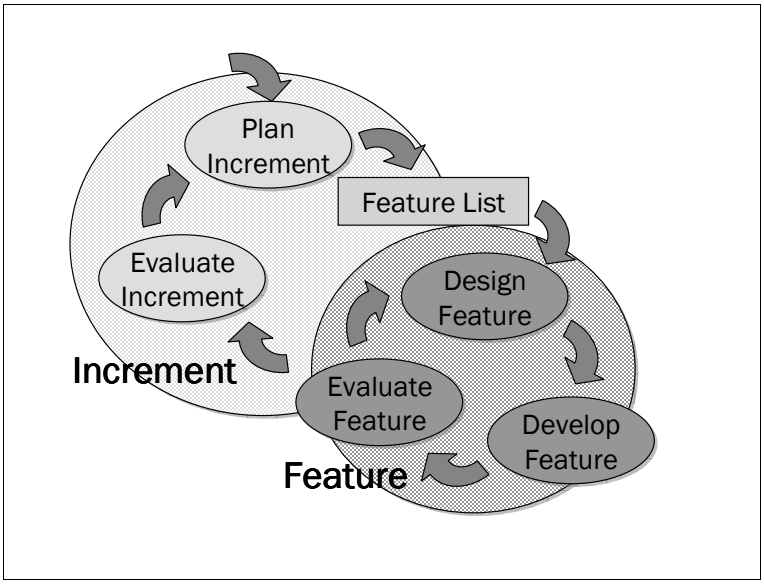

Figure 1 – An Increment

An increment looks somewhat like Figure 1. The flow starts with planning resulting in a feature list. Given a feature list those participating in the increment can proceed to design, develop, and evaluate each feature. At the end of the increment, all completed features are integrated and the increment evaluated as a whole. At this time those evaluating the software may choose to release it into a more formal acceptance test environment or into a production environment. If there's more software to

develop, another increment is started with increment planning. What follows are more detailed definitions for the parts of the increment.

Planning

Increment planning is generally done collaboratively. Ideally the plan will be built by developers, customers, and stakeholders. The resulting increment plan lists a number of *feature*s that you can reasonably expect to be completed during the increment. Often, planning sessions include an element of high level requirements gathering and/or simple design to arrive at or understand software features.

The Feature

Most popular agile methodologies label the work to be developed. Extreme Programming describes a user story. Scrum describes a backlog of backlog items. FDD describes parking lots of features. Regardless of the terminology these items of work that I'll call "features" have a few things in common.

Shared Understanding

The features are described in a way that is understandable by the developers building the feature and the customer, user, or business person requesting it. This shared understanding is arrived at through collaboration between developer and the feature's sponsor. Shared understanding implies that developer and feature sponsor can agree when the feature is completed. This understanding of completeness might be expressed as an acceptance test, manual or automatic, that we can run to validate that the feature performs as required.

Business Value

The features have some sense of business value. This value may be expressed through a sophisticated model for determining ROI, or just with a subjective label of "high," "medium," or "low," or any variation in between.

Cost

Developers building the feature and testers validating it are able to estimate, however coarsely, the time it will take to do so.

The Feature List

Given the shared understanding, business value, and cost of a feature, features can be labeled and prioritized. Generally speaking, agile methods attempt to build highest priority features first, and a prioritized list of features is critical to doing so.

Designing

Detailed design of a feature generally occurs during a particular increment. Ideally design is a collaborative activity that involves customers, users, or stakeholders along with developers. Methodologies such as Extreme Programming require a customer be on site with the development team to help with this design.

Developing

It's common for agile methodologies to place a particular emphasis on the quality of developed software. Extreme Programming requires that software be developed by programmers working in pairs, and that code is supported by automated unit tests. Whatever the methodology, the developed feature is expected to be completed and tested at the end of the increment, so quality should be high enough for this feature to be released into production.

Evaluating

As features are built, testers or customer representatives evaluate them to determine if they're acceptable. At the end of an increment, all completed features are reviewed together to evaluate the status of the software as a whole.

Agility Isn't About Speed

A common misconception of agile software development methodologies is that they are rapid development methodologies. Incremental development gives those running a software development project the ability to evaluate the position of the software as it currently is and:

- release it
- add, change, or remove features
- cancel the project

Agility is about having the ability to respond to change quickly. Often this translates into faster development speed. But, not always.

Increment to Release

In agile methodologies such as Extreme Programming, there's a preference for short increments of 1-3 weeks. These shorter "iterations" give more frequent opportunities to gauge development performance, evaluate the current state of the product, and adjust features and priority. It may be difficult in this short of a duration to build a set of features complete enough to be placed into use by actual end-users. In situations such as this, it's helpful to package several small increments into a software release.

When releases are used, a release plan is constructed to indicate the features that will appear in the release. The number and length of increments for the release is decided. Then plans are constructed for each individual increment. It's common for the evaluations at the end of increments to result in adjustments to the release plan. The evaluation of the final increment in the release then rolls into the evaluation of the release as a whole.

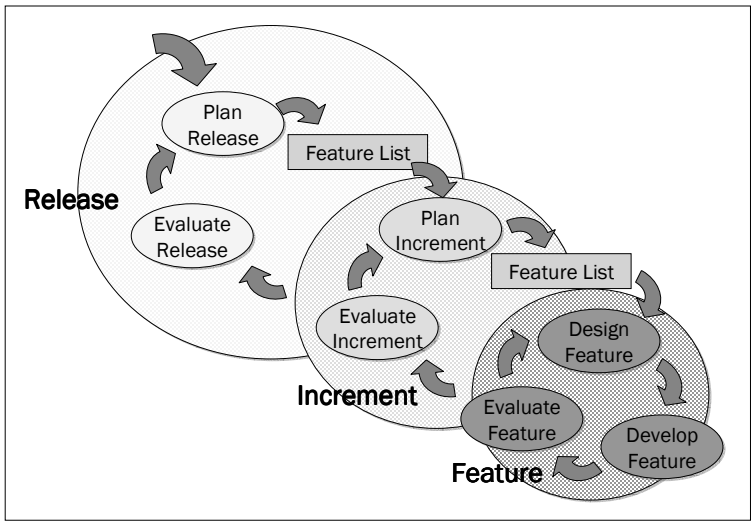

Figure 2 – Release with Increments

Figure 2 shows what a release might look like. A release can be thought of as a bundle of increments. Feature development cycles within an increment. Increments cycle within a release. It's easy to extend this concept to a project level where a project may cycle through multiple releases. A very large project may contain multiple smaller projects, each with their own incremental development cycle.

Inserting Usage-Centered Design

This is Tough Stuff

The basics of incremental development are simple enough. However, at every part of the incremental cycle, there are difficult design decisions to address and difficult scope trade-offs to make. Agility isn't for amateurs. It takes experience and skill to plan, design, develop, and evaluate a growing piece of software. This is where Usage-Centered Design offers valuable practices and insights.

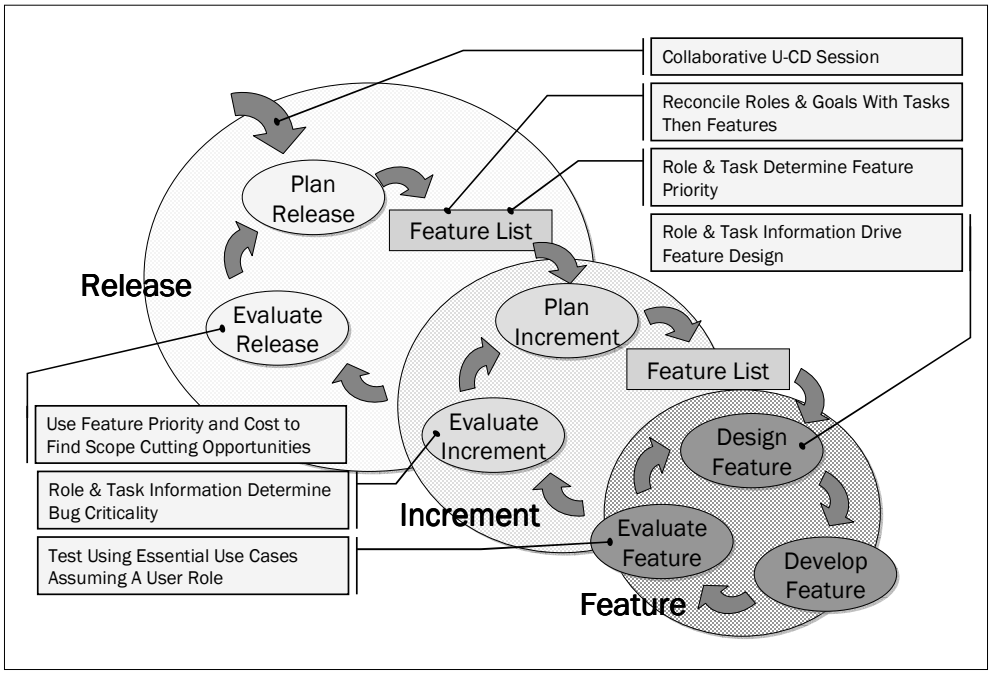

Figure 3 – Release with Increments and Insertion Points for Usage-Centered Design

Figure 3 points out the insertion points for Usage-Centered Design and specific U-CD techniques to apply at that point in the incremental development cycle. The remainder of this paper addresses these insertion points in the form of a problem statement followed by a potential solution using Usage-Centered design. Figure 3 calls out these solutions and the specific point in the incremental development cycle where they might most likely apply.

Apply U-CD When the Following Questions Arise:

What Exactly Are We Building Here?

When determining the features to build for a piece of software, it's valuable to involve users, domain experts, stakeholders, programmers, testers, and anyone else with information or experience that could benefit the project. However, once we assemble all those critical people, how exactly do we make best use of their time?

> **Collaborative U-CD Design Session**
> *Use Usage-Centered Design as a process framework to facilitate collaborative design sessions.*

Usage-Centered Design works particularly well as a process framework to facilitate a collaborative requirements gathering, designing, and planning session. Card based techniques are easy to learn and effectively engage everyone. The basic steps of U-CD can be run through as quickly as a couple hours or the session can be extended to several days depending on the size of the software under consideration.

A collaborative U-CD session goes a little like this:

- Sequester a diverse group of participants with the goal to determine the feature list for the software project or release, and from this feature list build a plan for one or more increments. This group should include developers, testers, customers, end-users, domain experts, and stakeholders. I've found the optimal size of the group is 6-10 people. Given that this is a small number, choose participants that can effectively represent their discipline as well as those who will be actively involved with the development, installation, or eventual use of the software.
- Build U-CD role and task models using 3x5 card sorting techniques. It's particularly important to identify the goals for each role and focal roles within the group of identified roles. For tasks, it's particularly important to identify the frequency the task is performed, the criticality of the task, and the subset of tasks that are focal tasks.
- From the task model, identify interaction contexts.
- From interaction contexts and tasks, identify software features. Often tasks directly translate to features.

- For each feature, derive value from the roles that will use the feature and task or tasks it automates. For each feature, have the developers in the room give a rough estimate of development time.
- Sort features by their value. Plan the next increment based on the time estimates given for each feature.

Figure 4 – A Collaborative U-CD Session effectively engages a diverse group of people in designing and planning.

How Do I Determine the Priority of This Feature?

You've identified a feature you know your software needs and you believe it's pretty important. How do you determine exactly how important? Based on what can you make that determination? You have multiple features to develop and you need to determine one feature's priority relative to another's. Exactly how can you do that?

> **Role and Task Priority Determine Feature Priority**
>
> *To aid in determining feature priority, use details about the user role that benefits from this feature and the task or tasks that this feature automates.*

U-CD's concept of focal role is important to determining priority. Features for focal roles generally have high value to the product and consequently high priority. Focal tasks are also indicators of high priority. If a feature is used by a focal role to engage in a focal task, the feature is likely critical to the software and will have a highest possible priority.

Beyond knowing what roles and tasks are focal, details about tasks also provide important clues to feature priority. Tasks that are performed at a high frequency are higher priority than those that aren't. Tasks that deliver high value to the business or that carry high risks when not performed correctly are high priority.

In practice, when a feature is supported by its known roles and tasks, assessing its priority relative to other tasks is usually easy.

How Can I Be Reasonably Sure I Found All The Important Features?

You've identified a list of features your software needs to have. You have a limited amount of time to develop this software before delivery is required and neglecting an urgent feature could be catastrophic. How can you be reasonably sure you haven't neglected an important feature?

> **Reconcile Roles and Goals with Tasks Then Features**
>
> *Using your user role model, make sure each real-world person you can think of is identified by a role in your role model. For each goal associated with each role, make sure a task exists to help the person in this role achieve that goal. Make sure each task is automated by a feature.*

Look back at the user role model you've prepared. The user represents an abstract role and goal that the real-world person might step into when using the software. Compare the user roles you've identified to the real-world people you believe will be using the software you create. Are all people identified by one or more role in the role model?

Look back at the task model. For the goals of each role identified, make sure tasks are identified in the task model that help the role fulfill their goal.

For each task, make sure it's automated by a product feature.

Reconsidering user roles, their goals, and tasks goes a long way toward gaining confidence that the software scope is as complete as it can be, knowing what we know. While no approach is perfect, I've observed this approach yields better results than others I've tried. Pay special attention to focal roles and tasks. If by chance some person or process was overlooked, there's a bit of comfort in knowing that it isn't one of the most critical people or activities represented by these focal roles and tasks.

What Should The User-Interface Look Like For This Feature?

You've identified a feature that needs to be developed. You know the user role it serves and the task or tasks it automates. Exactly how do we determine how this feature looks or behaves in the software? How much development time is it appropriate to spend making this feature behave smoothly?

> **Role and Task Information Drive Feature Design**
>
> *Use Essential Use Cases to describe the interaction between the system and user role. Derive software behavior from this. Use the user role information to determine specific interaction details. Use what we know about the user role's skills and experience along with what we know about the tasks frequency and criticality to determine the appropriate amount of rigor to apply to development.*

For each task write an essential use case. This is best done collaboratively involving a domain expert and developer. Use the process described in "From Abstract to Realization in User Interface Designs: Abstract Prototypes Based on Canonical Abstract Components" (Constantine, Windl, Noble, & Lockwood, 2000). Extract canonical components from the essential use case. Combine these canonical components into an abstract user interface. From the abstract user interface, render a rough wireframe user interface. Proceed to development.

This process need not be deferred until immediately before development. Designing user interface for focal roles or focal tasks is particularly critical. So, an approach of prototyping and testing the effectiveness of the proposed user interface is appropriate. Conversely, tasks that or performed infrequently by roles that are of low priority may be designed quickly and simply. For tasks such as these, you may elect to skip this process.

How Do I Go About Testing This Software?

Development of a particular feature is complete. We want to be sure the software is tested thoroughly and meaningfully. We don't have time to test every variation of usage. How can we most effectively test the software?

> **Test Using Essential Use Cases Assuming a User Role**
>
> *For the tasks implemented in the feature, write test cases using the essential use case for the task. While testing, assume the computer skills and domain knowledge of the role performing the task. Confirm the task helps the role reach his or her goal.*

Essential use cases provide the framework to write literal test cases. In practice, I find an experienced tester can derive test cases on the fly using an essential use case as reference. The tester tries to understand the environmental context the task is being performed in. Noisy, distracting, or time-constrained situations place unusual demands on the user and testing should take this into account. Also, take into account the frequency the task is performed. Users will gain expertise at repetitive tasks and be intolerant of extra keystrokes. However, infrequently performed tasks may require an extra-helpful user interface to aid the user with the details of performing the task correctly.

It's important the tester is familiar with the role or roles who will be performing the tasks in the feature. The tester needs to understand and assume the skills of the role performing the task. If multiple roles perform the task, evaluate the task from each role's perspective. Does the task being performed make sense in the context of the goal held by the user performing it?

I find that testers appreciate the opportunity to understand the context the software they're testing will be used in. It simplifies their job of testing to know what the goals of the actual user at the keyboard are. I've observed instances where testers didn't understand the goals of the person using the software and it resulted in bug-free software that was useless to its end user.

I've Identified a Minor Bug In A Finished Feature, Does It Need to Be Fixed?

During testing you've identified what looks like a minor issue with the software. The business process can still be performed dependably. The bug only occurs in some rare circumstances and during those circumstances the bug can be worked around. How do you determine the criticality of this bug?

> **Role & Task Information Determine Bug Criticality**
> *Use what we know about the role or roles using this feature to assess those roles' tolerance for issues in the software. Use what we know about the tasks this feature automates to determine the risks if this task is not performed correctly*.

Will the user performing in the role using this feature be tolerant of minor errors? If the user role performing the function is a skilled computer user, they may understand and tolerate minor idiosyncrasies in behavior. Less skilled users may not recognize idiosyncrasies as such.

If the task being performed occurs in a time constrained situation, or if the risks of incorrectly performing this task are great, then there's little or no tolerance for minor issues with the software. On the other hand, if the task is performed infrequently, time isn't constrained while the task is performed, and recovery is trivial, you may consider letting the issue go unresolved.

Some may consider it shocking to allow any known software issue to go unresolved. However, I've found it common to have to make such decisions as a hard delivery date draws near. It's generally not a question of fixing this issue or not, rather it's a question of fixing this issue or that. In situations where you have to choose, the supporting role and task information in U-CD give valuable information for making these trade-offs.

We're Nearing the Delivery Date For The Software And We Won't Finish On Time!

You've completed several successful iterations and can gauge by your past performance that your team can't possibly finish all the features in time for the promised delivery date. The date is fixed and end-users are counting on the new software being in place. You must deliver on time and cutting scope seems to be the only way to do so. How can you cut scope with minimal impact on the software?

> **Use Feature Priority and Cost to Find Scope Cutting Opportunities**
>
> *For each feature, use role and task information to determine priority. For features with low priority and high development times, look for alternative design approaches to reduce development time. Alternatively, look for low priority features that can be removed from the project to be replaced with manual processes*.

If increment plans were constructed such that highest priority features were designed and developed first, by the time you understand that the delivery date might be missed, the most critical features will have been developed. If you've developed a good sense of feature priority using user roles and task information, you'll find features that appear out of balance – features with low priority that take a long time to develop. This is where to look for opportunity.

Look at changing the user interaction to reduce development time while still meeting user goals. For example:

- interactive graphic user interface may be converted to command line tools
- interactive query user interface may be converted to a static report
- complex stand-alone computer process might be replaced with a paper process.

No matter what the approach, feature priority taken with estimated development time illuminates options for scope reduction. In practice I've found that when customers are made aware of development time constraints, and are given opportunity to collaborate on solutions to problems, very creative and pragmatic design approaches can arise.

Wrapping Up

Agile methodologies use incremental development to create flexibility when developing software. General functional design decision are initially made and described in the form of features to release at the end of one or more development increments. Increments start with a plan composed of a list of features to complete during the increment, and end with an evaluation of all features actually completed during the increment. Often, detailed design decisions are deferred until an increment starts and development is about to take place.

Usage-Centered Design techniques help all involved in the design, development, and testing of the software to understand critical information about the features they're working with. U-CD techniques are particularly applicable at specific, critical times of the incremental development cycle. I believe that adding U-CD to an agile development process helps simplify things at these critical times and greatly increases the likelihood of project success.

References

Agile Alliance, (2001) *The Agile Manifesto,* http://www.agilemanifesto.org/.

Beck, K., (1999) *Extreme Programming Explained*, Addison-Wesley

Coad, P., LeFebvre, E., & DeLuca, J. (1999) *Java Modeling in Color with UML*, Prentice Hall.

Cockburn, A., Crystal Methodologies, http://www.crystalmethodologies.org/,

Cockburn, A., (2001) *Agile Software Development*, Addison-Wesley.

Constantine, L., and Lockwood, L., (1999) Software for Use: *A Practical Guide to the Models and Methods of Usage Centered Design*. Reading, MA: Addison-Wesley.

Constantine, L., Windl, H., Noble, J., & Lockwood, L., (2000) *From Abstract to Realization in User Interface Designs: Abstract Prototypes Based on Canonical Abstract Components*, http://www.foruse.com/Files/Papers/canonical.pdf.

Schwaber K., and Beedle, M., (2000) *Agile Software Development with Scrum*, Prentice Hall.