EC|Wise

Integrating User Centered Design in Agile Development processes

© 2003, EC Wise, Inc., All rights reserved

Agile processes are software development processes that are guided by the principles of the Agile Manifesto. [1]. These processes all share the following values:

1. **Individuals and interactions over processes and tools.**
2. **Working software over comprehensive documentation.**
3. **Customer collaboration over contract negotiation.**
4. **Responding to change over following a plan.**

Additionally, they adhere to a set of twelve guiding principles. The first principle states that *"our highest priority is to satisfy the customer through early and continuous delivery of valuable software."* [2] Important themes of Agile practice are embrace change and team responsibility for organizing projects and establishing high quality standards. Here at EC Wise, we have long practiced in accordance with many of the principles of the Agile Manifesto, and welcomed this articulation of principles in 2001.

A number of development processes have evolved in accordance with the Agile principles; these include Feature Driven Development, Dynamic System Development Method, Adaptive Software development, and Scrum. The oldest and best known of the Agile processes is Xtreme Programming (XP). Kent Beck and his colleagues developed XP while working on on a large human resources project for Chrysler in the 1980s Ron Jeffries, a leading proponent of XP, describes the key XP practices in detail at http://www.xprogramming.com/xpmag/whatisxp.htm.

Drivers

The developers of the various agile processes were responding to a variety of frustrations related to the inability of traditional software development models to produce software that fulfilled user requirements on a timely basis. (This overview of traditional process models describes the Waterfall, Incremental, and Spiral methodologies that were popular in the 80's and 90's). These methodologies prescribed clear delineation in project roles and phases, with the result that programmers got handed massive, complex sets of specification and design documents by analysts and designers who then walked away from the project and left programmers holding the proverbial bag. In contrast, developers played the lead role in formulating agile processes, which minimize formal specifications, require individuals to play multiple roles, and integrate analysis, design and development throughout the development cycle.

A number of forces have accelerated the adoption of agile processes. Earlier methodologies were based on the assumptions that requirements were stable and could be determined before initiation of coding activities. In the current environment of changing business models and relationships, these assumptions no longer hold. Another driver has been the continual refinement of developer toolsets that incorporate components for producing requirements, design and code artifacts in a

single product. IBM’s recent purchase of Rational Software with the intent of integrating the Rational suite into its development tools underscores this trend. Ten years ago, it really did take a long time to simply code, compile and debug a relatively small feature. Today’s high power IDEs accelerate code production with wizards and forms tools that generate code, and move the focus of work to designing components and their interfaces with users and with other components.

By reducing the cost of distributing and deploying new versions of software, high speed corporate and public networks have made it economically feasible to frequently release new versions that provide incrementally greater value. For example, when we developed commercial accounting software products, the upgrade cycle was so painful that our customers could only tolerate it every couple of years. Now we use Web based accounting software that’s upgraded every couple months with very minimal impact on our business operations.

Agile processes reflect this move to delivery of products in a service model, characterized by “streaming” versus bulk delivery of value. Software can be delivered this way because it is highly divisible. Unlike with a physical product like a car, a customer can get value from a subset of the features planned for the final product. Both the developer and the customer gain advantages by delivery of a limited feature set version of the product early, the customer in operational efficiencies and the developer in better feedback throughout the balance of the development process.

Rapid changes in markets and technologies and the general emphasis on quick return on investment mean that the only projects getting approved are highly focused applications that provide very clear operational or competitive advantages. These are ideal scenarios for Agile processes. An associated factor is that many businesses are engaging in transient ad hoc relationships where large investments in supporting software infrastructure are not justifiable.

Project managers buy into Agile processes because their focus is on reducing risk. Agile processes are more empirical than waterfall and other blended phase processes, using trial-and-error to reduce the risk of designing and building the wrong thing. Project managers are willing to accept somewhat higher maintenance costs in return; these arise from low-level re-work and the effort involved with having to keep all code close to release-quality at all times. Viewing an Agile project as a series of very high-fidelity design experiments, project managers accept high-level uncertainty in exchange for low-level certainty. Agile is evolutionary versus revolutionary, allows fewer assumptions, and prevents taking on too much at once. Thus, it addresses some of the problems of the phased approaches.

figure 1: Ideal phased approach

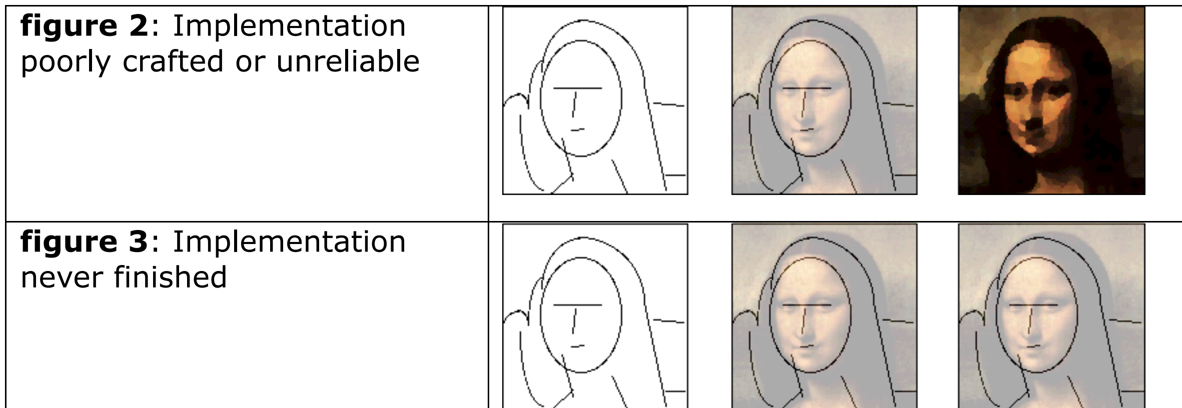

figure 2: Implementation poorly crafted or unreliable

figure 3: Implementation never finished

The essence of Agile processes is that projects are broken down into a series of mini-product extensions that are started, completed, tested, and delivered to the customer relatively quickly. Customers can provide feedback on each cumulative delivery. The mini-projects, called iterations (based on *stories*) in the XP lexicon, would be built upon one another over time, resulting in “emergent” rather than “predetermined” software systems. Each iteration can be independently and automatically tested for reliability, which provides the confidence to make the inevitable design changes in subsequent iterations. In the end, customers get more of what they want faster, because they have regular opportunities for input on real working product throughout the development process.

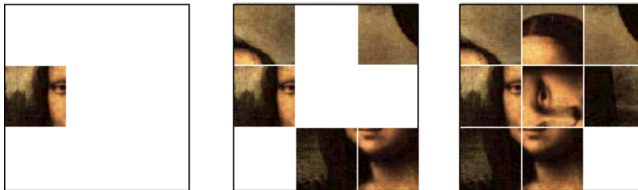

figure 4: The risk of the XP approach

Agile Challenges

Of course, proceeding without an overall plan risks producing a product that lacks a coherent overall structure. [fig 4] Continuous refactoring is the touted solution. The pain of such a result is felt first in the user interface. Lacking a design threatens traditional usability practice because experience and foresight is replaced by empirical tests with real-world use. Lower transaction costs allow trial-and-error to replace predictive expertise and user research. The problem is that while you can test “badness” out of a user interface, you need to design “goodness” in!

Without (relatively) long periods to gestate larger ideas and designs, there is less ability to assign value to vision, be aware of opportunities not taken, or conceive of complex, interrelated, consistent designs. These constraints risk solutions that are acceptable but suboptimal. And even unimplemented concepts have intellectual property and intelligence value. For example, it is valuable to know what you could (or should) be doing, but are not. This knowledge can point you where you want to go, or expose what competitors might offer in the future. In the long run, generative time spent creating this intelligence can be as valuable as time spent in iterative, marginally improved product releases.

To a traditional designer, the implications for practicing good, user-centered design on Agile projects may appear grim. Martin Fowler articulates this concern for us in his essay "Is Design Dead?" [6]. Fowler's answer to his question is actually "no", but note that his use of design refers to engineering-oriented system design versus user-interface, interaction, or user experience design. In fact, there is very little discussion of user interface issues in the agile community. This may reflect the fact that agile methods are more commonplace for products with less UI, or that in Web applications development of the user interface layer is often dissociated from implementation of the underlying business logic.

Of course, agile process emerged in reaction to the failure of large, ambitious software projects that included a heavy up-front "design" phase.[4] While much of the "design" work included in these projects was concentrated at the system level, the ambiguity of the term has led to some derogation of the user interface design role. The fact that many smaller projects undertaken during the Web boom failed because graphic designers, primarily trained for print design, were unable to design plausible interactive systems beyond the simplest of functionality fueled this fire.

Despite these shortcomings, agile processes do have aspects that can enhance design practice. For example, it is very valuable to have working applications that customers can use, even if these versions do not represent the best design possible. These releases can be treated as ongoing usability tests that occur during actual use, and can produce deeper feedback because the users get much more involved with a product that they have to use regularly. Also, this arrangement allows automated usage tracking and testing tools to be brought into play sooner and in a more relevant context.

Tight interaction cycles also can encourage experimentation in a low-risk environment. With a team geared a tight release and revise cycle, unproven designs can be tried, and discarded if need be, with relatively small impact on the project. Also, since there is no large, ominous specification documents to write, a smaller team of analysts and designers can focus on managing the impact of changing business requirements within the iteration cycle. As a result, the process can be both more predictable and more responsive. Because there is closer team interaction, shared goals, and less solitary time invested in elaborate design or engineering schemes, there is less defensiveness and territoriality among team members about individual designs, which makes for more cooperation.

User interface designers working independently from a development team might commit to a design without building and testing it in its final implementation medium. When it's later turned over to development specialists and implemented, it's not unusual for designers to be disappointed upon seeing the design in its finished form. Its also for designers isolated from developers to deliver designs that are far more difficult to implement than a comparable design that had been considered by the designers but rejected, and thus remains unknown to development.

Whether designers are forced to work this way, or whether they do so willingly, it results in either rework by the team or the release of a flawed product. Agile projects avoid such incorrect assumptions by making fewer assumptions, making them at a smaller scale that minimizes risk, and providing regular feedback to the designer in the form of working product.

The Role of Design

Although design can be minimized and marginalized into iterations, it cannot be eliminated entirely. Martin Fowler describes the process this way:

> After reaching consensus on the details of how the new functionality should work, [the domain experts] divide up the responsibility of articulating that functionality in a work product we call a narrative. A narrative is written for each story card slated for the iteration. A narrative is similar to a use case, but is less formal.
>
> At the end of the narrative, a sketch of the relevant test scenarios is laid out. At a minimum, the test scenarios must be articulated sufficiently to allow a developer to estimate the detailed programming tasks related to the story card." [5]

It is interesting that the team "designs" what will be done in this iteration. However, the result is actually called a "test" to be compared to the final code, and the emphasis is on completing the iteration on time. However, the authors admit that they continue to struggle with ways to effectively communicate the requirements and design issues to the developers, which is one of the reasons that agile processes require frequent, ongoing communications between developers, analysts, and often the business users. Fowler and his associates comment:

> "Ours is certainly a developer-centered process, and the analysts have found this a challenge. In particular XP doesn't really give much advice as to how analysts should develop and organize stories, particularly for such a complicated domain. We're learning slowly, but this is an on-going problem." [7]

It is precisely at this point that EC Wise began its efforts to insert design in a more meaningful way into the agile process. Regardless of how a team works, the system has to be designed. Whether the design occurs in a large, potentially spectacular, but risky up-front effort, or in small, iterative steps that steadily grow into a usable product, it has to occur somewhere.

Art as a Metaphor for Product Development

Drawing and painting provide a good metaphor for understanding product development. When we were learning to draw as kids, we would start with one part of the drawing and try to render it in full detail, then move on to other parts in the same fashion. However, things often went drastically wrong when a lack of planning caused gross inaccuracies. [fig 5]

figure 5: Ad hoc development

Frustrated, we read instructional books, and they recommended making a light sketch first to plan the composition and scale relationships. Then, gradual commitments to light and shade should be applied to shape the composition. The idea was to develop the whole drawing gradually, in "layers", to get the most accurate view of the overall drawing as possible. Initially we resisted making lines that would later be erased, but the advice eventually worked; arriving at an accurate solution became more predictable. [fig 6] The eraser became our friend, and this fundamental method has stayed with us to the design of complex software systems.

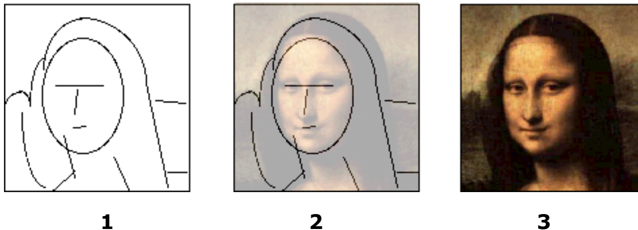

figure 6: Staged progression: agile approach

(1) We execute a design spike each time a completely new sketch is needed

(2) We use the eraser (refactoring) to constantly change the design as needed based on user feedback.

(3) The finished product is complete and coherent.

Large software projects are more complex than drawings. They traditionally involve large teams of people, long timelines, and shifting business and technology contexts. Throughout a project lifecycle, the form, or "current version" of the end product, is expressed in different media. It begins in the form of words, then shifts to designs and prototypes, and then to alphas, betas, candidates, and releases. With each media shift the product form is expressed in greater detail. [fig 7]

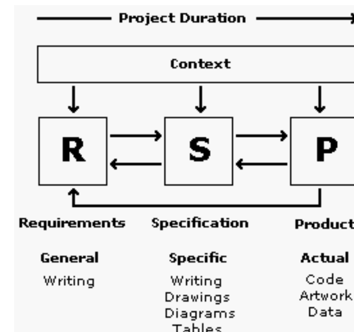

figure 7: Forms of product representation

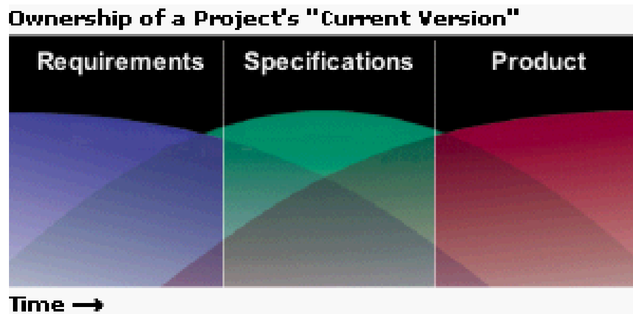

figure 8: The Development Relay

As displayed by the Development Relay, [fig 8] a product's current version resides in a series of media, from writing to images + writing to code. These transitions introduce inefficiencies that cause product information to be lost. XP minimizes such inefficiencies.

Of all the external forces that serve to change project requirements, in most cases **the project has the greatest impact on the project.** In a portrait, the point where we put the mouth has an impact on where we put the nose. The same is true with product development, and design exists to be able to predict quickly and cheaply what the product will be before it exists. Proponents of heavy up-front design, such as Alan Cooper [4], claim that all these interdependencies can be solved in a specification. This is true for some cases, but in cases where technology is new or untried, requirements are volatile, or the domain unfamiliar, it is risky to invest so heavily in assumptions without conducting "reality checks". Also, when customers see a product actually work, they can change their mind about what they want.

Given this set of opportunities and constraints, EC Wise was determined to find a hybrid process that took advantage of the benefits of both agile processes and design planning. The solution we found involves simultaneously operating ongoing design efforts on a number of levels.

Extending Agile Methods from Development to design

To add the most value in the least time, we have adopted Agile development methods where we often design and code in parallel, relying on close, articulate communication to stay in synch. Our teams consist of 5-20 members, often including client team members, representing roles such as project manager, product designer, programmer, analyst, researcher, graphic designer, and system architect. We emphasize usability and system performance in our work, and practice user-centered, iterative design practices when possible.

Since we practice agile methods, we try to avoid producing heavy documentation, and prefer to devote effort to design and coding. However, user-centered design requires making iterative cycles of design representations in multiple fidelities and formats. The following diagram (fig. 9) describes our design and development process:

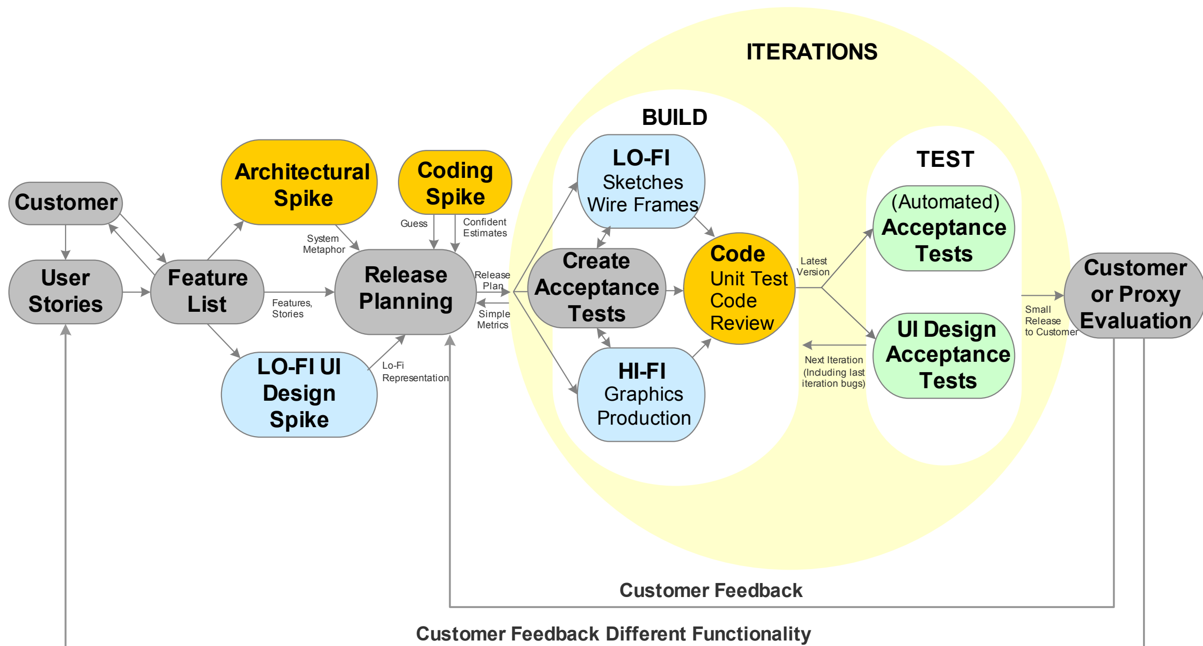

Figure 9: EC Wise development process.

When we first engage with a client we investigate user and business requirements and quickly generate a catalog of user stories and a candidate feature list. Then we go through an Architectural Spike and a low fidelity user interface design spike. These typically last from one to three weeks. It's the project manager's responsibility to keep them focused, and in synch, and to cut them off as soon as they have generated enough information to support release planning.
The purpose of release planning is to develop a first cut iteration plan, with features assigned to iterations. There's typically a coding spike that occurs in conjunction with release planning to validate that the allocation of features to iterations will be realistic.

Our design spike focuses on the usage scenario level, where we design the product at the fidelity needed, ranging from quick sketches to Shockwave-based storyboard depictions of users completing workflows. Storyboards show how users would use a series of components, or groups of UI functionality (e.g. a browser tree, a form, a dialog box, etc), to complete a task. In the storyboards, the design shows partial depictions of these components; we also build just enough to gather feedback from users and stakeholders on the merit of the overall design. [fig 10]

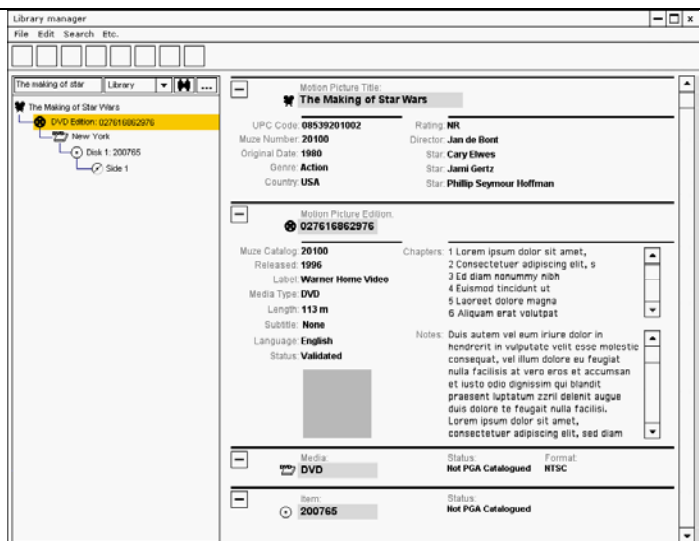

Figure 10: Storyboard wireframe, with multiple

	components

Once a project schedule has been agreed to, iterations begin. During the first iteration, we would typically build out some core functionality (usually the highest risk), with a relatively low fidelity user interface based on the work done during the design spike. At the same time, the design team looks ahead into the next iteration and starts to flesh out the design of scheduled features. For those that were not addressed at all in the design spike (due to their not being part of focal interfaces), designers will create low fidelity representations of the functional workflows. For those that have had low fidelity designs, designers will create any required graphics and more detailed workflows that may be necessary to support the engineer.

We follow Scott Berkun's practice of including in a specification only what the programmer needs. "If programmers do not understand something, add more detail."[3] The effectiveness of this approach is dependent on regular interactions between designers and developers, both in weekly iteration meetings and daily project reviews. The developers need to realistically articulate the level of detail they will need for upcoming features; that level varies based on the complexity of the feature and the developer's familiarity with the domain.

In many cases, a particular feature will go through a series of refinements throughout the project, based on the principle of functional divisibility described earlier: in this case there is value in providing the customer with a working version that lacks some of the refinements and details that could be imagined for the feature. One application of this pattern is the division of a complex feature into its minimal form, and then building and releasing only the minimal form. If it works and the client likes it, we add more functionality over time. Below are two simple, illustrative examples:

A. In a standard scrolling list, you can:

1. Select one item
2. Select multiple items
3. Select discontiguous items

We might initially implement a scrolling list that supports only item 1, which has value to a customer. Though it is possible to provide features 2 and 3, there can be value to having 1 released sooner and more reliably.

B. Update = Insert + Delete

1. First, implement the Insert and Delete functions separately.
2. To update an item, its possible for the user to copy an item to the clipboard, delete it, create a new instance, paste the item on the clipboard, make changes, and save the new item. Or to create an entirely new item "from scratch".
3. When users complain that this is too cumbersome, we can create a user interface that simulates the editing, but continues to delete the old item and insert a new one "under the covers".
4. Finally, when users realize that the semantics of this operation are not equivalent to an edit, we would reimplement the operation "under the covers" as a true update of the existing item, with the requisite multi-user controls.

Insert and Delete have value, and Update will take longer. There are two levels of functionality along the way that may fully satisfy the customer, or may become obsolete or unwanted.

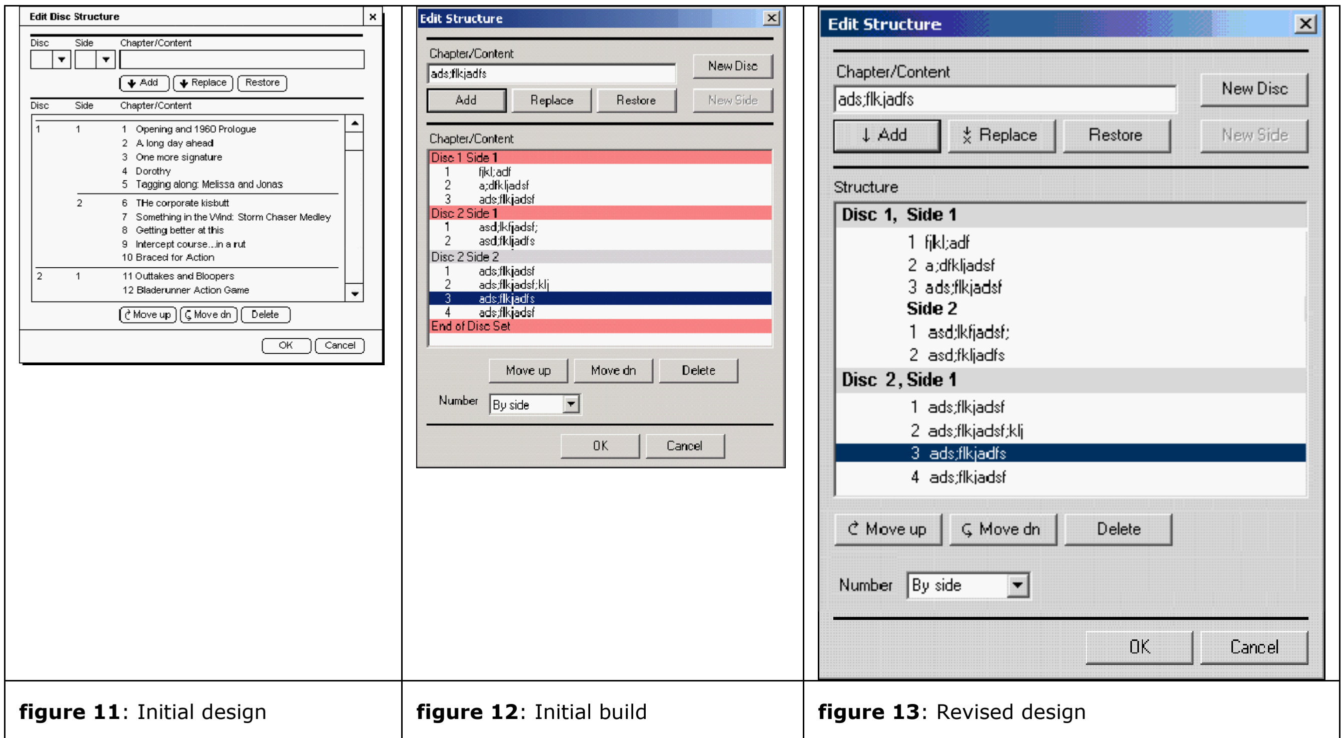

figure 11: Initial design	**figure 12**: Initial build	**figure 13**: Revised design

A dialog box component provides a counter example, where the approach of providing minimal functionality at first resulted in significant cost savings. [figs 11-13] The initial design of a dialog box for creating and editing DVD chapter lists used an existing project design pattern for list entry, and a graphic display with hairline dividers. Because the time required to build these relatively custom displays would not fit within an iteration, standard list management solutions were used in the first programmed iteration. When we operated this version, the customer liked it so much that we abandoned the considerable extra work to build the original design and merely upgraded alignments and colors, and added the button icons.

This process requires close communication and openness. We distribute designs to users in Shockwave format for phone feedback, or meet with them in person for walk-throughs. We then revise the designs and re-submit them to the engineering team. We use new design iterations to change a design, add more detail to the stories and test cases, and incorporate new features. We do not typically maintain schematic models to describe the systems, using Fowler’s principle of creating spot representations based on need: “draw a diagram of important things, then throw it away”.[7]

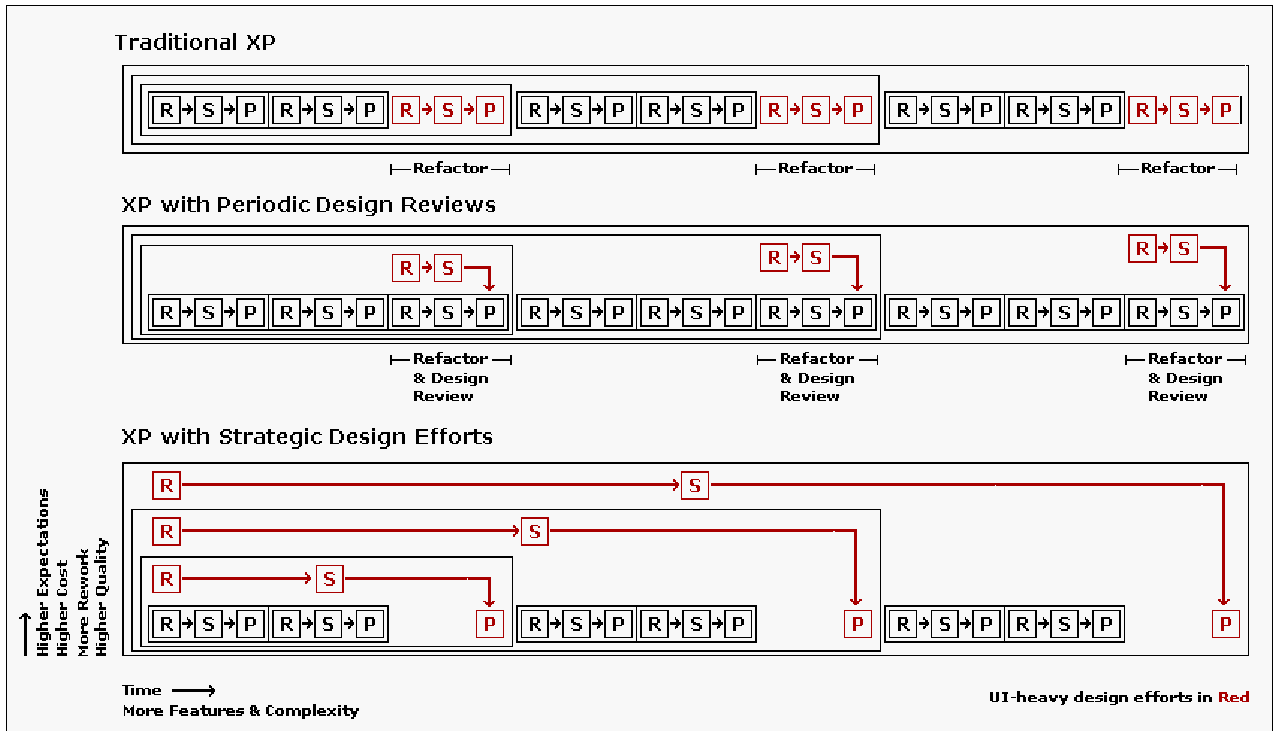

figure 14: Integrating design into an agile process

EC Wise's hybrid approach requires the designer to work on two levels. The high level presents low-fidelity product representations of scenarios in order to drive the user-centered process. Concurrently, the designer works on the short-term iterations by providing detailed component designs that enable construction. This split duty limits the designer's ability to finish either in one pass, but because all design and product assets are built in a flexible manner, it's easy to layer on improvements as the project progresses. [figs 14]

[1] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C, Mellor, S., Schwaber, K., Sutherland, J. , and Thomas, D. The Agile Manifesto. http://agilemanifesto.org

[2] Ibid.

[3] Xprogramming.com: An Extreme Resource. http://www.xprogramming.com/xpmag/whatisxp.htm

[4] Gibbs, W. W. Software's Chronic Crisis. Scientific American, September 1994, 86-95. http://www.cis.gsu.edu/~mmoore/CIS3300/handouts/SciAmSept1994.html

[5] Fowler, M., Taber, C. Planning an XP Iteration. http://www.martinfowler.com/articles/planningXpIteration.html, 4-6