Finding the Forest in the Trees

Jeff Patton
ThoughtWorks
jpatton@thoughtworks.com

ABSTRACT

While the iterative development approaches found in Agile Software Development fulfill the promise of working software each iteration, the task of choosing which software to build first can be formidable.

This experience report discusses my team's experience working with a large healthcare company writing software for use in their hospital's newborn intensive care unit (NICU). The very large scope of this project and the urgent need for delivery made project release planning difficult. Focusing on capturing feature details in XP style user stories led to confusion about priorities and release strategy. Making good use of User Centered Design user role models and task models gave us the big picture we needed to un-stick the release planning process and effectively choose the bit of project scope we needed to focus on for our first and subsequent releases.

Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: Elicitation Methods, Methodologies

General Terms

Management, Documentation, Design, Human Factors

Keywords

Agile Software Development, User Centered Design, Software Release Planning, Requirements

1. WHY AGILE?

1.1 The Problem

During fall of 2004 ThoughtWorks had the privilege of helping a major health care provider transition from a mostly waterfall software development process to an Agile approach to developing software for use by its over 20 hospitals and dozens of clinics and surgical centers. Prior to the decision to adopt an Agile approach this health care provider, hereafter refereed to as HCP, followed a somewhat traditional process of gathering requirements during a requirements phase, building software based on those requirements over the course of many months to over a year, then testing the software for release into the clinical environment. This approach had its pros and cons.

On the positive side, requirements were generated during the course of many collaborative meetings involving the Information Systems team (development and medical informaticists, specialists with both medical skills and software development skills), and clinicians who needed the software. Requirements documents were relatively light since much of the understanding could be retained in memory by those involved in the conversations.[1]

On the negative side, it took many months before the results of design and development were visible to the clinicians expressing the requirements. Upon seeing the software, unpredicted new requirements and requirements changes would often emerge. Changes might be necessitated by initial misunderstandings or simply by modifications occurring in the approach to doing work over the several months after requirements discussions. IS owned the resulting software design and could then make the choice to respond to, or defer new requirements. Trying to accommodate many new requirements and still meet deadlines often resulted in reductions in quality, making the people on the business side unhappy. The *business side* included the clinician users of the software and the administrators sponsoring its construction. Deferring requirements left the software missing important features which made the business side unhappy. Simply stated, the business people were often unhappy.

1.2 A Solution: Business Drives Development Using an *Agile* Approach

During 2004 a new HCP Information Systems director recognized the problems and suggested that following an Agile approach to development might resolve them. An Agile approach would change requirements gathering from an early phase of development to ongoing involvement with clinicians on the business side. This ongoing involvement would help ensure that problems with ballooning requirements were detected and dealt with earlier. Following an Agile incremental development approach would allow clinicians to watch the software emerging from short development iterations as opposed to the long many-month development cycles they'd seen before. The hopes were that an Agile approach would allow the business side to appropriately drive the development of the software and allow them to better accommodate emergent requirements and get working software into hospitals and clinics faster. This should improve clinician productivity and patient care at a higher rate, and make the business side, clinicians and administrators, happy.

To make a fast transition to an Agile development approach, ThoughtWorks was selected as a vendor to coach HCP staff and

[1] Alistair Cockburn's Agile Software Development [5] discusses modalities of communication with face to face communication in front of a whiteboard as an example of the richest form of communication.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

co-develop software in an initial high impact business area that built software for the care of women and newborn infants.

2. SETTING UP TO SUPPORT AGILITY

2.1 Continuous Collaboration Isn't Free

Where in the past HCP had collected requirements during meetings held over the course of many months, an Agile approach suggested we have continuous involvement from an *on site customer*. An on site customer as described in Beck's Extreme Programming Explained [1] suggests a person or group of people familiar with the software that needs to be built. HCP filled the *Agile customer role* by taking nurses directly from hospitals and giving them the dual role of part time Agile customer along with their part time nursing duties. These expert users should make ideal Agile customers since they fit Extreme Programming's description of a customer that "will really use the system when it is in production." [XPE p60] While adding these expert users into this role isn't free, the improvement in software quality and delivery times should more than make up for the extra expenses.

It's worth noting that these expert users weren't normally the same expert doctors and nurse practitioners that participated in the up-front requirements sessions in HCP's old process. Those original doctor and nurse stakeholder's time was too constrained to participate on an every day basis. These new expert user/Agile customers now had the responsibility of participating in early collaborative sessions with the original stakeholders to capture long range product vision, goals, and priorities, along with details of their business processes. The Agile customers would then share ongoing progress on a regular basis with the original stakeholder group. The developers and medical informaticists from the IS side that normally collaborated with stakeholders now relied on the Agile customers for their information.

2.2 Customers Manage Their Details With Stories

In most Agile processes development work is divided into small pieces that are released into an iterative development cycle normally lasting one to four weeks. In Extreme programming these small pieces are called *user stories*. Feature Driven Development [11] refers to them as features; Scrum refers to them as items in a large product *backlog* [14]. Early definitions of a user story given by Extreme Programming, from 2000, suggested they were a promise for a conversation between developer and customer. What was written down was just enough to start that conversation – usually a story name and a short paragraph written on an index card. [XPE p90] Over time the definition of story evolved from something the customer wants the system to do that has business value to suggest that a story must have business value, have enough information so as to be estimable by development, and be small enough to be completed within an iteration. For the HCP Agile customers that iteration was two weeks, with a preference for stories that could be completed in three developer days or less. This meant that those in the customer role create user stories and write tests to validate that the stories are implemented correctly. Customers should then be available on site for daily collaboration with developers to help explain and validate that stories are being implemented correctly in code.

2.3 Describing a Complex Software Product Takes Hundreds of Stories

As HCP customers began to describe their software in stories, they quickly realized that even a small software release, one expected to be completed in three months, would be composed of dozens of stories. It was more true to say the entire scope of the software would be composed of hundreds of these stories from which dozens would be selected for a smaller three month release. Identifying product requirements as small stories was new and very hard work for those in the Agile customer role.

In their old process it was development's job to break the software into smaller pieces should they choose to do so. In HCP's old process it was assumed a release would be delivered "feature complete" so there was no need to carve out a subset of features for an early incremental release.

2.4 Stories About Medical Software Read More Like "Novels"

As development progressed, the customer team found that for developers to confidently estimate the development time for a story, they needed a fair bit of information. The medical domain is very complex. There are many legal rules to consider when implementing new functionality. There are many existing services that the new software must integrate with – services that manage medical terminology and external application customization points. Developers found that by understanding these rules and external dependencies before estimating, they could estimate with more confidence. To help developers, those in the customer role began researching these rules and external dependencies and documenting them for inclusion in a written user story. The result was a user story that, while functionally simple, might contain many pages of supporting documentation. Writing these detailed user stories was hard work for HCP's Agile customers.

3. THE FIRST AGILE DELIVERY

The delivery team composed of HCP's expert user customers, ThoughtWorks analysts, HCP developers, and ThoughtWorks developers worked together for three months to successfully deliver additional features and changes on an existing project.

This first delivery wasn't easy. HCP developers were burdened with learning new development techniques such as test driven development [2], refactoring [7], and more sophisticated object oriented design principles and patterns [8]. Those in the customer role needed to learn story writing skills, test writing skills, and adapt to the continuous need for communication with the developers they supported and the stakeholders they conveyed ongoing progress to.

Given those challenges, the team successfully delivered a release that was met with responses from its user community like "this is the best version we've seen yet" and "…I see major improvements!"

4. STARTING THE NEXT PROJECT

4.1 The Newborn Intensive Care Unit

The Newborn Intensive Care Unit (NICU) currently used a system for documenting medical records that while useful had a number of issues users wanted to see corrected:

- The performance of the existing system was often a problem such that the daily documentation done for infants in the NICU might take several hours to complete for an attending physician or nurse practitioner. If the system could save and retrieve information faster that would help.
- Doctors and nurses had identified a number of new features and changes in workflow that would help them do their work more effectively.
- Doctors wanted to see improvements to the quality of information they gave at patient discharge time. While the legacy system did give all the information, the information could stand to be presented more concisely and clearly.
- And finally, many of the existing applications for medical records used in the hospital had been upgraded to a newer, faster, flashier J2EE platform. The NICU system hadn't been updated yet and stakeholders believed this would help resolve many issues.

Taken all together, these reasons made the urgency of getting a new system to those in the NICU high.

Through many planning discussion over a long period of time the NICU department initially received funding to replace the NICU system as-is with newer technology, then eventually received funding to add additional features and workflow changes to improve the system. Armed with a goal to replace the NICU system with something newer and better, and funding to do so, a development team was assembled to begin the process.

4.2 Writing NICU Stories

Although story writing was challenging for the previous project because of the large number of stories, and the large amount of detail needed, it still seemed like an effective way to drive the new project. Story writing began for the NICU project.

Initial discussion and requirements gathering on the new NICU system had taken place over many months prior to the introduction of an Agile development approach. Requirements fell into two categories: the general need to re-build most of the functionality of the NICU system as-is, and long lists of specific changes and improvements to make in the new system. In the old methodology it might have been acceptable to point to the massive legacy system and say "build it like this one," but in our Agile approach we needed to break that up into smaller, bite-size, stories. Given that, experts on the old system and the NICU processes, doctors and nurse practitioners, went to work decomposing the old system into user stories, and rolling changes and improvements in along the way.

4.3 No End in Sight

NICU story writing faced a number of challenges. The exiting Agile customers, while they were experienced nurses, were not strong experts on the NICU business processes. So, the Agile customer team collaborated with NICU nurses and doctors to help them write user stories.

Early NICU stories ended up being rather "large." By large we mean that based on the Agile customers' past experience, it would take developers far longer than an iteration to implement.

Although the stories weren't written in a tremendous amount of detail, it still took a long time to write these stories. The strong NICU domain experts engaged in this task often asked "Why am I doing this? I should be more of a resource to help others do this rather than doing it myself." And, indeed, in the prior methodology they had been such a resource that developers and medical informaticists used to gather information from. This Agile approach of having expert users drive the development was taking a lot of time, and participants were feeling it. Much of what was being written down was a restatement of what the existing system already did, or a description of an additional feature previously recorded elsewhere. Consolidating this information into written user stories was time consuming work.

After recording just the names of each candidate user story, our Agile customers and expert NICU users ended up with a list of 600-700 stories. Based on the few detailed stories already written, writing each story might take anywhere from a couple hours to a couple of days. Clearly we couldn't write details for all these stories since that would take thousands of hours to accomplish. We'd have to prioritize these stories and elaborate the most important early. The primary physician sponsor felt like we were "going back to square one – re-conveying all the requirements again. This Agile process is slowing us down – getting in the way of the development we already know we need to do." He and others on the project began to feel frustrated.

Prioritizing a list of this size was difficult. A first pass attempted to categorize stories as high, medium, and low priority – As, Bs, and Cs. However, we often ended up with stories of "A" priority being dependent on stories of "B" or "C" priority. The list of As was still a substantial list. So, it wasn't as simple as writing details about all the As.

As the customer team continued to ponder over the massive story list, time went by. Expert users continued to elaborate stories we were pretty sure we'd need soon. But, since many of the stories hadn't been elaborated, no development time estimates had been given. With a poor understanding of priority and development time, it was difficult to determine what might go into a first release, and when that release could be delivered. Sitting on a huge list of stories with no good understanding of where to start building or how long it might take we felt lost in the woods.

5. HOW CAN WE FIND THE FOREST IN THE TREES?

5.1 Using a User Centered Design Strategy

It was becoming clear that we were lost in details, that there was no clear place to start building this product. We needed a strategy for getting unstuck.

An approach I'd used before for requirements elicitation leveraged a number of different User Centered Design[2]

[2] The term User Centered System Design was first used in Norman & Draper's 1986 book of the same name [10]. Shortened to User Centered Design, the term refers to a class of design and testing approaches that leverage the understanding of the users of the software to make design decisions, leverage user observation and study to gain that understanding, and involve users collaboratively to prototype and test software.

techniques. Normally I might use these techniques to better understand the users and usage of software to arrive at a better understanding of what the scope should be. Our group already felt they understood what the requirements were; it was just this darn story writing, estimating, and release planning process that was holding us back. However, after talking through the basic progression of techniques used in a UCD approach, it looked like using a few of them might yield different results. And, at this point the group was willing to try anything.

5.2 User Centered Design Distilled

Garrett's Elements of User Experience Model [9] gives a high level view of the progression and dependency of activities used in UCD – see Figure 1.

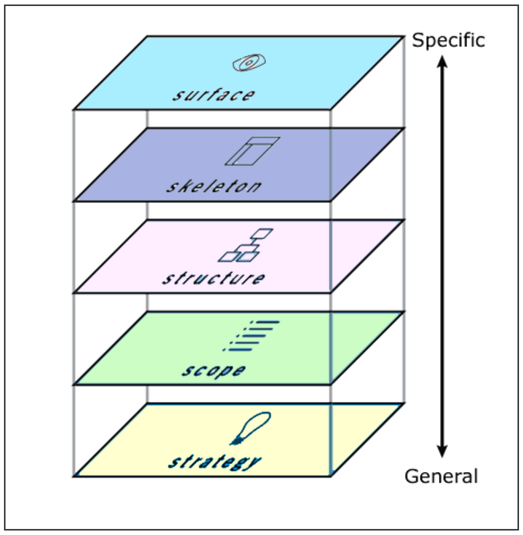

Figure 1: Garrett's Elements of User Experience

Garrett's model slices software into several planes of design activity from the most specific to the most general. The top layer, the surface, refers to the visual design we see when we look at a finished piece of software. The skeleton layer below refers to the screen or pages' internal design that supports user's workflow within the page. The structure layer below refers to the organization of software features into pages or screens and how the user might navigate from page to page. The scope layer below that refers to those features the software has and/or the activities the software supports. And finally, the strategy layer refers to the reasons we're building the software. More specifically: how the software earns return for its stakeholders, and what users it serves and their goals.

While we might describe software by peeling away layers from surface down to strategy, we design it by starting from strategy and working our way up through to the surface.

I find that in practice most software projects pay a fair bit of attention to the scope of the project without developing strong understanding and agreement on the strategies that motivate that scope. Arguments over scope are often rooted in disagreements on strategy.

At HCP we were having trouble prioritizing scope. That is to say that stakeholders believed it all should be built and released,. But was that true? And if we were to release part of the application first, what should that first part be? Possibly a better understanding of our strategies might help with this effort.

5.3 Understanding User Tasks

A common concept used in User Centered Design approaches is the *user task*. If I were a user in pursuit of a goal, tasks are what I'd do to reach that goal. For example, I have a goal to keep my teeth into my old age. Daily brushing and flossing are the tasks I do to try to achieve that goal. UCD approaches identify the users of their software and their goals, then seeks to identify the best set of tasks the users might engage in to meet their goals. That inventory of tasks is a great starting point for the scope of our software.

The user task concept used in UCD is similar to, and often interchangeable with, a use case. For instance the name of a user task might look just the same as a use case.

People often confuse the concept of task with a feature or design solution. For example in my task of "brush teeth" the feature my bathroom needs to support that task is a toothbrush – and maybe a cup to put it in. That toothbrush might be hard, medium, or soft, might be available in a variety of colors. We'll need to know those details before we build a toothbrush. The activity of determining those details is one of the design parts of UCD. Sometimes the design activity results in different solutions altogether – like WaterPik instead of toothbrush for instance.

Understanding user tasks is an important first step to designing solutions. When determining the general scope of a software product, a UCD approach might first have us identify the scope in terms of user tasks.

Looking back at the scope of our HCP NICU project it turned out that scope we'd discussed so far was a mixture of tasks and solutions. Our scope contained lots of toothbrushes – descriptions of features without any clear connection to the user tasks those features supported. How many items would we have in scope if we expressed it in terms of user tasks instead of developer sized user stories? At this point we didn't know, but we felt like it would be less. By identifying them as such we might have a more concise idea of what user activities the software needed to support.

5.4 Tasks and Goal Level

In Cockburn's Writing Effective Use Cases [4], he uses the concept of goal level to describe how use cases elaborate on or subsume each other. Since user tasks are like use cases, they also can be described in terms of goal level. Understanding goal level helps write user tasks, and more importantly controls how much detail to go into when doing so.

Cockburn describes goal level using an altitude metaphor (see Figure 2). Sea-level, in the middle of the model, describes functional level goals pursued by a user. You can think of a sea level activity as one you'd do in a single sitting without interruption. Brushing teeth is like that.

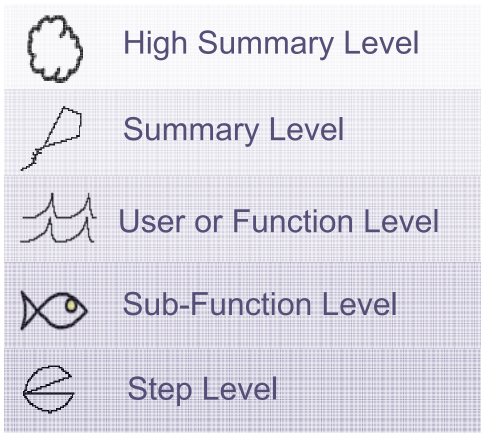

Figure 2: Cockburn's Goal Levels

Below sea level is the "fish" or sub-function level. To brush my teeth I use other steps like getting the toothpaste, opening the cap, squirting it on the toothbrush. A use case or user task titled open cap on toothpaste might be considered sub-function, or fish level.

Below fish level is the clam level used for the smaller steps within fish level task – like twisting the cap of the toothpaste counter-clockwise to open and clockwise to close. Usually explaining software in this much detail early is both speculative and time consuming.

Above sea-level is the kite or summary level. Brushing and flossing are part of my daily regimen of dental care. A use case called daily dental care might include steps like brushing and flossing. You can see that summary level goals might be accomplished over one or more sittings and are built out of lots of sea-level goals.

Above kite level is cloud level. You might describe your software and the world around it with a cloud level use case. These are good for executive summaries or short high level descriptions of the entire system.

5.5 Scoping the Project with User Tasks

I think of UCD's user tasks as sea and fish level activities. When articulating tasks for the NICU software we wanted to pay attention to the goal level to make sure we didn't get too low level. This would keep our number of scope items at a manageable level.

To make sure we were identifying as many of the tasks as possible, we needed to be clear about the users we were serving and their goals – since tasks were the thing they did to reach their goals. We also needed to string tasks together in possible workflow scenarios to validate that we hadn't forgotten about important tasks.

5.6 Contextual Observation

Contextual Observation refers to the idea of observing users performing their work in the environment where they normally would be performing it. Holtzblatt and Beyer's Contextual Design [3] uses contextual observation and inquiry as the cornerstone of its UCD approach. To really be sure we understood the people using our system, what their goals were, and what their current challenges were, we needed to spend some time in their environment. While much of the understanding and most of the requirements we had so far came directly from those users, our expert user customers and the developers hadn't spent much time in their environment. Contextual observation was a UCD technique we decided to use to better understand and validate our strategies with the NICU software.

6. APPLYING THE UCD STRATEGY

6.1 Collaborative Modeling Sessions – Taking a Step Back

It would be revisionist history to say we clearly saw our issues, and then determined that taking a UCD approach would solve our problems. What actually happened was something a little different.

The group working on requirements and story writing for the HCP NICU project routinely met for collaborative worksessions. During one of these sessions the group decided to get a little more information about User Centered Design approaches. On hearing more, the group then decided to extend a few more hours for a timeboxed activity of collaborative UCD modeling.

Over the course of the next few hours we discussed and noted the business level goals for the system, described the users who'd be using this system and their goals, then brainstormed user tasks they'd be engaged in. These tasks were arranged into a simple end to end workflow that expressed the business process across all users of the system.

To many of those involved, the process seemed like we were taking several steps backward in our requirements process. "We know so many detailed requirements today, what's the value in discussing these general ideas that were already commonly understood?" However, for some in the room who weren't doctors or nurses practicing in the NICU this was the first time they'd gotten a clear simple understanding of the scope of the project. "We could now see the whole process from beginning to end."

It took more discussion and a subsequent collaborative modeling session to arrive at list of about 80 user tasks. Managing 80 scope items felt a lot better than the story backlog that was edging toward 700.

Clearly we'd made a step forward in communicating within the group of Agile customers and NICU domain experts. More people shared some common understanding of what we were doing than before. But we were clear that we still had the same amount of software to build; that building that software was going to take a long time, and that we still couldn't decide exactly where and how to start.

6.2 On Site Contextual Observation and Hard Truth

Collaborative modeling sessions helped convey the big picture from the doctors and nurses intimately familiar with the processes in the NICU to the Agile customers and developers who'd be designing and building the software. But for the Agile team, we needed that last bit of understanding and empathy we'd gain through contextual observation.

It's usually an understatement to say that spending time with actual users engaged in their work is enlightening. Our team observed intelligent, competent doctors and nurses spending large amounts of time in front of medical records software documenting their work. They were extremely frustrated with the poor performance of the software. All had suggestions for features that would improve or streamline the documenting of their work. All agreed they spent far too much time in front of the software. And, all complained that they'd been promised relief in way of new software for a very long time.

It was true that requirements teams and business teams had been working for over a year to acquire funding for replacing the software, then working to determine exactly what that replacement might be. All the while the existing software users continued to struggle with the software knowing it could and should be better. Their frustration was visible and obvious. Understanding this raised the level of concern and urgency within the Agile team.

6.3 Identifying a Release Strategy

While task modeling helped us better understand the project scope, it was the understanding gained from observing and visiting with users in context that helped gel a release strategy. Our users needed relief sooner than later. It was critical that we get some part of the new system in place as soon as possible both to improve the quality of the user's daily work and to show them in a tangible way that progress was being made.

The focal business goal of improving the speed of user performance was set. By user performance we don't necessarily mean the speed of the system, but rather the time it takes users to perform their work using the system.

The NICU doctors and nurse's business was patient care. We couldn't deliver a fractional part of a working system, rather the system needed to support the work they were doing electronically today. No matter what we did, adversely affecting the quality of care, or taking necessary features away from the NICU staff wasn't possible.

Armed with the goal of increasing user performance and knowing that we should deliver some usable fraction of the system soon changed the way we looked at scope. Instead of deciding where to start building the software to re-automate the entire business process, we asked if we could keep the old software up and running. Then we sought to identify a high impact part of the business process that could be carved out and re-built in the new software. The result would have users working with both the legacy software *and* the new software together to accomplish their goals. The end blended result needed to be something that met that goal of increased user performance.

Using a UCD approach we'd identified one of our most important user constituencies as the doctors and nurse practitioners who documented their daily assessments of various problems their infant patients exhibited, and their plans for treating those problems.[3] This description of problems and documentation of the daily "assessments and plans" took place at patient admission, every day of the patient's stay, and at patient discharge. Replacing this bit of the software directly impacted this focal user constituency and the tasks they performed every day as part of their work. This bit of functionality we called Patient Problem Management, or PPM, represented about 30 of our 80 user tasks. This was what our first production release should be.

Now our initial 600-700 user stories had been distilled down to a product of 80 user tasks, with a first production release taking in 30 of those. We were seeing light at the end of the tunnel.

6.4 Building the Release Strategy Using a Span Plan

We'd decided our first production release should be patient problem management – PPM. We had only the user tasks to account for the scope. Clearly these user tasks were too big to fit our definition of user story – the definition that allowed each of them to be completed with a single iteration. But, that said, they were clear enough that experienced developers could give course grain estimates on them. They'd seen and built software similar to this, and they could read the user task and imagine sort of how it might look on the screen, then give rough estimates in weeks rather than days[4]. Using course grain estimates it was relatively simple to estimate the user tasks and doing so left us with a rough estimate that indicated it would take about 6 months for the work to be completed given our current team size.

Agile approaches prize frequent product delivery and for us six months wasn't frequent enough. We needed to chunk out this functionality into two or more releases that allowed us to see and evaluate the software earlier. An early internal release would allow us to validate our detailed design choices with our end users, test our architectural design for dependability and scalability, and react to anything we'd learned with additions or changes to scope.

To select a first set of functionality to release we knew we needed a set of tasks that would best represent the entire end-to-end business process. In their book Lean Software Development [13], Tom and Mary Poppendiek refer to this end-to-end business process as the *system span*. To find the system span we used a blended task modeling and planning technique referred to as span planning. [12]. The span plan arranges user tasks written on cards from left to right in order of dependency; then arranges tasks top to bottom in order of criticality to the business process. Tasks that must happen in the business process bubble to the top of the model. Tasks that must be completed before another task is started fall to the left and right of each other respectively. The result is an easy picture that both describes workflow, and by slicing the model left to right starting at the top and working to the bottom, suggests best possible groups of features to release as complete system spans.

[3] The process of reviewing subjective and objective data collected on a patient's problem, then writing an assessment and a plan for treating that problem is commonly done by physicians. The resulting document is often referred to as a SOAP note.

[4] It's interesting to note that while Agile projects today consider a story as something that can be completed within an iteration, in 2000 Extreme Programming Explained [1] defines a story as "one to five ideal programming weeks."

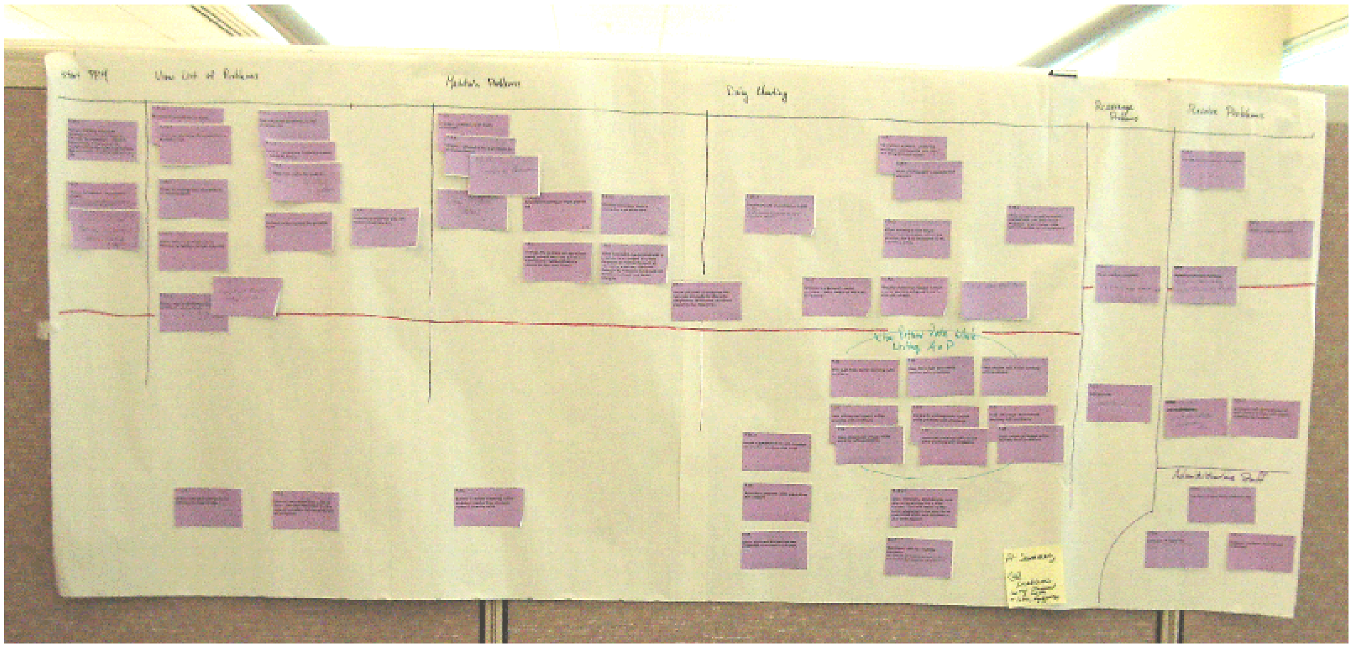

Figure 3: NICU PPM Span Plan hanging from cubicle walls in the Agile team area

Figure 3 shows the span plan used for the HCP's PPM system. The heavy line left to right in the middle of the model shows where we chose to divide the functionality into two candidate releases. The clever reader might note that there are more than 30 task cards in the model. Through more discussion some of the larger tasks were split into multiple smaller tasks. This is done by looking closer at the goal level. One sea-level task can easily be decomposed into multiple fish-level tasks.

The span plan was built collaboratively by the Agile customer team with some developer involvement. Building, and periodically rebuilding, the span plan not only helps release planning, but helps confirm and improve upon everyone's understanding of the business process.

6.5 Story Writing from a Span Plan

The span plan gave us a high level view of functionality covered in the release expressed in user tasks. But, like the brushing teeth example used earlier, at some point in time we need to decide that we need a toothbrush, and give our toothbrush maker an idea of how that might look. Our user tasks needed to be expressed as stories to release into our iterative development cycles.

Before, we didn't have a clear idea of where to start elaborating our user stories. Now the span plan gave us some guidance: start at the top left and work our way to the right and down.

For each user task we thought about the steps a user might likely take in the course of doing that task. With those steps in mind we constructed low fidelity paper prototypes for the user interface that might satisfy that task. We used a thought process best described in Constantine et al's paper: From Abstraction to Realization [6].

Thinking about Garrett's Elements model, we knew that tasks cluster themselves together in parts of the user interface. This clustering of tasks formed our applications structure. Knowing that, we layered support for multiple tasks into the screens we'd prototyped. Designing the screens to support the workflow of multiple tasks is the work done in the skeleton layer of Garrett's Elements model.

With simple UI prototypes for the screens that supported our most important tasks, we could begin to write user stories. The screen designs made that jump between task and tools to support it – brush teeth to toothbrush. Now we could write stories about these screens. Those simple story narratives paired with the low-fi UI prototypes gave developers what they needed to more accurately estimate and build the software.

6.6 Binding Stories to User Tasks

Each story we wrote for development was connected with at least one user task. Important to project managers was the idea that each story was connected to the course grain development estimate given to that task. As we built the software, we needed to be sure that the time estimates for stories didn't exceed, at least by too much, the initial estimates given to tasks.

To keep track of this we used a numbering system that made it easy. User tasks were numbered sequentially. Stories applicable to a task were given a decimal number relating back to the original task. For example stories written for task 10 were numbered 10.1, 10.2, 10.3, and so on. At any time it was easy to roll up story estimates and compare them to the original task estimate to see if we were going over budget. At any time we could see how many stories had been written against a particular user task.

Over time we began to refer to our user tasks as "*planning-grade"* stories. By this we meant stories small enough to plan with, but too large to build from.

7. REFLECTING ON THE APPROACH AND RESULTS

7.1 Less is More

HCP's old processes leveraged face to face communication in requirements sessions held during an initial requirements phase. With the change to an Agile approach and the addition of continuously involved people in the customer role, that face to face communication between the developers and expert users was lost – replaced by Agile customers. Although those in the customer role were experts, they needed to discover quick ways to acquire and retain the understanding of the time-constrained experts, then effectively communicate that understanding back to developers. Building artifacts like task models and in particular the span plan helped to do that.

Diving down early into writing user stories wasn't productive. The stories the expert users initially chose to elaborate actually didn't end up being the stories included in the first or even second prospective release.

Stories written at the level of granularity most useful for estimation and planning iterations seemed too granular for release planning.

Prioritizing detailed user stories was hard when the stories referred directly to the design solution. For example we might know that it was critical for the user to be able to accomplish a particular task, but if the story described a particularly elaborate way of doing it, was the story high or low priority? What happened when the task was high priority, but we needed a less elaborate and less expensive way for the user to accomplish that task? In those cases the story in the backlog was a distraction from what the user really needed to accomplish.

By working with user tasks, much of the design work had been deferred until later. This made for lots of hard work for the Agile customers to elaborate stories ahead of iterations.

Managing scope as user tasks did make many stakeholders uneasy. It seemed like we were losing important details.

Using fewer planning grade stories and having a clear release objective helped release planning proceed and productive story writing to begin.

7.2 Recommendations for Agile Projects

Based on our experience planning and writing stories for HCPs NICU project we'd make the following recommendations for Agile teams:

Clearly identify the business goals for a product release. Fewer goals are better than more goals. Measurable goals are better than intangible goals.

If you aren't a user of your software, spend time observing your users engaged in doing their work.

Use a course grain planning-grade story to initially scope your project. Using fewer stories allows you to more easily prioritize and chunk them into releases.

Good planning-grade stories might take the form of a UCD task or a Use Case written at a function or sub-function goal level.

Avoid premature elaboration. Choose to support the user task as part of release planning. Decide exactly how as part of detail story writing.

As important details and considerations emerge, capture and associate those user details with your user tasks – or planning-grade stories. As you write detailed stories, move those details from the planning grade story to the detailed story.

Use some visual model such as a span plan to give the development team and stakeholders a single high level view of the product you're working on.

Use paper prototyping to design and validate the design of user interface prior to describing it in a user story.

8. CONCLUSIONS

While the iterative development approaches found in Agile Software Development fulfill the promise of working software each iteration, the task of choosing which software to build first can be formidable. While simple guidelines like "choose features with the highest business value" may seem useful, what really has high business value may not be clear. On large projects, features described as user stories might easily number in the hundreds. Prioritizing such a list can be exhausting and frustrating.

In our project we found that adopting a strategy of writing fine-grain stories as we had in the past left us lost in the details – we couldn't see the forest for the trees. As a result we couldn't effectively prioritize work and construct a release plan. We couldn't effectively communicate to stakeholders how we could begin to release software to deal with their problems.

Adopting some techniques from a user centered design approach helped us see the forest in the trees. We clearly identified business goals, critical user constituencies and their goals, and built task models that helped us understand their workflow. Using user tasks as planning-grade user stories we were then better able to prioritize, estimate, and build a good candidate release plan. Using that release plan we were able to begin writing detailed user stories for the most important work.

9. ACKNOWLEDGEMENTS

Thanks to the hard work of the team at HCP including Jean, Larry, Tom, Tara, Becky, Sherri, Tiffany, and many others. Thanks to Robert Biddle for his encouragement and Alistair Cockburn for his discussion and support. Thanks to the team at ThoughtWorks for their support and participation. Thanks finally to Stacy for without her support I'd not finish a single thing.

10. REFERENCES

[1] Beck, K. *Extreme Programming Explained*. Addison-Wesley, 2000.

[2] Beck, K. *Test Driven Development: By Example*. Addison-Wesley, 2002.

[3] Beyer, H. and Holtzblatt, K. *Contextual Design: A Customer-Centered Approach to Systems Designs*. Morgan Kaufmann, 1997.

[4] Cockburn, A. Writing Effective Use Cases. Addison-Wesley, 2000.

[5] Cockburn, A. *Agile Software Development.* Addison-Wesley, 2001.

[6] Constantine, L., Windl, H., Noble, J. and Lockwood, L. *From Abstraction to Realization: Abstract Prototypes Based on Canonical Components.* ForUse Website, July 2003, http://www.foruse.com/articles/canonical.pdf

[7] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] Fowler, M. *Patterns for Enterprise Application Architecture*. Addison-Wesley, 2002.

[9] Garret, J. J. *The Elements of User Experience: User-Centered for the Web*. New Riders Press, 2002.

[10] Norman, D. and Draper, S. *User Centered System Design: New Perspectives on Human-Computer Interaction*. Lawrence Erlbaum Associates, 1986.

[11] Palmer, S. and Felsing, J. A Practical Guide to Feature Driven Development. Prentice Hall, 2002.

[12] Patton, J. *It's All in How You Slice It*. Better Software Magazine, January 2005, http://www.abstractics.com/papers/HowYouSliceIt.pdf

[13] Poppendiek, M. and Poppendiek, T. *Lean Software Development: AnAgile Toolkit for Software Development Managers*. Addison-Wesley, 2003.

[14] Schwaber, K. and Beedle, M. *Agile Software Development with SCRUM*. Prentice Hall, 2001.