

User-Centered Agile Methods

Hugh Beyer
InContext Design
www.InContextDesign.com

SYNTHESIS LECTURES ON HUMAN-CENTERED INFORMATICS #10



MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

With the introduction and popularization of Agile methods of software development, existing relationships and working agreements between user experience groups and developers are being disrupted. Agile methods introduce new concepts: the Product Owner, the Customer (but not the user), short iterations, User Stories. Where do UX professionals fit in this new world? Agile methods also bring a new mindset—no big design, no specifications, minimal planning—which conflict with the needs of UX design.

This lecture discusses the key elements of Agile for the UX community and describes strategies UX people can use to contribute effectively in an Agile team, overcome key weaknesses in Agile methods as typically implemented, and produce a more robust process and more successful designs. We present a process combining the best practices of Contextual Design, a leading approach to user-centered design, with those of Agile development.

KEYWORDS

agile, agile development, scrum, xp, extreme programming, usability, usability engineering, HCI, UX, user experience, user-centered design, customer-centered design, human-centered design, iterative design, design, design methods, methodology, user interface design, user research, project management, user experience project management, human factors, prototyping, contextual inquiry, contextual design, user data collection, interactive design

Contents

1	Introduction	1
2	Common Agile Methods	3
	2.1 Scrum	3
	2.2 XP	6
3	Agile Culture	9
	3.1 There is Only One Team	9
	3.2 The User Is on the Team	10
	3.3 Plans and Architectural Work are a Waste of Time	10
	3.4 Face-to-Face Communication is Better Than Documentation	12
	3.5 Short Sprints are Good. Shorter Sprints are Better	12
	3.6 Continual Feedback Guides the Project	14
	3.7 How These Values Go Wrong in Practice	14
4	Best Practices for Integrating UX with Agile	17
	4.1 Get User Feedback from Real Users in Context	17
	4.2 A Phase 0 to Define System Scope and Structure	19
	4.3 UI Design Done One Iteration Ahead	21
	4.4 Validation Done One Iteration Behind	22
	4.5 Parallel UX Stream	22
	4.6 Programmer/Designer Holiday	23
	4.7 Architectural Spikes for Difficult Issues	24
	4.8 UX as a Full Team Member	24
5	Structure of a User-Centered Agile Process	27
	5.1 Phase 0: Project Definition	27

x CONTENTS

5.1.1	Contextual Inquiry and Affinity Diagrams	28
5.1.2	The Affinity Diagram	30
5.1.3	Work Modeling	30
5.1.4	Personas	33
5.1.5	Visioning	34
5.1.6	Storyboards	36
5.1.7	User Environment Design	38
5.1.8	Paper Prototyping	39
5.2	The Release Planning Session	41
5.2.1	Writing User Stories	41
5.2.2	Estimating Cost	42
5.2.3	Planning the Release	43
5.3	Running Sprints	44
5.3.1	Sprint Planning	45
5.3.2	Working with Development	46
5.3.3	Sprint Interview Preparation	46
5.3.4	The Sprint Interview	47
5.3.5	The Interpretation Session	47
6	Structuring Projects	49
6.1	Jumping on a Moving Train	49
6.2	System Extension	50
6.3	Major New Release	51
7	Conclusion	55
	Bibliography	57
	Author's Biography	61

Best Practices for Integrating UX with Agile

Fundamentally, agile development comes down to this: develop in short iterations and test progress at each iteration with real user feedback. However, getting that real user feedback is not possible without the skills and techniques developed by the UX community. Integrating UX with agile development is not only possible, it is critical to the success of agile methods.

Already, we are seeing best practices emerge for such a combined process. These practices are being implemented in agile companies across the industry and provide a baseline for a workable development process. Many of these practices are identified in Martin, Biddle and Noble's survey¹¹ of agile teams across multiple companies. They reveal elements that must be included in a successful integrated approach. These best practices enable a team to address the potential weaknesses of agile development directly by:

- Providing space for understanding the user and envisioning a coherent solution;
- Collecting real, end-user feedback during sprints;
- Supporting real iteration—reworking the design in response to user feedback.

4.1 GET USER FEEDBACK FROM REAL USERS IN CONTEXT

For reasons we discuss below, the experience of the UX community shows there is no substitute for talking to the actual end-users of a proposed product or system directly, in their own workplace. Originally, agile methods tended to finesse this need by specifying a customer or product owner role and leaving it undefined how the role was to function. This bias came from the history of some early agile projects. They were internal systems for internal users, and it was in fact not too difficult to walk over to a user, sit down beside them, and talk about how the system should be changed.

But, on most projects, it is no longer so easy. It is important to for the organization to recognize that there are real problems getting in the way of good user feedback and to recognize the limitations of the mechanisms that projects often use. UX people need to know how to talk to the organization about the need to augment existing methods with user research. The following points are familiar to UX professionals, but we have found they can be new to even quite senior agile developers. Goguen¹², Greenbaum and Kyng¹³, and Wixon and Ramey¹⁴ discuss these and other issues in collecting good user feedback in more depth.

18 4. BEST PRACTICES FOR INTEGRATING UX WITH AGILE

Users are not good at articulating what they do. This is the problem of *tacit knowledge*: users have internalized the details of how they work. When asked questions about what they do, what problems they have, and what they need, the details of their own work are hard for them to recall. So the requirements they give are inaccurate and incomplete.

Users want to be helpful. Agile methods attempt to overcome the problem of tacit knowledge by delivering early and often. When users attempt to use the system to do their own work, they discover and report on problems.

But users tend to assume that to be helpful, they should respond to the system they were given. They may report on some minor inconvenience in that system; they are much less likely to explain that what they really need is another system entirely, or that the bulk of their work is spent on another task, or that what they are really doing is collecting and preparing all their data in a spreadsheet and then using the designed system just for data entry.

Users are not available as team members. Real users have real jobs. It is not reasonable to expect them to attend a daily meeting or to be interrupted every minute for questions about the details of a UI. Often, a business organization will insulate users from persistent developers by appointing user representatives to act as an interface. But the more they interface with developers, the less they do the real job. Some organizations establish business analysts, but generally, business analysts focus on defining business rules, processes, and data elements. They do not usually have interaction design skills. When making a product for a market, users are not even in the same company. And, of course, whether internal or external, users are often not located anywhere near the development team.

Surrogate users aren't. Product development organizations typically handle this problem either by putting surrogate users on the team or using marketing methods to find out more about them. There is real value in both methods, but by themselves, they are incomplete.

Surrogate users are never a good stand-in for real end-users. The managers of end-users describe how they wish the work was done, or how they think it ought to be done, not how it actually is done in their organizations. A new system may intentionally change and simplify the work practice, but if the team does not understand the real issues and real workarounds—which are often invisible to management—they cannot account for them in the new design.

Hiring users away from their organizations to be on the team full time has been tried but is also of limited value. Intentionally or not, they tend to take on the perspectives and values of the development team. “We found out our users representatives were being too nice to us,” was one team’s comment that tried this¹⁵.

Product owners as defined by Scrum also do not make good user surrogates. They may be responsible for representing all the stakeholders of a system, including end-users, the customer who makes the purchase decision, and the internal stakeholders. But they are not any of these people and need mechanisms for understanding them and their needs just as much as any other project team member.

Marketing methods do not collect design data. Product development organizations attempt to use marketing methods such as surveys and focus groups to define products. But these methods provide high-level information about attitudes and desires. They do not reveal how users do work, define what system they need, or what such a system should do. Neither do they provide low-level detail about the requirements of system function and behavior such as how a task is structured, interactions within a work group, detailed steps users perform, how they collaborate in accomplishing a task, or the context and constraints on how the work is done. Marketing methods are good at collecting sales points and market requirements; they do not collect design data.

So, any effective user-centered agile process must include real user research: finding out who the end-users are and how they work; analyzing the tasks they do and the strategies they use to achieve those tasks; getting quick feedback on design ideas and on system baselevels to determine whether the project is on track; and testing designs against success criteria. The process must also support discovering the requirements of the users' management and purchasers and of internal business stakeholders.

Online data collection is incomplete. In this age of web-enabled apps, it is easy to instrument the apps themselves and use the data collected to guide development. Different versions of a design can be posted and usage data compared; click-throughs and pause times can be measured. Quick surveys can be popped up to collect data in the moment.

But although this kind of data may usefully augment field research, it is limited in the same way as more traditional market research methods. A survey can gather data on questions the designers thought to ask and that the user is aware enough of to respond to—but much of work practice is tacit. An instrumented app can report on what the users did but not why, or what they were trying to accomplish, or what the larger work task was. Any data collected this way will start from the assumption that it is the right app. The team will not discover that there are other opportunities they do not know about.

4.2 A PHASE 0 TO DEFINE SYSTEM SCOPE AND STRUCTURE

Much of the agile community, especially from the coding side, is driven by a strong skepticism of any sort of up-front planning. “Big Design Up Front” (BDUF), they call it. Also: YAGNI (“You Ain’t Gonna Need It”), meaning that most of the big plans you have in your head will never turn out to be useful or relevant once you get to the point of implementing them. Business direction will change, user requirements will change, or the evolution of the design itself will make your idea irrelevant.

But successful projects that have significant user interfaces and have a significant impact on how users work have found that some level of up-front design is necessary¹⁶. (The Martin, Biddle, and Noble paper¹¹ refers to this as BPUF—Big Picture Up Front.)

Practically, projects discover the need for BPUF as soon as they sit down to write story cards. Suppose a team is developing an online newsreader. Think about the questions that must be

answered before a story card can be written: What are the possible news sources? How should they be represented to the user? How should they be organized? Can the system organize them or must the user be able to, or both? What should the first screen contain? Should it show all new news items, or should it prioritize, and if so how? The questions are almost endless.

And each story card the team writes captures an element of a design that they never thought through, never represented in any concrete way, and which impacts the user's world in ways that are completely implicit and undesigned.

In theory, any problems will be resolved through iterations with the user. That is what agile is all about: keep iterating and fixing until the product works.

In fact, these problems cannot be entirely solved through iterations. As mentioned above, even in the best possible scenario, users will only renovate on what they are given. Unless a system is completely hopeless, they will not throw it out and tell the developers to start again with something different. So only about 20% of the overall system can be changed. The base assumptions and core structure are simply not amenable to stepwise change *within practical product timelines*. To counter a common argument heard in agile circles: it is true that evolution produced humans from single-cell proto-organisms using nothing but stepwise refinement. But evolution had millions of years to do it. Product teams do not.

And in the real world schedule limitations will be determinative. Any engineering project will only tolerate reworking the same design element so many times. Any product or business manager will only tolerate so much deviation from the initial project scope. Most agile projects struggle to get any rework time into their schedule at all. Once a user story is implemented, there is *no* time to go back and rework it in the project plan. The rework could be written into a new story and reprioritized at the next sprint planning meeting, but then some other committed story would go unimplemented. Unless the problem is a real show-stopper, that reprioritization is unlikely to happen.

Furthermore, iterating an existing solution will inevitably focus on fixing problems with that solution. What if the real opportunity is not in fixing problems with the current solution but recognizing that a different approach is needed? For example, the spreadsheet was invented when Dan Bricklin recognized that computer technology could be applied to the problem of manipulating accounting (paper) spreadsheets. Calculator technology existed at the time, but no amount of stepwise iteration would be likely to lead from number manipulation to an electronic spreadsheet.

If a team gets past writing user stories without any big-picture design, the next point where they will feel the lack of it is during sprints. In theory, a developer can go to the user and ask detailed questions about how a story should be implemented. In practice, this theoretical user is a product owner or a UX designer.

The programmer might ask: "What is more important to show: breaking news, or a story the user is actively following? What should be pushed to the top of the page?" If our user surrogate were honest, the answer would likely be, "Darned if I know." Unless they have worked with users in the field, how can they possibly answer such a detailed question correctly? The actual answer they give is usually something like, "Do it this way, that should work." This is a guess, dressed up in professional

language. Without a solid understanding of the users to back it up, one of agile development's core drivers—immediate feedback—is broken.

Instead, problems with the design for a user story are not discovered until the story is implemented, built into an iteration, and shown to users. Even if the problem is discovered, and often it is not, a great deal of work has already gone into a wrong implementation. This rework is unnecessary.

Finally, but equally important in this day of sophisticated user interfaces, designing a coherent and consistent user experience across the whole product means that the entire UI must be designed together. If one part is changed, the impact of the changes on the whole system must be considered. This is very different from implementation design, so developers tend not to appreciate how important it is. When designing code, the goal is to separate each module from every other, so that each module of code can be modified and reimplemented without affecting any other module. Sometimes, this separation breaks down, and the relationship between modules has to be re-thought and several modules re-implemented together. This *code refactoring* is considered a major task, more difficult than ordinary coding.

For all these reasons, an effective agile process will make room for some high-level, up-front user research and high-level design, tested and iterated with users. This is often called a *phase 0*, or sprint 0. (We prefer the former name, as it better communicates the scope of the work.) User research grounds the agile customer team in the real user work practice, culture, goals, strategies, and issues. The high-level design sketches a systemic response to the user work practice, within the defined scope for the project, meeting the business needs of the organization. It also defines the groundwork for the user experience, defining consistent layout and interaction paradigms across the system.

This design, itself, even though high-level and provisional in its details, needs to be tested and iterated with customers. This realizes another core agile value discussed above: reduce the time lag between when a decision is made and when it is validated⁴. With a validated high-level design in hand, user stories can be written in confidence that they make sense and will deliver a system the user wants. We will discuss the structure of a phase 0 in Chapter 5, below.

4.3 UI DESIGN DONE ONE ITERATION AHEAD

It is hard for developers to appreciate how much work goes into a good UI design. Accordingly, processes designed by and for developers—as most agile processes are—often do not allow enough time for UX work to be done. Many agile proponents do not understand why it is hard to design the UI for a component, test it, do the graphic rendering, and communicate it to the developer, all within a few days, while still leaving enough time for the component to be coded, integrated, and tested within a two-week sprint.

Successful agile teams usually give the UX designers room to breathe by having them start on a component design one or more sprints ahead of coding. This way the UX team gets the whole of a sprint to do their UI design and user testing. The testing can and should be iterative with

22 4. BEST PRACTICES FOR INTEGRATING UX WITH AGILE

users—which means that the resulting design is refined with users, even if there is little time to rework stories in the development process.

Then in the following sprint, the UX team member can communicate the detailed UI to the developer, who gets the whole of that sprint to do the implementation. Meanwhile, the UX designers, in parallel, start on the UI design of stories for the iteration after that¹⁷.

4.4 VALIDATION DONE ONE ITERATION BEHIND

Agile methods tend to be structured as though the last thing that has to happen is integration testing. Code a story, unit test it, integrate it into the code base, show it to the product owner, and it is done. Quality Assurance (QA) or User Acceptance Testing (UAT) is given a role in the process, but that role is limited by the short sprints. Generally, QA can do a reasonable job within the iterations by working closely and incrementally with developers as stories are checked in; generally, user acceptance testing cannot. A short presentation at the end of the sprint (as described, for example by Schwaber¹⁸) is *not* a realistic acceptance test.

Doing real user testing necessitates, first, bringing real users in or going out to them. It requires walking through the product with the user's own real-world examples. It requires that all the stories for the iteration be completed so that the interaction between new features can be investigated. It demands time to understand the implications of the users' feedback, which is always more complex than pass/fail. And it requires time to rework the parts of the system that have problems.

Realistically, it is very hard to do all this within an already-short sprint without making the iteration too short to do useful work. Even if such testing were possible, it does not require the whole team, and they need something to do while it is going on.

So, successful agile projects have found it effective to do their real testing with users one sprint behind the implementation. The sprint completes and the internal QA and UAT processes confirm that, to the best of the project team's ability to determine, they have correctly implemented the stories of that sprint. Then, during the following sprint, UX people can take that baselevel to customers, walk through real examples with it, and bring back problems and fixes to be addressed in the following sprint.

This practice also supports large-scale agile projects. In such efforts, the product of an agile team is, itself, just a component in a larger system. Time is needed to integrate the components from all the development teams and do integration testing. All this work can also be easily done one iteration behind¹⁹.

4.5 PARALLEL UX STREAM

As mentioned above, ensuring the user experience of a product is coherent and consistent is a full-time job. Unlike refactoring, which is a special case, the UX team *always* needs to be concerned with overall consistency. Furthermore, there are usually larger constraints on the UI of the product. An individual agile team may be only implementing one part of a larger system. It may be implementing

one product in a suite or one component of a product. The UI for their team part, must be coherent within itself and also within the larger product or suite. It must also conform with whatever standards or guidelines the company mandates.

So, the UX team needs to maintain a larger focus than just the product delivered by the team. They are always looking at both the design of individual screens and at the larger system they are a part of. Maintaining this wider focus is very hard in the time-compressed world of agile development. There is barely enough time to get the tasks of the sprint done; looking at larger architectural issues is usually the first thing to be traded off.

To make sure the wider focus is not lost, many organizations find it useful to create a separate UX stream. This stream is not part of any agile development stream—it runs in parallel to them, but synchronizes with them at key points. This stream addresses inconsistencies across the whole system and feeds fixes to those inconsistencies back into the agile development streams as new stories. This UX stream can do additional user research, design, and iteration without being tied to the particular schedule and deliverables of any one development stream. This stream maintains the whole-system coherence of the design.

4.6 PROGRAMMER/DESIGNER HOLIDAY

One of the best practices Martin, Biddle, and Noble¹¹ identified in the agile world is the idea of a *programmer's holiday*. Agile experts recognize that there is never enough time during focused, intense sprints to do all the architectural work and refactoring necessary to keep the code base clean and maintainable. Technical debt tends to build up over time despite the team's best efforts. They also recognize that the pace demanded by short sprints becomes exhausting over time. When you eliminate the natural down time created by project planning and field test, there is just no letup in the pace of project work.

So a programmer's holiday dedicates an iteration to code cleanup. Each developer can choose what to work on, whether refactoring a badly-implemented module, addressing bugs, or doing large-scale restructuring that affects whole subsystems.

Such a holiday can be useful to the UX team as well, especially if there is no parallel UX stream. This can be an opportunity to step back and look at the overall coherence of the UI: whether the basic structure holds up and whether it is appropriate to the user tasks. If there is a need for large-scale restructuring of the UI, affecting several screens all at the same time, this is an opportunity to do that kind of cleanup.

A team working in two-week sprints might schedule such a holiday once a quarter or so, often enough to stay on top of things but not so often that it interrupts development significantly. A team working in one-month sprints might devote half a sprint to the holiday, committing to half the normal amount of work for that sprint.

4.7 ARCHITECTURAL SPIKES FOR DIFFICULT ISSUES

Architectural spikes allow the development team to address a difficult or risky development problem. They might devote a whole sprint to studying a technology problem, prototyping alternative solutions, running load tests, and making sure that they have understood the risks and settled on an optimal solution. Such spikes happen early in development, to get the most important project risks out of the way early.

But not all project risks come from implementation concerns. Challenging UI problems may well pose as much risk to the success of a product. Consider Microsoft's ribbon interface, for example, new with Office 7. Would the ribbon be intuitive for new users? Would it be frustrating for experienced users? Could it present all the many functions of an Office product coherently? Would users get lost navigating multiple ribbons?

Use architectural spikes to address UI problems, too. When a team identifies a UI problem as a key risk, it is appropriate to devote an iteration to addressing that risk. Remember, the risk is the whole team's problem. The team can use the time to brainstorm alternatives, mock them up, and test them with users to verify that they actually work as implemented. This can be critical: not only did the ribbon have to work intuitively in theory, it had to be possible to actually put the Office products' real functions into the ribbon in a sensible way.

4.8 UX AS A FULL TEAM MEMBER

The discussion to this point has tended to treat the UX designers as separate from the development team. But one key agile principle is that everyone is on the team, and the team is co-located. Being on the team means that every team member is responsible for the success of the whole product. Everyone, including the UX designers, is a pig, not a chicken. It means that they participate in the work of implementing user stories and they are present for the daily meetings. It also means that if they are stuck or behind, it is the team's problem, not just theirs.

This principle creates a tight, cohesive team when all the team members are developers. When some team members—like UX designers—have very different skill sets, it is less obvious how to make it work well. It is easy to move around developers to address coding problems (and agile principles such as “no code ownership” are intended to make it still easier). Having a UX person help with a coding problem or having a developer help with a UI problem is more problematic.

It is also common for UX designers to be shared across multiple projects. Ideally there would be one or more UX people assigned to a project, and when the user interface is a success factor for a product, this should absolutely be the case. In the real world, the UX professional is usually juggling the demands of several projects at once.

The UX part of the organization also has its own focus that transcends individual projects: for example, developing common UI standards across the company, ensuring that the company's branding and image comes through in product UIs, ensuring that the company's UX professionals stay up to speed on the latest developments, and cross-fertilizing across UX professionals. A UX

professional always has a dual allegiance: to the team they are on and to the UX community they are a part of.

Successful UX team members act like full team members. They are co-located with the team they support if at all possible. If they support multiple teams, they have a desk in each team's development room, and they do the work there as much as possible. They are present for the daily meetings unless they have a conflict; when multiple teams have their meetings at the same time, they choose which meeting to attend based on what is happening in each project. They do work based on story cards, generally, as UX tasks associated with a particular story.

As full team members, UX people can draw on the team's resources. Just as a programmer can say, "I'll need to work with a database expert for this story," the UX person can say, "I'll need someone to come with me on this paper prototyping customer visit," or even, "This is a simple, non-critical UI. Can we agree the developer will design something basic which I can just look at as a sanity check, while I spend my time on this other critical UI task?"

CHAPTER 5

Structure of a User-Centered Agile Process

Given the above best practices and what is known to work in both the user-centered and agile domains, it is possible to construct an overall approach to design and development that makes agile development truly user-centered—or, conversely, that makes user-centered development agile.

This approach makes room for coherent design by preceding agile development with a phase 0 for user research and user experience design. With a sound understanding of the user under their belt, the team can then write good user stories and go into agile development sprints confident that they know what problem they are solving and have an initial direction for solving it.

In the rest of this section, we will show how each of these parts of development can be structured to include both the UX perspective and the user iterations on which agile methods depend. For user-centered techniques, we rely on Contextual Design (CD), co-developed by the author and Karen Holtzblatt²⁰. For agile techniques, we draw on the agile best practices described in Chapter 4 above and include concepts from both Scrum and XP.

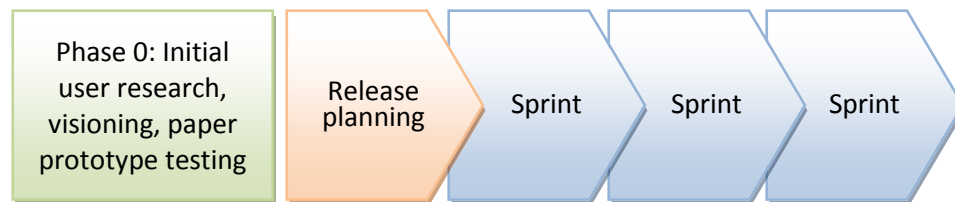


Figure 5.1: The basic structure of agile development. Coding starts when the basic goal and structure of the project is set in phase 0. This phase may be anywhere from 2-8 weeks, depending on the scope of the project.

5.1 PHASE 0: PROJECT DEFINITION

Few projects start with a completely blank slate. Existing products and systems provide the context for any new development. An existing product may need an additional feature or may need to support an additional role. The competition may have come out with a new feature, and the company may be scrambling to catch up. An internal system may need to support a redefined business process. A

business may be struggling with too many standalone tools and may be looking to integrate them. (See Section 6.3 for a discussion of how to deal with larger scopes—new product development and large-scale projects.)

In each case, development starts with a general idea of the scope of the system, but so far, with no detailed agreement on the specific tasks to be supported, how those tasks are to be supported, or what the requirements on the system are. The team is not yet ready to write user stories—there is not yet agreement on what they should say, and the customer team do not yet know enough about the user to provide trustworthy guidance.

Starting the project with a phase 0 gives the team a chance to organize themselves, find out about their users, and understand the kind of solution they should build. This is the process for building the BPUF—the Big Picture Up Front—which becomes the basis for iteration during sprints.

5.1.1 CONTEXTUAL INQUIRY AND AFFINITY DIAGRAMS

The first step is to discover how users approach their work. This study of *work practice*—how people structure and perform the jobs they do—is essential to designing effective systems^{21;22;23}. And the best practice for understanding work practice is Contextual Inquiry (CI).

In a Contextual Inquiry, project team members interview users in their workplaces, watching them work and talking to them about what they are doing and why. If the project is building a consumer product, the team interviews consumers in their home, cars, stores, or other life contexts, observing the life task the new product will address. At this point, the primary focus is not on design at all. The focus is on how people perform the work, what they are trying to accomplish, how they go about it, and what gets in their way.

Any kind of field research needs some training, but that is no reason for limiting this work to UX professionals. Developers and other team members can be trained to assist, and even untrained engineers can go along as observers. This supports the agile “one team” value, promotes cross-fertilization across team members, and ensures all team members understand the users’ problems at a visceral level.

The contextual interview. The first principle of CI is to observe the work directly, as the user works. For traditional office work, this means going to the user’s office and sitting with them, watching the user interact with online systems and with paper files and forms. The interviewer can see the informal notes and cheat sheets the user has created to help track the job and the piles the user creates to organize and stage a job. He or she can see the use of Excel to do the real organization and calculation before loading up the results into the official tool. Interruptions and informal communication happen while the interviewer is there, revealing tacit aspects of the work that might otherwise remain unarticulated. If the task requires moving around, the interviewer goes with the user, running up and down halls, driving to remote sites, crawling through access tunnels, or riding bicycles to get around a huge assembly plant (all real examples, by the way).

If the design is for a consumer product, the interview is conducted wherever the life task is performed: in the home or car, in public, while commuting or shopping.

Nor is the interviewer constrained to only the events that occur during the interview. *Retrospective accounts* give the interviewer a way to recover detailed task information about events in the recent past. Together, the interviewer and user replay a specific event of interest, using artifacts and probing questions to reveal the detailed steps of the task. In this way, the interviewer can learn about and capture the details of important tasks and situations whenever they happened, as long as it is within the recent past.

Throughout the interview, interviewer and user engage in a discussion of what the user is doing and why, and what the implications are for the project's design direction. These discussions reveal the user's intents and strategies (more tacit knowledge) that underlie their actions.

Note that the interviewer does not focus on problems and issues in the work, though those are certainly revealed in the interview. Unlike a usability test, which explicitly focuses on identifying problems, the goal of a CI is to understand the whole work practice. This understanding allows the team to envision solutions that transform the user's work rather than merely fix existing problems. For example, CIs of work practice in offices during the 1990s revealed that users tended to collect phone, calendar, rolodex, and day-timer style personal organizer together on their desks. This implied that these tasks—communication, identifying contacts, and organizing schedule and tasks—went together in the work practice. They made a natural whole. This recognition anticipated by years the introduction of combined email, address books, calendars, and task organizers in products such as Outlook.

CIs give interviewers the understanding of the users they need to represent users on the team: knowledge of the users' tasks and needs and, critically, the gut feel for what design alternatives will work for this population and what will not. Internalized knowledge and gut feel are valuable, but knowledge has to be captured and externalized if it is to be the basis of design. That is done in *interpretation sessions*.

The interpretation session. Data from any sort of field research tends to be unwieldy. The interviewer's notes are lengthy and unstructured, transcripts are difficult to create and hard to understand, and videos, if used, are cumbersome to manipulate. Interpretation sessions give the team a way to deal with field data.

In an interpretation session, the team reviews each interview by going through the interviewer's notes of the interview in detail. The interviewer retells the story of the interview from beginning to end, in order. The rest of the team captures key information which has design relevance. Individual observations are captured in a list to be printed later on sticky notes and used in the affinity. The work practice of the user is captured in work models (described in Section 5.1.3 below). By the end of the interpretation session, all important information from this user is written down and ready to be used.

From the agile perspective, it important to note that these notes (and, indeed, the affinity and work models as well) are used by the agile customer team to understand the customer. This is not an

30 5. STRUCTURE OF A USER-CENTERED AGILE PROCESS

example of documentation created by one group to be passed off and consumed by another, which would be anathema to an agile team. This process enables the agile customer to understand who the user really is, what their work practice is, and consequently, what the project might do to help them.

U02-20	U02 ensures that everyone in the house knows that she is making her shopping list and that they had better get their requests to her
U04-15	Husband and wife share responsibility for tracking household needs
U07-18	U07 likes specialty stores because they carry things he can't find elsewhere, but sometimes they don't have enough turnover to keep goods from getting stale

Figure 5.2: Sample affinity notes.

5.1.2 THE AFFINITY DIAGRAM

So far, the team has focused on each user individually. But products support whole markets, not individual users. Even internal systems support roles and job positions, not the individuals who happen to be doing those jobs at the moment. So it is important for the team to be able to see the common structure of work practice, independent of individual users without losing the variation that exists across users.

Affinity diagrams have come to be widely used as a method for organizing large amounts of unstructured data. An affinity diagram is built from the user data collected through CI and it reveals key issues across all users studied: key elements of the work practice, user objectives and how they are achieved, pain points, workarounds, tools used, and so forth. The team builds an affinity from their observations after interviewing users.

An affinity is built from the bottom up by first grouping similar observations, labeling them, then building larger groups out of these small groups. It is not built by starting with large classifications or by sorting notes into predefined categories. The result is that the structure of an affinity reflects the weight of the data, with less influence coming from the team's preconceived ideas. This is essential to push insight; the team discovers new ideas and perspectives by how the data comes together.

An affinity can be built by the team in one or two days.

5.1.3 WORK MODELING

The affinity collects and organizes issues across all users studied, but it does not show the structure of work practice coherently. There is no place on the affinity wall that does a task analysis, for example—that shows the steps users perform to accomplish a task. There is no organized representation of the physical environment where the work happens, or the artifacts people use to accomplish it.

Work models are the appropriate tool to show the structure of work practice. They drive design thinking by suggesting how different design approaches will or will not support the work. Work

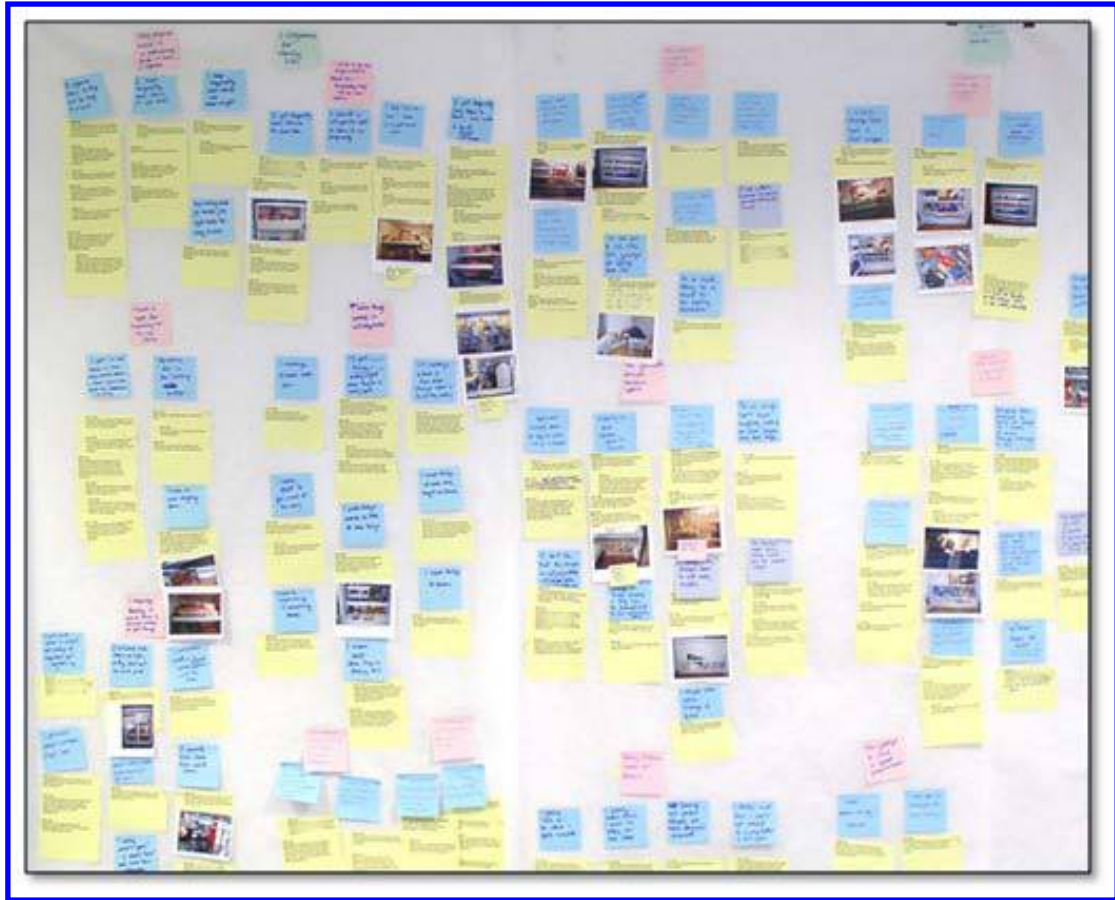


Figure 5.3: A section of an affinity showing how individual observations (the yellow stickies) are grouped to reveal design implications. Photographs taken during interviews may also be incorporated into the affinity.

models are built during the interpretation sessions from the data collected in Contextual Inquiries. Like the affinity, they are consolidated across users to show the common structure of work.

Contextual Design defines five types of work models, but phase 0 for an agile project usually depends primarily on sequence models. That is because the larger context of the project has already been defined through marketing research, business process redesign, or existing systems that are being revised. Models that help to understand the larger context are not needed in a more focused project. (If your project is not so focused, *Contextual Design*²⁰ describes all the models and how to do concepting and ideation in a user-centered way.)

32 5. STRUCTURE OF A USER-CENTERED AGILE PROCESS

The sequence model. Specific tasks are analyzed and represented with sequence models, and nearly every project will want to build them. Sequence models show the common structure of each relevant task users perform. They show the high-level structure of the task, the different strategies used, intents the users are trying to achieve, and the problems getting in the way of the work.

Sequence models are built from current work practice, but their importance is precisely because they help keep designers from getting stuck in the current way of doing things. Incremental development runs the risk of focusing on incremental fixes, eliminating pain points one by one without ever looking at the overall structure of the task or thinking about a radical redesign of how it is supported.

Sequence models show the many tedious steps required to achieve a user's intent. They show how different strategies imply different intents and suggest how a different approach in the system would allow users to achieve their intents more directly. They show how whole tasks have been created merely to overcome system constraints and suggest how those tasks might be eliminated. By looking at the whole task together, they move the team from thinking about point fixes to systemic solutions.

Sequence 1: Monitoring incoming customer email

U1-1-1 Intent: Be sure priority customers and sales opportunities are handled quickly

U1-1-2 Trigger: Realize that mail hasn't been checked over an hour

U1-1-3 Go to tech service mail box and scan the senders and subjects

U1-1-4 Set priorities for how to work through the email

U1-1-5 See a request for a recommendation for replacement part

Figure 5.4: A partial sequence model showing how the steps of the sequence are captured. Only the exact steps done by this user when this particular event occurred are recorded.

The artifact model. People create artifacts to help them do their work. They may keep a list of contact numbers on a piece of paper by their computer; they may use a spreadsheet to calculate discounts and then circumvent the tool in the system that is supposed to calculate discounts for them. Each artifact offers insight into how the user approaches their work and what they need to support it. Each artifact offers details for how to get that support right: Exactly what information does the user keep for each contact? How are they calculating the discount, and how is that different from what the system would have done?

When artifacts are important to the users' work, it may be useful to capture them. Sometimes an artifact suggests that the system must do something and do it a certain way, such as calculate a discount a certain way. Other artifacts provide the exact details needed to implement a story, such

as the specific contact information needed by users. Even if the details are not written into the story—and since stories are high-level, they may not be—developers will come asking what to do in the implementation and the UX designer needs to be ready with an answer.

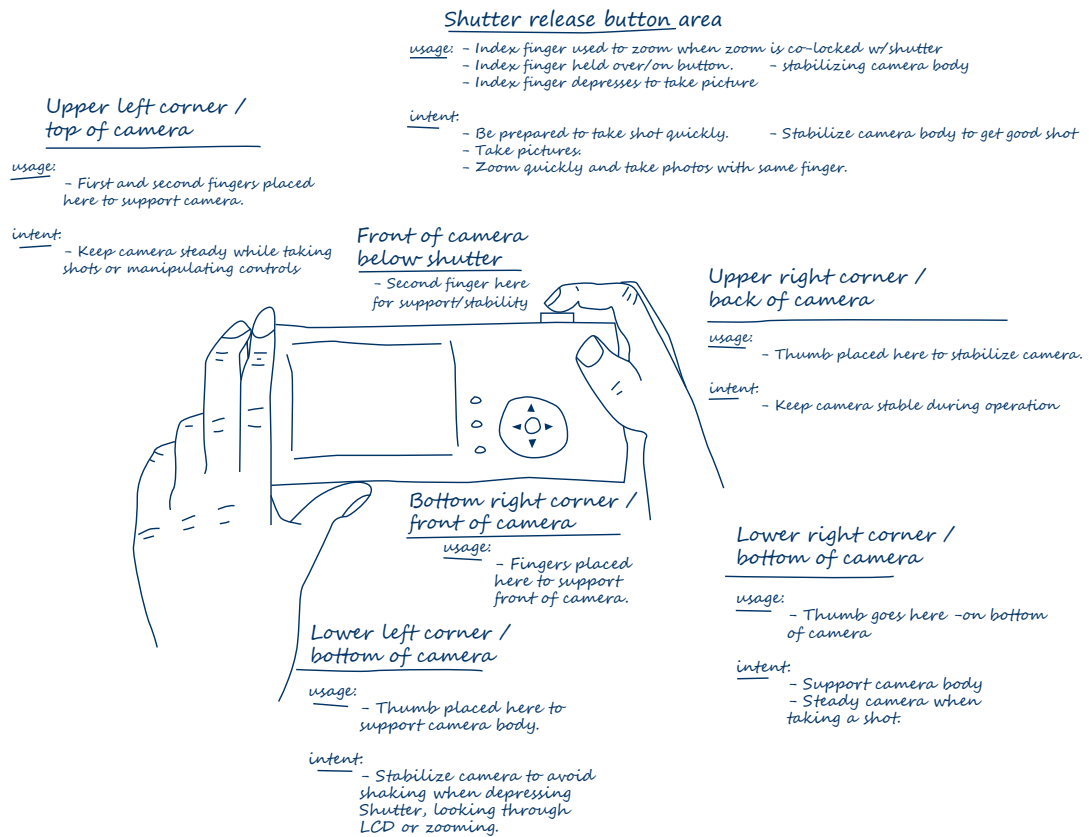


Figure 5.5: An artifact model showing the important aspects of the artifact (a digital camera) and information about how it is used.

5.1.4 PERSONAS

Personas are a popular way of representing a project’s user population. Unlike affinity and work models, which give the team a way to see the structure of their users’ work, personas are primarily a communication mechanism to give stakeholders insight into who the users are. As such, they break the agile value of “no documentation for the sake of documentation,” yet teams have found them so useful as a concrete touchstone for understanding their users that it is often worthwhile to create them.

Personas are characterizations of the different types of users as though they were real individuals. In projects operating without a good grounding in user research, personas are built by capturing stakeholders' impressions of who their customers are. But when a project has started with extensive user research, as outlined above, personas can and should be based on the actual users studied.

Since no actual individual is a perfect characterization of a type, information from actual users is combined to produce a single persona. Those aspects of the persona which are common and relevant are collected from different actual users to present a full picture of the persona. At the same time, details which are idiosyncratic or distracting are left out.

Personas can be used to ground communication throughout development. Initially, user stories can be written to support the specific personas: "As Anna, the lightweight project manager, I want to..." During iterations, referring to specific personas reminds the team of how roles, tasks, and attitudes combine in real people to ensure the design takes all a user's needs into account. Because personas are written as though the people are real, they are easy to deal with. They do not appear to be an abstraction.

AI Johnson, Reluctant HR Expert



"We think we are a modern employer"

AI Johnson is the finance director at a small manufacturing firm. He'd really prefer to spend 100% of his time overseeing finances and operations but he usually needs to devote almost half his time to HR issues. He's not trained for it and has no particular HR expertise, and the laws are so complex that though they try to treat their people well, AI constantly worries about whether his firm is at risk. He tries to keep on top of the latest updates in HR law and regulations, but there's so much information it's easy to miss things.

AI Johnson's time is stretched thin at his small manufacturing firm. Not only is he its full-time financial director, he's also the only one who can act as an HR expert in the firm. As a result, he is forced into the position of working on HR issues for which he is ill-trained.

Recently, for example, an employee got sick just before a planned vacation. Does his time out qualify as sick leave or vacation time? Another employee wanted to take vacation right before her maternity leave started, leaving a big hole in the staffing of this small firm. Should this request be honored?

And yet the firm wants to treat their employees well. For example, they have generous flex-time policies and participate in Britain's bike-to-work program. AI worked extra hard to make sure the pregnant employee, a recent Iraqi immigrant, understood the process and her rights.

Keeping on top of HR law and best practices is an ongoing headache for AI. He subscribes to an HR news services, but they send much too much information and very little of it is relevant to his specific situation. It is easy to miss important issues—it was his factory manager who noticed and told AI about the last increase in minimum pay.

AI does work with an external HR firm to augment his own knowledge. But he often doesn't think to bring them into a problem right at the beginning, which reduces their effectiveness.

Goals

- Focus on his "real job" of finance and operations, minimizing time spent on HR work
- Make sure his firm is not at risk because of how they treat employee issues
- Make a good employment environment for his people
- Stay on top of the "right way" to execute HR tasks

Tasks

- Oversee HR processes: Hiring, firing, discipline, and other employee issues
- Review proposed employee policies to ensure they meet HR regulatory requirements
- Work with external HR experts on handling specific issues
- Track employee absence, vacations, and sick leave

Roles

- Finance Operations Overseer
- Regulation Compliance Ensurer
- Disciplinary Process Manager
- Dismissal Process Manager
- HR Documents Maintainer

Figure 5.6: A persona. All the details in this persona are actual data from customer interviews.

5.1.5 VISIONING

Creating a system vision—a high-level concept of what the system is to do—is a necessary precursor to writing user stories. To see why, consider a typical story that might be written to support a network

monitor role: “As a network monitor, I want to see a warning when a communications line reaches 90% of capacity so I can prevent network outages or slowdowns.”

“No,” someone might say. “The system should rebalance network traffic on its own.”

“No,” another team member might reply. “I don’t think the system can do that. But we shouldn’t bother the network monitor with warnings they can’t do anything about. They’ll be overwhelmed.”

Who is right? The traditional answer from agile methods is: the agile customer or the product owner, and the team should defer to them. But that is not good enough for us. We need to say *how* those roles develop the understanding so that they can give an accurate, trustworthy answer. As we showed above, the people representing users to the team are rarely the actual end-user. Even if they were, end-users themselves might not know how to give a trustworthy answer. How should they know if they would be overwhelmed or not?

In the section on paper prototyping (Section 5.1.8 below), we will take up the question of how to decide what the right answer is. For now, notice that each speaker has a very different idea of what the system should do, how it should be structured, and how extensive the support it provides users should be. Should the system be automatic, doing much of the work for the operators, or not? Should we expect the users to be sitting in front of a screen all day looking for problems, or will they be distracted with other work?

These are the basic questions that define the system—its scope, its desired impact on the work practice, the basic function and structure to be provided, and the details of its behavior—and they need to be resolved before release planning. Often they are not, and the result is that the release planning meeting is not just about writing story cards. Participants have to design the whole system, in their heads, with no process support and no way to see the whole thing at once, and then write the results of their implicit design on story cards. This makes for a difficult meeting.

Instead, the BPUF (Big Picture Up Front) best practice suggests going into release planning after first sketching out an overall understanding of what the system is and what it is to do. The Contextual Design way to do this is with a *product vision* built together as a team, after collecting and analyzing user data with contextual inquiries, building an affinity, and creating work models.

A vision is built cooperatively by the whole team, including the product owner, UX designers, and developers. The only constraint is that everyone in the vision must also “walk the data:” review the data in detail and together. This is necessary to ensure that everyone’s ideas respond to the actual user data, rather than just repeating their own prejudices and misconceptions.

The vision, itself, is drawn on a flip chart, written in pen. The team describes how the user will do their work in the context of the new system, inventing features as they go to improve the work process. They naturally structure the system into coherent parts, each supporting a set of end-users (often referring to personas) and a set of work tasks. As in a brainstorm, the team does not evaluate while visioning; any problems are overcome through further visioning.

A team will always do several visions, exploring different approaches to the problem. Then the team can evaluate. They look at each vision in turn, decide what works and what does not about each

vision, and then consolidate the visions, eliminating those aspects that are infeasible or unnecessary, and bringing together the parts that work into a single, coherent view of the system.

This visioning process fits well into the agile approach and mindset. It is a quick process. Visioning can be done in one to three days, depending on the scope of the project and the number of visions. The documentation created is exactly the documentation needed by the team to do the work: the drawing of the vision on the flip chart. And letting the vision be hand-drawn ensures that it remains a sketch—a big picture—and keeps the team from over-focusing on the details.

5.1.6 STORYBOARDS

It is possible to write user stories directly from the vision. Each element of the vision can be captured in one or more user stories, which can be prioritized and used to drive agile development. But the vision itself has not yet been validated. Any stories based on the vision will need to be tested and iterated with users, usually needing several iterations to get it right. The lag between creating the vision and testing it, possibly many sprints later, will be long.

It is better to test and iterate the vision with users right away. This puts off the start of agile development but ensures that the initial stories reflect users' real needs. Less rework will be needed during sprints to respond to users' feedback.

In Contextual Design, the vision concept is tested by mocking it up in paper and iterating it with users. Users can respond effectively to a concrete user interface; it is much harder for them to respond to an abstract concept. A quick, high-level user interface design provides a concrete representation of the vision that acts as a communication device between designers and users. Detailed UI design can safely be put off until the development sprints.

However, jumping directly from the vision to UI design can leave holes. The task flow in the new system must be coherent and convenient for users. Unless task support is designed directly, some activities may not be supported or some sequences of steps may not flow well in the system. These problems will be discovered and fixed during prototyping iterations, but that is a slower and more cumbersome process than getting them right up front. Storyboards help the team discover and fix such problems sooner.

A storyboard lays out how the user accomplishes a specific task in the proposed system. Like laying out a storyboard for a movie, each storyboard cell shows how a step in the task is done. It shows the users involved, the UI screens they interact with, the data that appears on those screens, the actions they take, and any system behavior that is triggered by the user's actions or by other events. Each storyboard ensures that the task is coherent, that it can be performed smoothly and efficiently from start to finish, and that it takes into account any human processes, offline systems, and external systems the user may have to use along the way.

Each storyboard covers a single task and case. For example, a storyboard might show how the people monitoring a network are alerted to and respond to an emergency such as a power outage. Another storyboard might show how they would respond to a different kind of emergency, such as a hacker attack, which would exercise different elements of the system. Other storyboards would

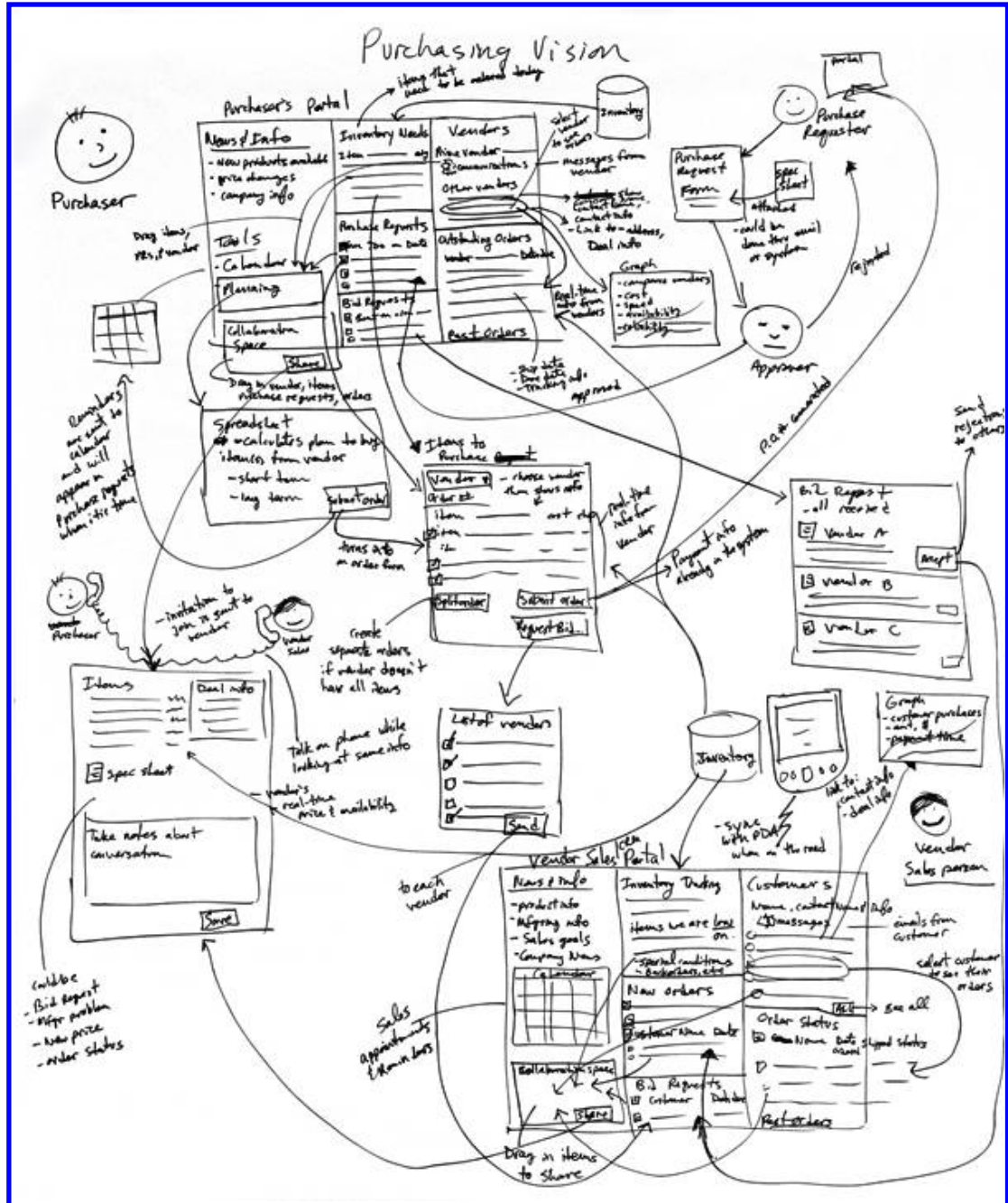


Figure 5.7: A vision: hand-drawn in a short interactive session with the team. It shows how users interact with the new system and do work outside it to accomplish their tasks.

cover other tasks. Usually 6-12 storyboards are enough to cover the main work practice of a typical system. A storyboard typically takes no more than a few hours to create.

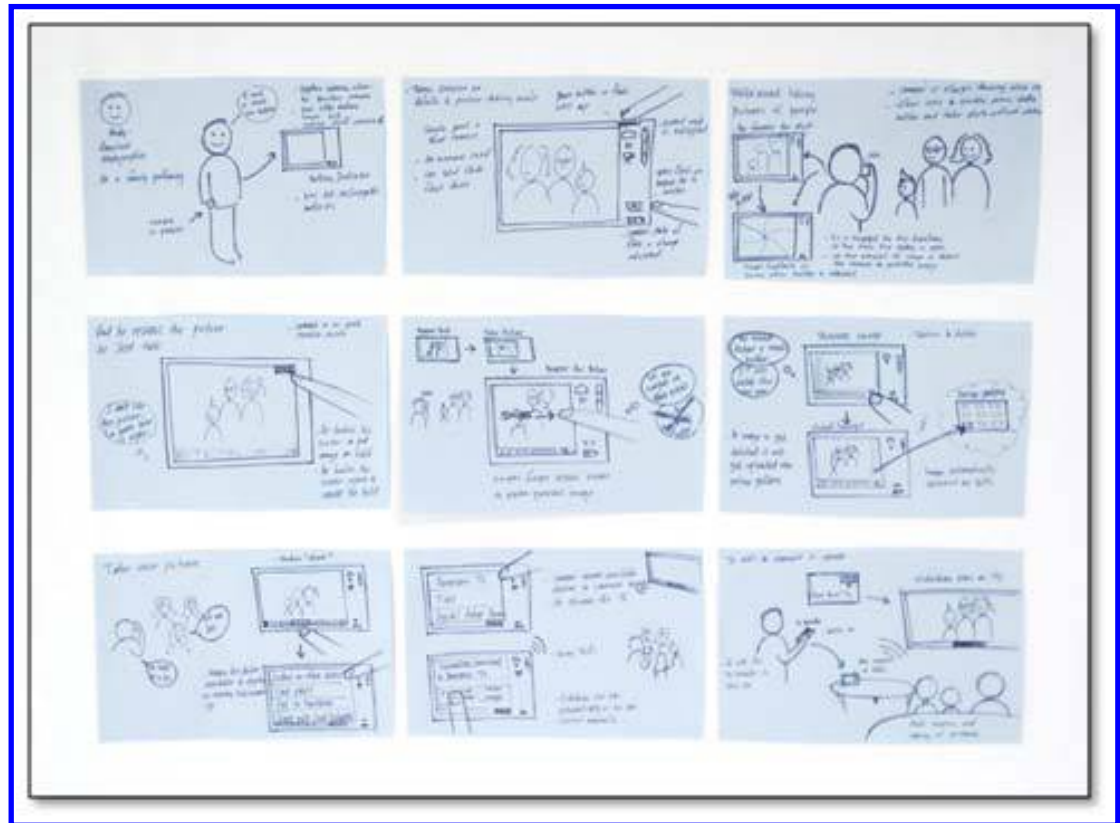


Figure 5.8: A storyboard, showing how the interaction between user and system is represented. (A camera is being designed in this storyboard.)

5.1.7 USER ENVIRONMENT DESIGN

Storyboards keep specific tasks coherent, ensuring that it is possible to go from step to step of the task in the system. But thinking only about individual tasks tends to create systems that are collections of wizards, each focused on supporting one task one way. Good systems are like houses: collections of rooms or spaces, each supporting a range of activities, each connected to other spaces in the house in ways that make sense for the work people do. Houses support many life tasks, many unforeseen by the architect; good systems do the same.

The User Environment Design (UED) gives the Customer Team a way to represent, at a high level, the structure of the system as experienced by the user. It does not show details of appearance

or implementation; instead, it just shows the places in the system and how they connect. It is like a floor plan for a software system or the site map for a web site.

Focus Areas are the places in a UED. Like the rooms of a house, they are experienced as a coherent place. They will be implemented as a coherent part of the UI such as a screen, page, or pane. Each Focus Area has a purpose, the work that it is intended to support. Each Focus Area provides function, describing what the system shows and does for the user in that place and what commands the place makes available to the user to invoke system behavior. In addition, each Focus Area provides access to other Focus Areas, allowing the user to move through the system.

Building a UED for a typical agile project takes 2-3 days.

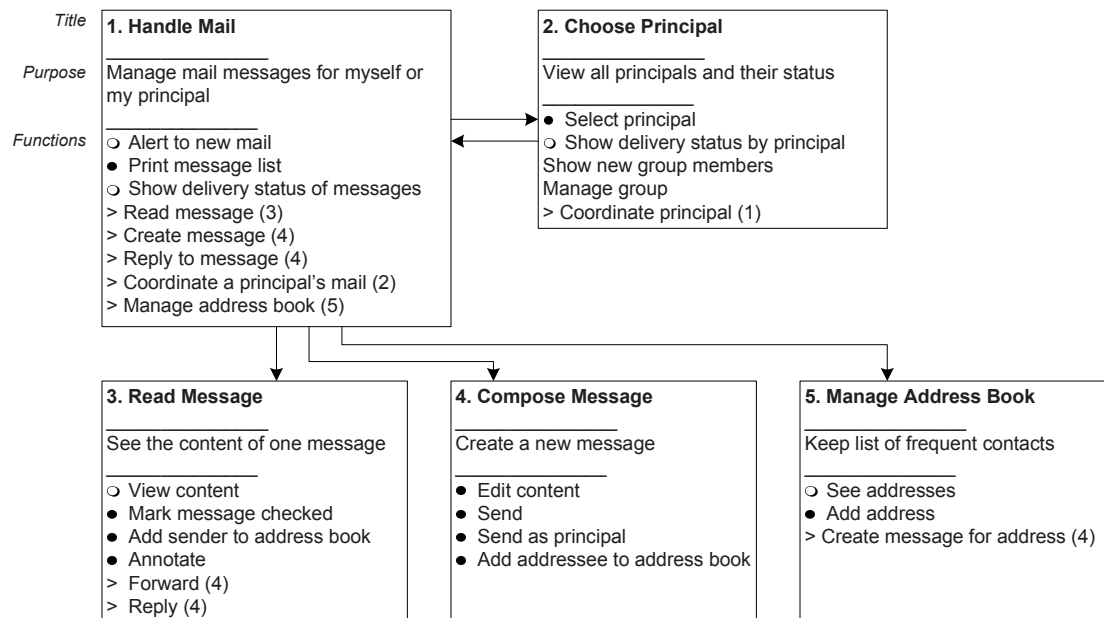


Figure 5.9: A partial UED showing the core of an email system.

5.1.8 PAPER PROTOTYPING

A key problem for any sort of user-centered design, including agile methods, is how to get accurate feedback from users. Users are not designers. They do not normally articulate their work practice and cannot envision how a proposed change would affect them on a day-to-day basis. How can they know how to advise the development team? An agile process allows the team to do the wrong thing and fix it more quickly—but we would like to do the right thing in the first place.

Paper prototyping provides an answer that has been widely and successfully used²⁴. Rather than describe a solution, show it. Rather than ask users how they like it, ask them to work with it. Rather than give them a task to do, ask them to do their own task in the prototyped system.

Prototyping can validate the system concept—that it is something people want, can refine the overall structure of the system, can ensure that the right functions are provided, and can refine the detailed behavior of individual functions. The prototype represents the system as a whole, so it helps to keep the design coherent across the system. User stories tend to break the system into independent bits, so this coherent view is especially important to maintain in an agile project.

Paper prototypes are quick to build. A designer can have a simple UI mocked up in half a day or so. Paper is easy to transport and, once at the user's workplace, easy to work with. Paper prototypes are also easy to modify. When user or designer suggests a change, they can mock up the change and put it into the prototype immediately. The user can experience it and decide whether they like it as well as they thought they would.

A paper prototyping session is run as a partnership between the designer and user. The first thing the designer does is find out about the details of the user's work and modify the prototype so that it represents the user's own data and situation. By using their own known information as clues for how the system is structured, the user can respond to it as though it were real. Therefore, they can pursue their own work tasks in the system, with the designer bringing out new system elements as needed. When there is a glitch, user and designer discuss the problem and decide together on a fix, which they implement (in paper) immediately.

Paper prototyping is done at two stages in an agile process: first, during phase 0, paper prototyping is used to validate the vision. Any vision represents a guess on the part of the team—a good guess, based on data, but a guess nonetheless. By mocking it up and testing it in paper, the team can reduce the lag between the time the design was created (the visioning session) and when it was validated by users. Two rounds of paper prototyping are enough to give the team the confidence that the stories they write are reasonable and valuable solutions to user problems.

However, paper prototypes and user stories alike are rough, high-level descriptions of solutions. They do not specify exactly how an interface will look or the detailed layout and interaction. Nor, by agile principles, should they. YAGNI (You Ain't Gonna Need It) advises that if you design too far ahead, you are likely to design features that will never be used. So instead, the detailed design is done immediately before implementation, using the best practice of designing one sprint ahead of implementation. That gives the UX designers enough time to work out and iterate the details with users before the developers come asking how a story is to be done. This UI testing is also done with paper prototypes, which are now more highly rendered, closer to the actual UI. If two rounds of prototyping were done during phase 0, a single round of prototyping is generally enough during sprints.

5.2 THE RELEASE PLANNING SESSION

Agile development starts with a release planning session. Together, but driven by the product owner or customer representative, the team writes user stories to represent the key elements of the release. UX designers and others who participated in user research during phase 0 with them act as the customer representative. They should act directly in that role on the team, and support the product owner with their knowledge of user work practice.

5.2.1 WRITING USER STORIES

Each user story captures an element or feature of the design. Stories are written on cards for the same reason that visions are written on flip charts: to keep the team from over-specifying and over-designing. All that will fit on a card is a simple, high-level description of the function or behavior.

The team writes stories by walking through their design (as represented by their vision, UED, or prototype), identifying each design element, and writing a story card to capture that element. A user story should provide value to the user; if several functions work together to provide desired user behavior, they may be grouped into a single user story.

User stories are written from the user's point of view to emphasize the value that the card will deliver. Take the example user story, "As a network monitor, I want to see a warning when a communications line reaches 90% of capacity, so I can prevent network outages or slowdowns." This specifies the feature, ties it to value, and specifies which users will be supported by the card. It does not specify every detail: how the warning is to be delivered, whether the warning will be delivered no matter what the user is currently doing, or how to show the warning so users have enough context to interpret it immediately.

The level of detail written in the card depends to some degree on the confidence of the UX designers. The number 90%, for example, indicates that the designers believe, based on their iterations with users, that 90% is the right threshold. If they were less confident or less specific in their knowledge they might leave the precise threshold open, to be determined later through additional customer work. Either way, inquiries and testing with customers may change the threshold once this card is selected for implementation.

Story cards are intended to deliver practical user value, and also to be entirely implemented within a single sprint. This means that complex functions will be split across several stories so that each is small enough to fit in a sprint. This is to be expected. It may well be that the part of the function specified on a single card may not be in itself enough to *satisfy* users, but it should be complete and implementable by itself.

Returning to our example, the initial story card might specify only that the warning must be given, no matter how. The implementation team would be expected to implement it in the simplest fashion possible, getting the underlying mechanism in place but with no effort expended in the user presentation. The team might implement a simple alert box.

Additional story cards would be written to complete the function. One might specify a visual icon to appear on the network map. Another might specify that hovering over that icon reveals

additional details. Another, that clicking on the icon would bring up a troubleshooting tool. Another, that when the network map is not visible a sliding pane will fade in and after a moment fade out in the lower-right corner of the screen. Each story is separately implementable and separately testable (though all depend on the initial story being implemented first).

Implementation teams new to agile development may find this approach awkward. It requires that they revisit the same function and same section of code several times over several sprints. This runs counter to traditional development practice. Traditionally, developers would want to implement all related functionality at once, create, get into, and get out of that module, and not have to revisit it later in development.

The upside of the agile approach to implementation is flexibility and predictability. Each story is small enough to be reliably estimated; if the estimation is off, since the story is small, it will be off by hours or a few days, not by weeks or months. And it allows the team to change course. For example, if it turns out that, once implemented, users like the network map so well that they leave it up all the time, the story about showing alerts in a separate pane might not need to be implemented at all. The time this story would have taken to implement can be allotted to other, more important stories.

5.2.2 ESTIMATING COST

As story cards are written, the team estimates the cost of implementing them. This may be done in “ideal programmer days,” the number of days a programmer would take if they had no interruptions and could concentrate only on this one task. This is a useful way for a new team to start with estimating stories. By referring back to real (if ideal) time, the team has a conceptual framework for assigning a number.

But experienced teams often find it more convenient to assign “points” to stories. A point has no unit, no real-world reference. But a story estimated at 4 points can be implemented in half the length of time required for a story of 8 points, and it will take twice as long as a 2-point story. An experienced team has estimated and implemented enough stories so that they can look at a new one and score it by comparing it with work they have done in the past.

Again, this seems odd from the perspective of traditional development. It works in agile teams because they measure progress in terms of *velocity*: the number of story points the team can implement in a single sprint. That number is tracked from sprint to sprint and the team assumes it will not change by much next time round. The team commits to implement specific stories in an iteration, and as long as the points for those stories add up to no more than their velocity, the team can be reasonably confident they can get all the stories done.

A brand new team, of course, has no measure of velocity to go on. For such a team, estimating by ideal programmer days for the first few iterations lets them use their gut feel to come up with a reasonable number. Velocity can be initially estimated by taking the total programmer-days available in an iteration and scaling back in the usual way to account for meetings and overhead. After a few iterations, the team’s measured velocity can take over.

UX designers should be ready to discuss the implementation cost of UI design. Remember that the cost includes all work not yet done to finish the story, including work to be done by the UX team. So a story card that refers to a complex part of the UI needing extensive UX work should have that work factored into its implementation cost.

5.2.3 PLANNING THE RELEASE

Once stories are written and estimated, they can be organized into sprints. Each sprint needs to make sense in several possibly conflicting ways. It should be implementable, which implies that if other stories implement necessary features or underlying modules, they should be scheduled first. Each sprint should support user testing, providing a coherent user interface. Risky stories depending on unknown or tricky technology should be implemented early so the risk can be eliminated or mitigated as soon as possible. High-priority stories, the ones most important to customer and stakeholders, should be implemented early.

In addition, of course, the total story points of each sprint must be less than the velocity. In fact, the team should allow a buffer of 20% or so in sprints following sprint 1, or else plan regular programmer's holidays. There will be bugs to fix; there will be rework to do in response to user feedback. If this time is not planned from the beginning, it will cause chaos later in the process. Either there will need to be extensive reprioritization at each sprint planning session to incorporate rework into the development stream, or the rework will be put off (increasing technical debt), or the team will have to overwork.

It is often useful for a sprint to have an organizing theme: a sprint might implement all monitoring with a bare-bones UI or might provide remote access throughout the system.

The first two sprints should be planned carefully, with the exact stories chosen. The following sprints can be organized more roughly. Careful planning of the entire release is generally counterproductive; too much might change before later sprints are started. Therefore, once users have experienced initial baselevels, the priority of following stories has a possibility to change: the business might change direction, rebalancing priorities or changing the delivery date; or the velocity of the team might change.

The release is a trade-off between function and date. Each sprint takes a fixed, known time; the velocity is known or estimated and determines the number of story points in each sprint. Therefore, the team can calculate how many sprints will be needed to deliver the critical function for the release. That sets the release date. UX designers should be prepared to argue for features to include in the release to support users' work practice. The business and marketing people may have requirements for what must be in the release for business reasons, and those are legitimate considerations. But it is the UX designers (and any other team members who gathered data with them) who understand the real impact of the proposed system on the end-users.

The release date can be adjusted by moving stories between releases; where a function has been split across stories, the stories that make the function more elaborate might be moved out of the release altogether to make room for more important stories. What must not happen is to jam

more stories into the sprints. This is a hard discipline for newly agile organizations. The traditional dynamic often has management pushing developers to “do more,” insisting on an unrealistic delivery date for business reasons. The agile tools help reveal that this is planning that creates chaos: it is planning for the team to execute at a speed the team knows and can demonstrate it cannot achieve.

This release plan should not be taken as cast in stone. It shows how the team can deliver certain functionality by a certain date, but it is expected that as the team learns more about its users and as users learn more about how the new system will affect their work, the plan will change. Each sprint planning session may change story priorities, rewrite stories, and introduce new stories. This is expected, and one of the reasons agile teams deliver useful results.

If any stories are critical to the business for external reasons, it is up to the team to know and represent those constraints. The team is in close contact with its users via continual testing and feedback. Any adjustments the team makes along the way should meet user needs more precisely. On the other hand, management may have made commitments to important customers. Product marketing may feel that the product must have a certain feature to compete in the marketplace, whether or not that feature is actually useful. (This happens more often than you might think.) Or other parts of the organization—manufacturing, or training—might be gearing up in parallel, expecting and depending on certain features to be present. Such commitments to stakeholders other than the end-user must be understood by the team, so they can be honored.

5.3 RUNNING SPRINTS

With the release plan in place, agile development proceeds in a series of sprints. Each sprint implements a set of user stories. Where these stories define user interaction, the visual design has not yet been done—rough wireframes and paper prototypes are enough to test the concept. Any further design is put off until needed.

To give the UX designers time to develop final UIs and test them with users, the best practice is to work one sprint ahead of developers and test one sprint behind. So in sprint 1, the UI people develop and test the final screen appearance for a user story; in sprint 2, they work with development to implement it; and in sprint 3, they test the implementation to ensure nothing was lost in the translation.

The first sprint may be treated specially. Often, there is a fair amount of work necessary to get the team up to speed. On the implementation side, such activities include defining the development practices, putting together an automated build and test mechanism, and deciding on and implementing a source code control system. On the UX side, there are detailed interaction designs and final visual designs to work out and test for the initial user stories. So, the first sprint may not deliver actual user value, but it may put all the building blocks in place for the project. This is especially important for a new team that has to set up its tools and procedures for the first time. It is also important for the UX team, who need to provide detailed visual designs to developers before the developers start coding.

Table 5.1: The UX work on a story interleaves with development of the story, with design done one sprint ahead of implementation and user testing one sprint behind.

	UX Team	Development Team
Sprint 1	Design UI for story 1; prototype and iterate with users	Put development system in place; implement low-UI stories
Sprint 2	Design & iterate UI for story 2 Consult on implementation of story 1	Implement story 1
Sprint 3	Test implementation of story 1 with end-users Consult on implementation of story 2 Design & iterate UI for story 3	Implement story 2

Once the team is in the swing of iterative development, they repeat the same pattern over and over again, as follows.

5.3.1 SPRINT PLANNING

The team meets to plan the work for the sprint. This starts with the set of stories identified during release planning, but this is the time to consider whether that plan still makes sense. Do these stories still reflect the next most important, useful, and risky set of stories remaining to do? Is there other work, unforeseen when the project started, that should be planned? In that case, stories need to be written, estimated, and prioritized into the schedule. Is there rework to do, either bugs to fix or redesigns to implement, based on user feedback? Then stories need to be written and prioritized into the schedule. The specific stories for the current and the next sprint need to be determined.

Then most teams will write *task cards* for each of the stories in the current implementation. It is a useful discipline to mandate that no one does work unless it is to implement a task card. (And task cards are only created to implement user stories, so all work provides customer value.) Task cards can represent implementation tasks, such as, writing code and designing databases. They can represent UX tasks: user tests, paper prototypes, and UI design. They can represent tasks to be performed by other members of the team, such as documentation and QA.

The UX team needs task cards for the stories in the last, the current, and the next sprints. Their task cards for stories in the current sprint specify UI design tasks: doing low-level layout and visual design to support developers before they start coding the story.

But the UX team also needs to look ahead to the next sprint. They write task cards to start UI design on those stories: taking the high-level design from vision, storyboards, and UED, and designing the final layout and look for this UI. If several paper prototype rounds were done in phase 0, they may design and test the finished UI directly; otherwise, they may need one or two

rounds of paper prototypes first. These designs are tested with users during the current sprint for implementation in the next.

And the UX team also writes tasks for the work completed in the last sprint, to take working code and test it with users to ensure the final implementation actually works for people.

5.3.2 WORKING WITH DEVELOPMENT

During the sprint, developers implement stories. The UX team has three jobs running in parallel reflected in the three types of task cards. First, they support the developers. They provide detailed UI designs and consult with developers on detailed behavior and look. Remember, in an agile team, these consultations are where the real decisions are made and the detailed behavior is communicated. There is no functional specification; UX designers work with the developer during the sprint to say exactly what they want. It is up to the UX people to be tightly tied into the development team, providing guidance and answering questions as they come up. Daily discussions between developers and UX designers are usual.

The UX team also runs customer visits throughout the iteration to bring real user feedback into the development process. These visits accomplish the dual purpose of testing work done in the previous iteration and doing low-level design for work to be done in the next iteration. The team does this by running a customer interview that combines contextual inquiry and paper prototyping.

5.3.3 SPRINT INTERVIEW PREPARATION

Before the interview, UX team members identify which elements of the work done in the prior sprint need user feedback. Ideally, all UIs would be tested with users; in practice, teams are often making trade-offs in the use of limited resources. If a UI is straightforward or similar to what users have now, the team might decide that it is not sufficiently important to spend time on. But parts of the UI which are new, tricky, introduce new interaction paradigms, or depend on the exact behavior or look of the UI elements should be tested in the running code. For these parts of the UI, this is where the rubber hits the road: does the implementation conform well enough to the design that it actually works as expected? Or did the translation to running code introduce enough glitches or changes that usability of the system is affected?

The UX team also looks at the stories to be implemented in the next sprint and evaluates where the trouble spots are. Which stories implement UIs that were not thoroughly tested, or that could not be well prototyped in paper because they are too interactive, or that depend strongly on the detailed visual design for success or failure? Or are they new stories for user tasks which were not covered by the initial research at all? The UI design for these stories will need to be started and tested with users before the next sprint starts.

To test the UI for these stories, the UX team develops the final, fully-rendered visual design and then chooses how to prototype it with users. If interactivity is not a problem, they may print the rendered UIs and show them to users on paper. Otherwise, they build a more or less complex online prototype ranging from a simple flat image to a fully interactive prototype. Remember, this

is a team responsibility. It is reasonable to collaborate with a developer to build a more interactive prototype than the UX team member could build on their own. That is just part of the cost of the story.

5.3.4 THE SPRINT INTERVIEW

The interview is run as follows: The interviewer introduces themselves and their focus to the user. They tell the user which elements of the system they want to test and what tasks the prototype supports. Then the interviewer gets an overview of the user's work, enough to get oriented and understand how to introduce the prototype. During this initial introduction, the interviewer plans how they will run the interview and what they will cover.

If the team needs to test a new design, the interviewer goes into paper prototyping mode. Sometimes the prototype really will be in paper, though more detailed and exact than the prototypes developed during phase 0. If so, the interviewing process is similar: walk through the user's own real-world examples, discuss how the prototype supports the user in doing the work, revise and modify it on the fly as problems are discovered. An online prototype, of course, is less flexible. But the interview still follows specific past events, replaying them using the prototype system. Interviewer and user discuss any issues and how they might be fixed. The interviewer can sketch solutions if significant changes are being made.

If the team needs to test parts of the previous iteration, they make the running code available, either on a laptop or through a web interface. The interviewer brings up the code for the user and then proceeds as for an online prototype except that now the interaction is real. The user interacts directly with the system, getting a feel for such issues as lag time, ability to recognize UI elements, interactivity, and so forth.

It is possible that the team needs to understand a work task, either because it is new to the team or because they need additional detail. If the user does the task, the interviewer moves into CI mode. Using observation of ongoing work and inquiry to build retrospective accounts, the interviewer discovers how the task has been done by this user. This data is captured in the interviewer's notes for interpretation and analysis by the team in a later interpretation session.

A typical session may incorporate all three styles of interview. The interview should be planned for two hours, which is enough time to investigate a range of issues but not so much that it becomes hard to get users to commit the time.

5.3.5 THE INTERPRETATION SESSION

As with CIs and paper prototype interviews, the data from the interview is analyzed in an interpretation session. The UX team and any part of the full project team that is interested goes through the events of the interview, in order. The team writes notes to capture issues and also validations.

Afterwards, the team evaluates the results of the session. Any notes about a proposed design—a design for a story that has not yet been implemented—are used to change the design and, eventually, produce a revised prototype. Notes identifying issues in the last sprint are more complicated. The

team redesigns the interface to solve the problem and writes a story card to represent that fix. The story card will be prioritized in the next sprint planning session.

This is the preferred method of handling rework. Some teams prefer to do rework through the bug fix process, treating UI problems as bugs to fix. This can work, but tends to bury UI problems with the rest of the bugs, and it breaks the rule of doing no work without a user story to justify it. In truth, large bug lists are a danger sign on an agile project. They represent technical debt; worse, any priority 1 bugs on the list mean that the system is not, in fact, ready to ship at the end of the sprint. It is better to prioritize fixing them into the next sprint, even if other user stories have to be put off. And it is better not to add to them with UI changes that could be handled through the user story process.

Note that this way of working requires continual customer visits throughout the sprints. This is inevitable if user feedback is to drive iterative change. An effective customer team will have someone charged with recruiting users and planning visits. Many teams find it simplest to plan these visits ahead of time, on one or two fixed days of the week. Then, rather than scrambling to set up a user visit once they have something to test, the team scrambles to finish their prototypes in time for the next visit. The first approach creates delay waiting for users to respond and arrangements to be finalized. The second encourages progress, as designers work to get enough in place to make the visit productive.