Hitting the Target: Adding Interaction Design to Agile Software Development

Jeff Patton
Development Team Leader
Tomax Technologies
224 South 200 West
Salt Lake City, UT 84101
USA
1.801.924.6924
jpatton@tomax.com

ABSTRACT

Extreme Programming appears to be a solution for discovering and meeting requirements faster (through close customer collaboration) as well as creating quality software. In practice we found XP did deliver high quality software quickly, but the resulting product still failed to delight the customer. Although the finished product should have been an exact fit, the actual end-user still ended up slogging through the system to accomplish necessary day-to-day work. This paper describes using interaction design in an agile development process to resolve this issue. Using interaction design as a day-to-day practice throughout an iterative development process helps our team at Tomax Technologies deliver high quality software, while feeling confident the resulting software will more likely meet end-user expectations. The method of Interaction Design followed here is based on Constantine and Lockwood's Usage-Centered Design. Recommendations are provided on how to practice an agile form of U-CD and how to incorporate bits of Interaction Design thinking into every day development and product planning decisions.

Keywords

Agile Methodologies, Interaction Design, Usage-Centered Design, Extreme Programming, Requirements Gathering.

1. INTRODUCTION

This experience report discusses my discovery and incorporation of Constantine & Lockwood's Usage-Centered Design [6] into the day to day work my team does to deliver high quality software.

Summarizing observations of several projects in Reflective Systems Development [13], Mathaissen observed "Systems development methods were seldom, or only partially, followed by experienced practitioners." We were no exception. Our current form of U-CD uses new skills taught by Larry Constantine & Lucy Lockwood. In addition we've made several modifications to the process to accommodate time limitations, information limitations, and an iterative development environment. The result is a situation specific "agile" form of U-CD that fits tightly into our team's local methodology.

When incorporated into an agile development process, the interaction design concepts in Usage-Centered Design pack a powerful 1-2 punch: Agile development methods allowed us to deliver high quality software sooner, and interaction design concepts lent us the degree of end-user empathy we were missing to help increase confidence that we hit our target of end-user satisfaction. All this results in our team being more successful today than 2 years earlier.

2. IDENTIFYING THE PROBLEM

2.1 There Has to Be Better Way.

I've spent years developing software "traditionally." Basically, this consisted of a blend of Waterfall Methodology and complete chaos. I saw intelligent folks work very hard to identify requirements, create a thorough definition of scope and functional design, approve that, and then finally build it. More often than not the resulting software would miss its target. It was often late. There were problems with quality – the software released with bugs. But even with quality issues resolved, the resulting software was hard to understand and cumbersome to use. It didn't seem to be appropriate for the actual work the end-users were trying to accomplish. Requirements were often missed in the design phase, resulting in features necessary to automate the business process being left out of the software. Features originally thought important during the design phase were often discovered to be unnecessary and went un-used.

Watching this cycle over and over left coworkers and customers paralyzed with fear. We analyzed and designed longer with the hope we'd get it right the next time. Customers reviewed designs longer or delayed reviewing them out of fear they'd miss something and be blamed for inevitable omissions in the delivered product. We started developing late. We finished even later.

There had to be a better way.

3. FINDING THE SOLUTION

3.1 Enter Extreme Programming

Extreme Programming [3] surfaced as an alternative to this madness along with other new ways of developing software – now branded as Agile [1]. Surely close customer collaboration and iterative development would correct customer satisfaction issues. Surely test-driven programming [16], pair programming, and aggressive refactoring [11] would improve software quality.

I stumbled into and spent a valuable year with Evant Solutions, a company committed to XP principles. We built high quality

software at an aggressive rate. Deliveries were on time and with the expected scope usually intact. However, I still found the company missing targets. The resulting product seemed to have features the actual end user didn't need or care about while lacking features the end user did need. Much time seemed to be spent on features that would go un-used by actual end users. These same actual end users had to devise lengthy procedures to force their actual business processes to work with the software that was built. Shouldn't close customer collaboration have mitigated this issue?

Ideally an XP customer is an expert end-user employed by the company actually purchasing the software. In Evant's case they were indeed expert users at one time, but now as product managers working for Evant, they had the responsibility to deliver commercially viable software to be competitive with other products in the same marketplace. They had to balance the needs of users we currently had with users we hoped to acquire. This was a daunting responsibility requiring tough trade-offs.

At delivery time, it wasn't always clear where those tradeoffs had been made. Product managers seemed to be surprised that actual end-user needs had not been met. We moved into a reactive mode delivering software that we hoped would address end-user needs, then waiting for the inevitable requests to make changes. An element of empathy with the actual end-users seemed to be missing. We were guessing what they needed. Was this just an unavoidable challenge of software development?

3.2 Beauty Is Only Skin Deep.

At Evant we'd always worked hard to make our software look good and be easy to use. Our UI specialist did a fabulous job with screen design and the product was consistent and easy to understand. However we still had the issue with actual business processes being hard to accomplish in the software and important parts of business processes being left out completely. Sure it looked good, but apparently there was more to hitting the target than looks.

3.3 We're All Crazy.

Years ago I'd read Alan Cooper's About Face [9]. It contained lots of good information on what not to do when designing the software's user interface. But our software seemed to have a good user interface – at least it didn't break any of the major rules.

During the spring of 2001 I was able to hear Cooper speak in Berkeley. His focus was less on bad screen designs and more on software missing its target as a result of not understanding its user. He pointed out the necessity of a *persona*. A *persona* was a walking, talking, fictitious user with well-developed fictitious needs and concerns. Reading Cooper's The Inmates are Running the Asylum [10], I found that as a technologist, I'd likely never be able to identify with my user. I, and the folks I worked with were the inmates and it would take serious effort to think like the *persona* we could create. Knowing you have a problem is half the battle. I proceeded under the assumption I could find the steps to recovery.

In an attempt to move forward, I latched onto what looked like a valuable starting point. Alan Cooper used the term "interaction design" to describe the missing role in software development, and further said "Almost all interaction design refers to the selection of behavior, function and information and their presentation to users." [10] Looking back at challenges I'd experienced before it seemed that we did a poor job of selecting the appropriate behavior and functions to implement in the software.

4. FINDING THE SOLUTION, AGAIN

4.1 It's not Chet's Fault.

While lurking in the ExtremeProgramming discussion group [18], I read Ron Jeffries recommendation of Constantine & Lockwood's Software For Use [6] as a possible source for good information on user interface design. Although Extreme Programming Installed [12] may encourage blaming Chet – I'll assign Mr. Jeffries the responsibility for starting me down this path.

Like Cooper's concerns, Constantine and Lockwood's justifications for effective user interface design and usability were preaching to the choir. But something different in this book was an actual documented method for arriving at a usable piece of software including choosing appropriate behavior and functions for the people identified as users of the software. However, the process described looked complicated, time consuming, and not easily adapted to an agile development approach. What's more it needed to happen up front. Adding a time consuming process to the front end of software development sounded too much like the bad experience I had been running from.

4.2 Ah-Ha!

During the summer of 2001, I had the opportunity to learn Usage-Centered Design from Larry & Lucy directly. The book is thick – and I'd wondered how we were going to compress this process into a weeklong class. As exercises between lectures we discussed a business problem, brainstormed ideas onto 3 x 5 cards and saw models emerge almost magically by shuffling cards around the table. We did this collaboratively in a group with lots of discussion. We learned an effective way to move from these arranged models of cards on the table to wireframe user interface. We learned how to validate – or test – our user interface using the information we'd put together still on those 3 x 5 cards.

The business problem we took on to solve in exercises seemed daunting at first. But surprisingly, in a short amount of time we arrived at an effective design. And what's more, the whole process was understandable and fun. If this was Usage-Centered Design in practice, I could easily see it used as a collaborative approach to generating story cards for use in Extreme Programming development. This would surely result in us delivering software that was high quality and effective at meeting the real business needs of the user. Simultaneously I stumbled onto an assertion on page 122 of Cockburn's Agile Software Development [5] that when we look at the scope of concern for Usage-Centered Design and XP that the two sets of practices could indeed inhabit the same project.

If past problems sprung from sometimes selecting and building the wrong behavior, possibly using U-CD as a method for interaction design would result in selecting correct behavior more often. With XP we could now accurately plan, develop, and release a set of features. We could hit the target of on-time delivery and high quality. With more method behind choosing

the features to implement, we now hopefully had a target that more likely included end-user satisfaction.

5. DEFINING THE SOLUTION

5.1 Agile Usage Centered Design

Although Usage-Centered Design is thoroughly explained in Software for Use [6], an Agile approach is first documented in Larry Constantine's paper: "Process Agility and Software Usability: Toward Lightweight Usage-Centered Design" [8]. The steps given here are an abbreviated overview of the process. This is Constantine and Lockwood's process with a few minor variations to match the way my team and I practice it today.

1. Identify participants.

    Sequester a diverse mix of people in a room to collaborate on this design. Include domain experts, business people, programmers and test/QA staff. Include a facilitator that knows this process.

Figure 1. Collaborative design sessions include a diverse mix of people.

2. Preconception purge.

    Let loose. Everyone brain-dump about the software we need to write. Complain about the product you're replacing. Explain the cool features you expect the new product to have. Get everyone's concerns out into the open. Write these concerns down in plain sight on whiteboards or poster sized paper hung on the wall.

3. Review the domain.

    Domain experts and users in the room explain the business process, as it exists today. Who is involved in the process? What combination of manual processes and computer based tools do current participants engage in to meet their goals?

4. Define user roles and role model.

    Brainstorm user roles onto 3 x 5 cards. Who will be using this software? What are their goals? Prioritize the roles by shuffling the stack of cards. Note the most important roles. Label those roles *focal* roles. Place them in an arrangement on the table that makes sense with similar roles closer to each other. Discuss the relationships these roles have with each other. This is a role model.

Figure 2. Fixing role cards to poster paper and annotating relationships allows the role model to be posted for everyday reference.

5. Define tasks and task model.

    Now that we know who will use our software, brainstorm tasks these roles will be doing to accomplish their goals onto 3 x 5 cards. Shuffle the cards to prioritize them based on importance, then on frequency. Note the most important and most frequent. Label those tasks *focal* tasks. Arrange the cards on the table. Place tasks similar to each other, or dependent on each other, together. Place tasks that have nothing to do with each other further apart. Discuss the relationships these tasks have with each other. This is a task model.

6. Define interaction contexts.

    You'll find tasks in the arrangement on the table clump up. Grab a clump. This is an interaction context. Give the interaction context an appropriate name.

7. Detail user tasks.

    For each task in your interaction context, write a Task Case directly on the card. The Task Case takes the form of a conversational Use Case similar to that described by Rebecca Wirfs-Brock in [17]. Alistair Cockburn in Writing Effective Use Cases [4] might classify them as "system scope, sea-level goal, intention-based, single scenario, Wirfs-Brock use case conversation style." U-CD would encourage you to simplify and generalize these Task Cases. Using a conversational form makes them easy to read. Limiting the scope and goal keeps them from being too broad or too detailed. Generalizing them keeps them short and allows deferring user interface details for implementation time.

8. Create an Abstract Prototype.

    For each interaction context, using the task cases you've detailed create an abstract user interface prototype. This process is best described in [7]. At the end of this process

you'll know what components will be on the interaction context.

9. Create wireframe user interface.

   Using pencil and paper create a wireframe drawing of the interaction context. Show basic size and placement of screen components.

10. Test the interaction contexts.

    Use role-playing to step through each task case used in the interaction context. One participant pretends to be the role that would perform the task, another plays the role of the user-interface. Validate that you can easily and effectively reach your goal using this interaction context.

6. PUTTING IT INTO PRACTICE

6.1 Starting In the Middle.

Armed with a year's worth of Extreme Programming development experience, U-CD training, and lots of other bits of useful information from books, papers and colleagues, I set out at Tomax, my current employer, to prove that U-CD + XP was indeed a potent combination that would lead to on time delivery, high quality and ultimately satisfied users. The rest of this paper describes how close we came and how much remains to be discovered.

While it's exciting to think we could put into place a set of new practices, we never quite have a clean slate. In my situation we had legacy practices to deal with. When it came time to apply Usage-Centered Design it was often a bit too late. There was no shortage of new software to write, but before our company had agreed to write the software, documents had generally been written up and agreed-to describing scope, features, and functionality. In many cases if we were to attempt to practice U-CD our company would have been accused of re-trenching the same material already discussed by marketing and/or project management. Looking at the use of the software often meant asking users to repeat conversations they'd already had when drawing up the agreement. In addition the results of such a conversation may yield changes in scope. This notion was at best unpopular.

6.2 Some Opportunities and Some Success

There were, however, some greenfield opportunities. These were projects where requirement were not yet agreed to and where the customers and management were willing to approach things in a slightly different way. In those situations we practiced Agile U-CD as described above with some success.

6.3 What Worked:

The preconception purge before the process seemed to be the chance to vent everyone was looking for. Giving the group permission to have an unorganized conversation where anything could be said brought to light many concerns and fears we'd have not gotten to any other way. This free form conversation supplied everyone involved with an immense amount of useful background. We left ideas captured during this process on poster paper taped to the wall. At the end of this process we were able to double back and make sure we'd dealt with the concerns, or found they weren't really concerns any more.

Working with 3 x 5 cards struck some participants as very low tech, but the results were very effective. The discussion took the same form as a CRC card session [2] might take. But, instead of classes, responsibilities, and collaborations, we talked about user roles and tasks. We saw lots of card waving and passing cards back and forth. People immediately understood what was important by looking at the position of the card on the table. People immediately knew what ideas were related by their position in relation to each other. An arrangement of cards on the table could communicate far more, faster than any paper document or diagram could. We found that taping card arrangements to poster paper, then marking up the taped arrangements resulted in a very valuable model.

Figure 3. Participants quickly learn to work with 3 x 5 cards.

Mapping Task Cases to Abstract Prototypes was a very simple and effective way to push through from knowing what we needed to do to how it might look on the screen. The Abstract Prototype consisted of post-it notes, signifying abstract components, stuck to poster paper. We could easily rearrange them and push through this paper-prototyping phase to a simple wireframe user interface.

6.4 What Was Bumpy:

Folks had problems with User Roles. In U-CD a role isn't a job title – but more accurately a high level goal. For example: Clerk is a job title. CustomerSalesTransactionHandler is a role. The distinction becomes important when someone looks at a list of roles later and is unable to determine what each does. Or when looking at a task case like ReturnMerchandise and ask who does it? In this case if you're using job titles, the Clerk, Assistant Manager and Manager may all have responsibility to perform that task – but, we'd have to know the business rules to be sure. However, we can reasonably assume a CustomerSalesTransactionHandler might have that responsibility. Choosing expressive role names is valuable – but is a hard idea to grab onto for domain experts. In practice I found it easier to let folks use roles like "clerk" initially. During discussion of the role and the goals the role had, we could easily

convert the job title to one or more role names that captured the users' goals.

Attention spans weren't long enough. By the time you reach the tail end of the process when it will really bear fruit, people are exhausted and unable to effectively do a good job building the UI. Reconvening the next day left us with a fair amount of ground to cover again to get everyone back on the same page. The process takes a while and for those who don't do it often, it's time consuming and tiring. Folks were accustomed to one person going off to a cubical to write functional specifications and not this long collaborative process. As anyone who practices pair programming can tell you, constant collaboration can be exhausting. We found it most effective to split the process at the point we'd identified interaction contexts. We could then continue the process at a later time using a smaller more focused group of people – those that were ultimately responsible for delivering the system.

The resulting artifacts look funny. In this organization functional design previously took the form of a list of "shalls" – the software "shall do this" sort of statements along with assumptions, a very literal screen design, and sometimes a narrative on how it would be used. Roles and a role model weren't immediately understandable. Task Cases seemed too general – too abstract for some folks. Wireframe UI drawings weren't quite literal enough. These issues impacted acceptance of the functional design. On occasions that we needed to produce functional design, it seemed to work best to document user roles, the names and goals of each user task, and cleaned up versions of wireframe user-interface drawings. These things dropped into a document seemed to look enough like requirements for folks to "sign-off" on the effort.

7. REFLECTING ON WHAT WE'D DISCOVERED

7.1 Were We Gaining Anything?

It sure felt that way. Although close collaboration within a large group was tiring, when we finished the amount of tacit knowledge in the group was irreplaceable. Everyone within the team understood who the users were and what their goals were. Those in the team who hadn't been present for the U-CD sessions quickly assimilated the vocabulary of those who did. Artifacts, such as role and task models, created during the session were posted in the development area to "radiate" [5] information throughout the implementation of the software.

Our priorities became clear. We need only find the focal, or most important user roles and their focal task cases to find the best starting point for development. If we became bogged down implementing functionality for roles that weren't focal, we could justify choosing a simpler, less elegant, but cheaper and faster approach.

Was this better than a long, functional design written by one expert? It's not easy to say that the results were definitely better, but it is easy to say that team members' understanding and ownership of the software was higher than before. By arriving at this functional design together, all knew how to accomplish this process and we'd eliminated what was before a single point of failure. This seemed like a definite improvement.

7.2 Test-Driven Design For User Interactions

Throughout the development process, whenever anyone on the team was unclear on the direction we were going with the software, we'd pick up the original task-cases and attempt to execute them on the software. They became our working acceptance tests.

Knowing user roles helped answer other questions – like what the ability level of the user was and what that user's goal was. For example, often in a business process the goal of the user doing the process is much different than a manager who needs to have visibility of what was done. They need to see different information at different times. Using user roles, circumstances like this became clearer.

Finally, when formal acceptance and QA had to occur, task cases could be "fleshed out" to contain specific references to the actual implemented user interface along with literal test data. Roles would serve as a collection point for acceptance tests. We'd focus on validating the software a role at a time essentially wearing the hat of the user role and performing the work they'd need to perform with the software.

Our confidence in the finished software was higher. The feeling seemed analogous to the feeling you get developing source code using automated unit testing and test-driven development. It's not really provable that code developed this way is better than other ways, but after doing it I find my confidence in the code is higher. I also find I'm unwilling to work any other way as that seems risky or foolish. As with test-driven development, there was no knowing if our finished results were indeed better than we could have come up without U-CD, but confidence was higher. Proceeding on a project without knowing what user roles existed for the product and what tasks they needed to perform now feels as risky as writing code without unit tests.

8. WHAT SHOULD I DO TOMORROW?

8.1 Interaction Design Incorporated Into Day-to-Day Processes of a Mostly Agile Company.

At Tomax Technologies, certain agile processes have taken off and work well. Scrum-style [15] daily stand-up meetings are commonplace. Cockburn's Information Radiators abound [5]. Teams develop iteratively, many of them using schedules generated by an XP style planning game. Some teams religiously use unit-testing, pairing, and refactoring. Other teams are still a bit suspicious of all these new-fangled ideas. Although we have product managers, they don't have the time to ride shotgun on a project the way an XP customer should. They rely on the team to make the detailed decisions about the implementation of features in the product. Acceptance testing is up to the team and performed by test/QA staff assigned to the team. Development methodology is a decision made more at the team level than the corporate level. In this sort of environment, how do we incorporate some interaction design into things we do every day?

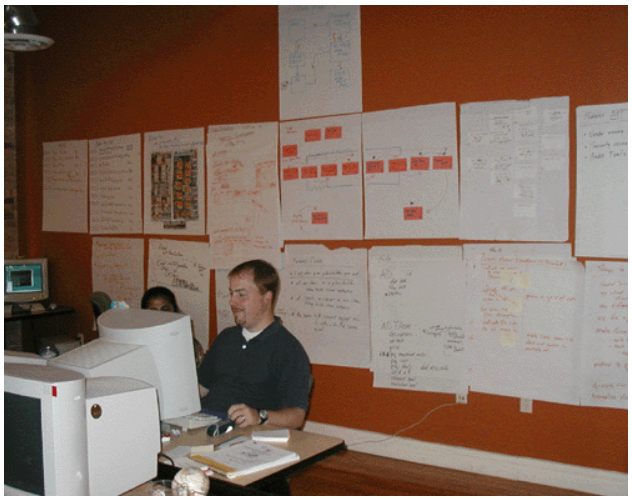

Figure 4. Our development environment at Tomax is wallpapered with role models, task models, task cases, and wireframe UI drawings alongside XP-style iteration schedules.

The following is a short list of Interaction-Design-centric guidelines our team tries to observe:

1. We always ask "who?"

   While we're looking at a piece of development we make an effort to understand who will be using it. What is the user role involved? If we don't know, we back up and do a quick user-role brainstorming session. Arrange a few 3 x 5 cards on a table to understand the role model, and then continue on. When we understand who will be using the application, we make better decisions about what they should see and how sophisticated the interactions can be.

2. We validate user interactions with a task-case.

   To make sure our user interface is usable, we write a simple task-case giving us the step-by-step intention driven process a particular user role might follow to complete the task. Does the current design of the application do this efficiently? This may be analogous to a manually executed XP acceptance test.

3. We strive to understand focal user roles and focal task-cases.

   Make sure everyone in the project understands who it is most important to satisfy and what specific activities need to run smoothest. Focus on those. Spend extra effort to make them right. Allow the less important roles and task-cases to slide. They need to be functional - but fluid and pretty may be a little less important. Time is most wisely spent elsewhere.

4. We look for features that don't serve any role or facilitate any task.

   There's always a temptation to scoop up seemingly easy features. Beware statements like "It would be cool of the software could..." - or - "right here we could show..." Always ask what user role needs this? What will they be doing when they do need it? Does this user role care about this information? What information do they care about?

5. We elevate the writing of stories into interaction design.

   Help the folks who know the business understand user roles and task-cases. Before requirements are created discuss roles - who's important, who isn't. Discuss task cases – what does each role do. Clearly understand priority and dependence. This makes planning an iteration easier. This allows us to deliver a truly usable product sooner by appropriately accommodating all the necessary tasks of a focal role.

6. We revisit our requirements often.

   In implementing the software thus far, have we learned of a role we didn't know about earlier? Have we found that to accomplish a goal it may take unforeseen tasks or that some of our tasks are unnecessary? When we're not sure, we pull out the 3 x 5 cards and reassemble role models and task models to evaluate if the design still makes sense.

9. INTERACTION DESIGN AND AIM

9.1 Beck & Cooper Face Off.

In an interview posted Jan 15th 2001 on Fawcette Technical Publications website [14], Kent Beck and Alan Cooper face off on the subject of up-front interaction design vs. agile methods. Excerpts from the conversation include the following comments.

Cooper: "…I'm not talking about having a more robust communication between two constituencies who are not addressing the appropriate problem. I'm talking about incorporating a new constituency that focuses exclusively on the behavioral issues. And the behavioral issues need to be addressed before construction begins."

Beck: "OK, wait. I agreed with you very close to 100 percent, then you just stepped off the rails. I don't see why this new specialist has to do his or her job before construction begins?"

Cooper: "It has to happen first because programming is so hellishly expensive… There's enormous cost in writing code, but the real cost in writing code is that code never dies. If you can think this stuff through before you start pouring the concrete of code, you get significantly better results."

Beck: "No. I'm going to be the programming fairy for you, Alan. I'm going to give you a process where programming doesn't hurt like that—where, in fact, it gives you information; it can make your job better, and it doesn't hurt like that. Now, is it still true that you need to do all of your work before you start?"

I hear Cooper asserting that software is too rigid to easily change – that we must get interaction design right, all of it, before we develop. I hear Beck saying that we've eliminated the cost of change curve so we can now get it wrong without incurring great expense. It seems that both Beck and Cooper share the same goal of cost-effectively delivering high quality software that results in end-user satisfaction. They seem to disagree on how this is done. Could they both be right to some degree?

9.2 Building Better Aim.

If our goal is to deliver high quality software on time while satisfying end-users, then, that's a good target to aim for. I'd interpret Cooper as saying we need to hit our target with one carefully calculated shot, and the interaction designer should be the one to take aim. I'd interpreting Beck saying we can shoot

often and cheaply, so keep shooting until you hit your target. Let businesspeople take aim since they're paying for all of this.

In my experience, I've seen evidence that we can indeed shoot often and cheaply. But, I've also seen evidence that businesspeople don't always have the best aim. And, although XP and agile methods do help minimize the cost of developing working software and decrease the cost of changing it, cost is still cost. And businesspeople don't like paying unnecessary costs.

If this metaphor holds, then a working solution might be to try to improve the aim of the businesspeople by using interaction design concepts to help better define our requirements. If we can dependably and repeatably apply interaction design tactics we should be able to build better aim.

Our experience at Tomax bears this out. The simplicity and repeatability of U-CD allows the actual customer, business leaders, and developers to all participate in "designing" the requirements. During this process we all feel more confident that we understand what the software should do and why. We still miss our target sometimes, however good development practices do indeed allow us to change the design quickly. Also important is that when we do get it wrong we now understand a little better why. It's often an undiscovered user role, or goal. Using an interaction designer's sensibilities and U-CD as process framework, we are all learning to ask better questions – which gives us the better aim we've been looking for

10. ACKNOWLEDGEMENTS

Thanks to valued team-members from Tomax Technologies & Evant Solutions for providing a laboratory to learn in. Thanks to Larry Constantine & Lucy Lockwood for being great teachers. Thanks to collaborators and advisors: Stacy Patton and Kay Johansen. Special thanks for valuable feedback and advice goes to Alistair Cockburn for help in motivating and revising this paper. Thanks to Lougie Anderson for her advice, and encouragement. Thanks also to the enthusiastic team at Sabrix who allowed themselves to be guinea pigs.

11. REFERENCES

[1] Agile Alliance http://www.agilealliance.com

[2] Beck, K., Cunningham, R., A Laboratory For Teaching Object Oriented Thinking, (1989) http://c2.com/doc/oopsla89/paper.html

[3] Beck, K., Extreme Programming Explained, Adison-Wesley (1999)

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

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

[6] Constantine L. & Lockwood L., Software For Use, Adison-Wesley, (April 1999)

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

[8] Constantine L., Process Agility and Software Usability (2001) http://www.foruse.com/Files/Papers/agiledesign.pdf

[9] Cooper, A., About Face, Hungry Minds Inc. (1995)

[10] Cooper, A., The Inmates are Running the Asylum, Sams (1999)

[11] Fowler, M., Refactoring: Improving the Design of Existing Code, Addison-Wesley (1999)

[12] Jeffries, R., Anderson, A., Hendrickson, C., Extreme Programming Installed, Addison-Wesley (2000)

[13] Mathaissen

[14] Nelson, E., Extreme Programming vs. Interaction Design (2002) http://www.fawcette.com/interviews/beck_cooper/

[15] Schwaber, K., Beedle M., Agile Software Development with Scrum, Prentice Hall, (2001)

[16] Test Driven Programming http://xp.c2.com/TestDrivenProgramming.html

[17] Wirfs-Brock, <Which paper did she first document conversational use cases? Find this.>

[18] Extreme Programming Yahoo Group http://groups.yahoo.com/group/extremeprogramming/