2006

Cowboy: An Agile Programming Methodology for a Solo Programmer

Ashby Brooks Hollar
Virginia Commonwealth University

Cowboy:
An Agile Programming Methodology for a Solo Programmer

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science at Virginia Commonwealth University.

by

Ashby Brooks Hollar
James Madison University, Bachelor of Science, 1999

Director: Dr. Susan Brilliant,
Professor, Computer Science Department

Virginia Commonwealth University
Richmond, Virginia
December 2006

Acknowledgment

This body of work is dedicated to the late Pocket Buster-Linwood Hollar, the best friend and companion anyone could ask for. Also to Duffel Hollar, not Pocket's replacement but a good successor in my heart of hearts.

Mom and Dad, thank you being so supportive of me and always believing in what I could accomplish.

ABSTRACT VII

1- INTRODUCTION 1

2- CURRENT THOUGHTS ON COWBOY PROGRAMMING 1

3- BRIEF OVERVIEW OF AGILE PROGRAMMING METHODOLOGIES 5

3.1- CORE PRACTICES OF AGILE DEVELOPMENT 6

3.2- POPULAR AGILE METHODOLOGIES EXPLAINED 8

3.2.1 – EXTREME PROGRAMMING 9

3.2.2 – AGILE UNIFIED PROCESS 11

3.2.3 – SCRUM 13

3.2.4 – GETTING REAL 15

4- COWBOY – HOW AGILE PRACTICES CAN HELP 18

4.1- CORE ELEMENTS 18

4.2- CUSTOMER RELATIONSHIP 19

4.3- ITERATIONS 20

4.4- CODE 21

4.5- TEST DRIVEN DEVELOPMENT 21

4.6- OUTLINE OF THE COWBOY METHODOLOGY 22

4.7- COWBOY, READY TO TEST 24

5- PLANS FOR THE APPLICATION OF COWBOY 24

5.1- PROJECT OVERVIEW 25

5.2- CUSTOMER/PROGRAMMER RELATIONSHIP 25

5.3- MEETINGS 26

5.4- ARTIFACTS 27

5.5- COWBOY SUMMARY AS APPLIED TO VLP 28

5.6- SUMMARY 30

6- FINAL PRODUCT ANALYSIS 30

6.1- CUSTOMER'S IMPRESSION OF THE FINAL PROTOTYPE 30

6.2- DEVELOPER'S IMPRESSION OF THE FINAL PROTOTYPE 32

6.4 SOURCE CODE ANALYSIS 32

7- PROCESS ANALYSIS 33

7.1- COWBOY'S SUCCESSES 33

7.1.1- CUSTOMER/DEVELOPER RELATIONSHIP 33

7.1.2 - MEETINGS 34

7.1.3- TESTING SCRIPTS 34

7.1.4- ARTIFACTS 35

7.1.5- CUSTOMER QUESTIONNAIRES 35

7.1.6- ITERATIVE BUILDS 36

7.2- COWBOY SHORTCOMINGS IN VLP PROJECT 36

7.2.1- SOURCE CODE 36

7.2.2- MISSING FEATURES 38

7.3- LIMITATIONS 38

7.4- HAVING A CUSTOMER PROXY 39

8- COWBOY 3.0 - HOW TO IMPROVE THE PROCESS 40

8.1- PROBLEMS IN THE APPLICATION OF COWBOY TO THE VLP PROJECT 40

8.1.1- CUSTOMER / DEVELOPER RELATIONSHIP 40

8.1.2- TIME COMMITMENTS 41

8.1.3- TEST DRIVEN DEVELOPMENT 41

8.2- ADDITIONAL STANDARDS FOR PROFESSIONAL ADAPTATION 42

8.2.1- CODE REPOSITORY 42

8.2.2- CONTRACTS 43

9- CONCLUSION 43

APPENDIX A: SOURCES USED 46

APPENDIX B: ARTIFACTS 48

B-1: REPRESENTATIVE GOAL LIST 48

B-2: REPRESENTATIVE STUDENT TASK LIST 49

B-3: REPRESENTATIVE TEACHER TASK LIST 50

B-4: REPRESENTATIVE GLOSSARY 51

APPENDIX C: EXAMPLE PROXY CUSTOMER QUESTIONNAIRE 52

APPENDIX D: EXAMPLE MEETING AGENDA 54

APPENDIX E: EXAMPLE PROXY CUSTOMER TEST SCRIPT 55

APPENDIX F: FINAL CUSTOMER TEST SCRIPT 58

APPENDIX G: RDOC GENERATED DOCUMENTATION EXAMPLE FOR STUDENT CLASS 63

APPENDIX H: SOURCE CODE FOR STUDENT CLASS 64

APPENDIX I: COMPLETED PROXY QUESTIONNAIRES 65

APPENDIX J: VITA 73

COWBOY:
AN AGILE PROGRAMMING METHODOLOGY FOR A SOLO PROGRAMMER

By Ashby Brooks Hollar

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

Virginia Commonwealth University, 2006.

Major Director: Dr. Susan Brilliant, Professor, Computer Science Department

Abstract: Very little research in software engineering has focused on the model of a programmer working alone. These so-called cowboys are disdained for not working in teams to build software. In reality many cowboys work by themselves due to the circumstances of their work environment, not because they are unable or unwilling to work with well with others. These solo programmers could benefit from a methodology to assist them in consistently developing reliable software on time and within budget while satisfying the customer's needs. Cowboy was designed to help fill this void. This agile-based system incorporates the benefits of agile methodologies into a lightweight, customer-centered approach to software development for the lone developer. This thesis describes Cowboy and its successful application in developing a prototype for a web application.

1- Introduction

Most research in software engineering focuses on getting groups of people to work together effectively to write large software projects. But what about the programmer who *has* to work alone? What about the programmers who are forced to work on projects by themselves? These programmers are prevalent in software development today. Examples include a person working as the lone developer for a services company, or a programmer who is the only one working on a given platform for a larger project.

This paper uses the term cowboy programmers to refer to those who are predominantly working by themselves, not because they are anti-social and don't work well with others, but because the circumstances of the projects they work on mandate they must work alone. This paper outlines some of the current thoughts on cowboy programming, surveys some modern agile programming methodologies, explains how agile programming practices can be adopted to help cowboys create great software, and outlines an agile cowboy programming methodology, Cowboy. It then explores a real-world implementation of this methodology, explains the results and examines how it could be improved.

2- Current Thoughts on Cowboy Programming

When computer science was born, almost all software was written by a single programmer. The supporting hardware and programming languages at the beginning of the modern computer age didn't facilitate the production of complex, massive and widespread software in use today. As hardware sped up and programming languages matured, projects grew in both size and scope, eventually requiring people to work

together to produce large pieces of software. Still, the cowboy programmer persisted, not working with others but instead locking him/herself away until a working product emerged.

Many famous cowboy programmers have shaped the world we live in today. Alan Turing, a mathematician who might be viewed as a precursor to the modern programmer, was described as "a confirmed solitary" [Hodges] as most of his life's work was achieved while he was working alone. Bill Gates and Microsoft, Inc. essentially got their start when he worked alone to write MS BASIC over a five week period in 1975. [Mateosian] In 1991, while a student at Helsinki University, Linus Torvalds wrote the first Linux kernel throwing open the floodgates of open source software. In short, there always has been, and there will continue to be a need for cowboys.

Tex Curtis's article, "So You Wanna be a Cowboy?", draws many real-world comparisons between computer programmers and cowboys. In his mind, cowboys and programmers are not so different in the obstacles they must overcome, the long, thankless hours on the trail, many hours of solitary work, and their ability to work as a team in times of stampede.

> Maybe true cowboy programmers would be good after all. Cowboys and programmers both tend to be introverts who fiercely guard their independence. Yet others must be able to work with them, especially under stress. [Curtis]

He suggests that a "maverick" is a better term for the lone programmer who doesn't get along with others, quietly sitting alone in the corner, writing code and eating Cheetos.

Curtis [Curtis] offers a perspective about the strengths of actual cowboys and how these can be embraced by all software developers.

Today, the software development industry stereotypes cowboy programming as follows:

> (The lowest level of development) ...in software is associated with a focus on self-reliance. Software experts often refer to software developers operating at this level of awareness as mavericks, cowboy programmers, Lone Rangers, and prima donnas. Software developers at this level tend to have little tolerance for other people's ideas. They like to work alone. They don't like following standards. The "Not Invented Here" syndrome thrives.
>
> [McConnell]

McConnell goes on to say that this approach is adequate for environments that employ few programmers who work independently. However this "lone-wolf" approach scales poorly to larger projects or organizations. While some software developers do match this stereotype, not all lone-wolves can be so neatly characterized. This statement was directed towards large software development houses and not the smaller companies who only have a single programmer. In a small company a programmer can have his independence and solitude but must also have the skills to be able to extrapolate requirements and work with customers and the other departments of the company.

James Bach has written many articles on what he terms "heroes." Bach is of the opinion that too much time and effort is spent focusing on the processes of creating software. "Process is useful, but it is not central to successful software projects. The central issue is the human processor - the hero who steps up and solves the problems that lie between a need expressed and a need fulfilled." [Bach, 95] These single-programmer projects

require heroes to implement them, and even though cowboys are working alone on their projects, they are not alone in the world. Bach talks of the greatest tool he uses in his professional life: his peer network. Online communities and conferences are great places for those involved in software development to engage fellow professionals, seek and contribute to knowledge bases, and have work reviewed by others. He encourages all, "...to look up from your project, your technology, and your company, and join the great conversation of software engineering." [Bach, 99]

Rather than being a problem or menace, programmers who work alone write much of the world's software. The challenge for these cowboys is to produce consistent, easy-to-maintain code that comes in on budget and on time while meeting customer expectations. Indeed, these are the very same objectives of team-based software engineering methodologies, but as experience has shown, even software products developed using these methodologies largely fail to meet all their requirements. So how can a single software developer accomplish what even the most ably-managed groups of developers often fail to do? Perhaps cowboys should adopt the same agile methodologies that are becoming popular in the team-based realm of software development. The core principles of agile software development are truly a set of guiding principles and a way of thinking that are uniquely qualified to guide solo programmers to better software development.

3- Brief Overview of Agile Programming Methodologies

Agile Software Development began in 2001 with the creation of the Agile Software Manifesto, an online document that is a line in the sand drawn by seventeen men working to find better ways of creating software. [Beck, 01]

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Agile programming eliminates much of the documentation and process overhead of formal software development and replaces it with customer interaction, working code, anticipating and responding favorably to change, and respecting and valuing the individuals involved. It attempts to remove many of the stumbling blocks that have plagued software development from the beginning. Agile relies on working code and customer feedback to direct the project and make sure that features are added based on what the customer wants/needs, not just because an engineer thinks it is "neat."

The signatories of the Agile Manifesto realized there are many shortcomings in modern software development methodologies. For example, requirements and contracts must be set in stone at the beginning of the process, making change impossible, costly, or at the least very troublesome. Also, much time and effort is wasted on creating documentation,

diagrams, or other artifacts that are not required in the final deliverable. Since Agile strips away so much of the non-software writing processes of software engineering, many "old-school" developers think that any agile methodology itself is cowboy programming. [McBeen]

The signatories realized that contemporary software development methodologies did not adapt well to change. Changes to the initial requirements are often necessary during development. Requirements, painstakingly extracted from customers at the outset, may have been incomplete or misunderstood by the engineers. Customers may have been confused and communicated their needs inaccurately, or the needs of the customer may change over the development life-cycle. Non-agile methodologies attempt to deal with change by contractually eliminating it from the project and not allowing new or different customer needs to be considered without re-negotiating price and delivery date. Agile programming methodologies expect and even embrace change as a normal part of software development. [Beck, 99]

Common Agile development methodologies include: Extreme Programming, Scrum, Agile Modeling, Adaptive Software Development, Crystal, Lean, Agile Unified Process, ICONIX and others. Although these methodologies vary, they all have at their core the Agile Manifesto and allow the process to be customized for each individual project.

3.1- Core Practices of Agile Development

There are many general practices that are common to all or almost all agile development methodologies. The most fundamental common feature is iterative development.

Iterative development means that only a few features are in development at any given time and are finished before new features are implemented. This allows engineers to focus on the task at hand rather than being spread thin working on many different issues at the same time. It also makes sure that in large projects bottlenecks don't form by, for example, too much code coming out of development at the same time to be adequately tested before the release is due.

Most modern software development relies on a code repository (Subversion, CodePlex, SourceForge to name a few) to manage versioning and conflicts that arise if multiple programmers are working on the same file at the same time. In agile development, a programmer should never "check-in" code that is known to have defects. This way when a module is checked out, it can be modified and not be affected by a fix that has not yet been applied. If a defect is found in a current piece of code, the repository allows programmers to roll back to earlier versions if need be. Any code in the repository is considered ready to proceed from development to testing.

Refactoring is central to the Agile process. Refactoring is the re-working of existing, working code to make it more efficient or a better reflection of the customer's desires. Knowing that refactoring is central also means that a programmer is open, as opposed to resistant, to reworking parts and pieces of his code. Refactoring is particularly common and useful in interface design. In the final analysis, most customers don't differentiate between the software and the user interface. Since the user interface is the only part of

the program they are aware of, it is the source of nearly all of their comments regarding the software's performance.

Much documentation is abandoned in agile programming in favor of working code, so it is imperative that code is well documented, using intuitive names for classes, methods, variables and constants. Automated tools (RDoc, JavaDoc, C#XML) are used whenever possible for generating external documentation.

Test-driven development is prevalent in most agile methodologies. Test-driven development mandates that programmers include unit tests in all the code that they develop, and if applicable, interaction or functional tests as well. Generally speaking, it is considered a good practice to write these tests before the code because when the code passes the tests, it is done. This reinforces the idea of never checking broken software into the repository. Writing tests first also helps clarify the requirements for a given module. Tests can help eliminate logical errors in code and ensure that the code solves the correct problem. In fact, many modern repository tools can automatically run these tests at check-in so "broken code" never sneaks into the repository.

3.2- Popular Agile Methodologies Explained

As previously mentioned, many agile methodologies have popped up in the last five years. Many of them are fully-specified, stand-alone development methodologies and have components that were borrowed for Cowboy. Of these, Extreme Programming and Scrum are two of the most widely used, according to the authors of their official websites. [Schwaber, 96] [Wells]Two other methodologies, Agile Unified Process and

Getting Real, also contain methods that may be extrapolated for application to the Cowboy model.

3.2.1 – Extreme Programming

Extreme Programming (XP) is approximately eight years old and was developed by Kent Beck in conjunction with Ward Cunningham while they were both working for DiamlerChrysler in the late 1990s. Together they had a very pleasant software development process that was simple and very efficient. The ease of this development resulted in Kent contemplating both what makes software development easy and what makes it hard. Kent came to the conclusion that software developers can improve any software project by focusing on communication, simplicity, feedback, and courage. (Courage reminds us of Bach's call for heroes.) From his experience with Ward and the success of their project at DiamlerChrysler XP was born.

XP is designed to work best with two to twelve developers. They work in teams while writing production code, one writing with the other looking over his/her shoulder, switching off periodically. This is called pair-programming and has been shown to increase the quality of the code generated, the abilities of the developers involved, and the speed at which working code is produced. A hands-on and respectful management layer is very important, as are dedicated testers, and the participation of a customer who is available at all times to answer questions and give feedback. All participants typically work in proximity with each other with many posters (or other visual aids) tracking different metrics outlining the current status of the project. Each day starts with all team

members standing in a circle for a short stand-up meeting where problems, solutions and each sub-team's focus is discussed. This stand-up format encourages all the participants to remain focused and helps keeps the meetings short.

An XP project starts with the collection of user stories from the project stakeholders (customer or stand-in for the customer). "User stories capture high-level requirements, including behavioral requirements, business rules, constraints, and technical requirements." [Ambler, 02] User stories should be unambiguous, clearly indicating which category of user is trying to perform which task, and be small enough to be considered atomic to the developers. XP suggests that if a user story reveals a particularly challenging or new problem, "spike solutions" should be created to investigate potential approaches to a solution. Spike solutions are simple, throwaway programs developed to give experience to the programmers so they aren't experimenting while developing the final application. User stories are also used to create the acceptance tests that are applied to the project to find bugs, which create new tasks, before being submitted to the customer.

Projects are broken down into a series of small releases, each one having been fully tested and approved by the customer. Each release consists of many iterations in which individual features are added to the system, bugs are fixed, and existing, working code may be refactored. These small steps inside iterations are referred to as tasks. Features and functionality are only added at the specific request of the customer, and developers should never add extra functionality that isn't absolutely required by these requests.

"Only 10% of that extra stuff will ever get used, so you are wasting 90% of your time." [Wells] Pairs of programmers may complete many tasks each day, making sure that each new feature or object added passes the unit tests they also develop. Each task is worked on by only one pair of programmers at a time. Assuming the new code passes all the unit tests, the code may be checked into the repository and considered complete and ready for the testing department. [Wells]

In the realm of design, the user stories are used to create Class, Responsibilities and Collaboration (CRC) cards which are used to model the design of the system. Individual cards are used to represent objects and are developed through team-based discussion. A system metaphor and glossary should be adopted to keep the team focused by standardizing the naming of classes and methods. Simplicity should be the deciding factor when negotiating on options for how to model or implement a feature. XP advocates always doing the simplest thing that can possibly work. [Wells]

3.2.2 – Agile Unified Process

The Agile Unified Process (AUP) was developed by Scott Ambler as a simplified, agile version of IBM's Rational Unified Process (RUP). One of the major simplifications from RUP to AUP is focusing less effort and manpower on the creation and maintenance of models and documentation, instead striving to make them "just barely good enough." Also, requirements management is designed to deal with changing, new and removed requirements as any agile development process should do. AUP as a methodology also uses many agile techniques to make a very large and complete agile toolkit including Test

Driven Development (TDD), Agile Model Driven Development (AMDD), Agile change management, and database refactoring. [Ambler, 06]

Ambler outlines four phases of the Agile Unified Process as follows:

- Inception: Defines the initial scope, potential architecture, funding, and stakeholder acceptance for the project.
- Elaboration: Proves the architecture is feasible and well designed.
- Construction: Incremental building of working software.
- Transition: Validate the final system and deploy it to a production environment.

Compared to XP, AUP requires much more up-front work before coding begins. The construction phase consists of many iterations that typically end in development releases. Development releases initially lead to production releases. Development releases are purely internal while the production releases are at least conceptually deployable to a production environment. Each development release is stored in what is termed a "staging area," a repository capable of associating notes with bits of code. These notes specify the state of the modules in the repository, for example to denote what works and what does not or what remains to be done to complete a given module. [Ambler, 06]

Managers of Agile Unified Process driven projects must adopt some basic principles to be successful:

- *Your staff knows what they are doing.* Allow your staff the freedom to do their job and simply help with high-level guidance and training if needed.

- *Simplicity.* Resist the temptation to specify requirements, models and other artifacts in anything other than a few pages.

- *Agility.* AUP was designed with the Agile Manifesto in mind.

- *Focus on high-value activities.* Make sure that developers are developing features that are known to be of high-value to the customer/stakeholder and are not distracted by things they think are "neat" or things they (the developers) *think* add value.

- *Tool independence.* Use any tools you are comfortable with and meet your needs for any given project. Consider, however, using the simplest available tools (whiteboards, sticky notes, index cards, etc.) or open source tools over complex or high-cost tools.

- *Tailor AUP to meet your project's needs.* Parts and pieces of AUP can be adjusted or removed to suit your particular needs. [Ambler, 06]

3.2.3 – Scrum

Scrum was initially conceived as a new way to handle manufacturing product development in 1986, and later elaborated upon in 1995 by Ikujiro Nonaka and Hirotaka Takeuchi. Ken Schwaber of Advanced Development Methods, based on research in

process theory performed while working with DuPont, formulated a formal definition for Scrum as applied to software development in 1996. Scrum is not a stand alone development methodology; rather it is meant to be an agile "wrapper" for existing engineering processes. It manages and controls development work incrementally (in bursts of work called "sprints"), improves communication, removes hurdles to development, all in an environment with rapidly changing requirements. [Schwaber, 06]

Once Scrum teams are formed, a Scrum Master is appointed, "who conducts the Scrum meetings, empirically measures progress, makes decisions, and gets impediments out of the way of slowing or stopping work." [Schwaber, 06] Although not technically a member of management, the Scrum Master leads a daily meeting in which he or she asks team members what they did since the last Scrum meeting, what got in their way, and what they are planning on doing before the next Scrum meeting. Decisions about features, implementation and design issues are made by the Scrum Master who also develops the initial backlog. A backlog is a prioritized list of features, bug fixes, and system improvements.

Scrum relies heavily on a backlog of all work that needs to be done to complete the project. Backlogs contain immediate needs that are well-defined as well as features to be implemented in the future that may not be clearly understood. A single team member is appointed to prioritize the items on the backlog, and the team will work together to decide what features will be implemented in the next sprint. A sprint is a single iteration with a fixed length, usually between two weeks and one month. The features assigned to

a given sprint are also prioritized so that if features need to be cut because of time or cost concerns, it is clear which ones have lower priority. [Schwaber, 06]

As mentioned previously, there is a daily Scrum meeting. It is held in the same place and time every day to avoid confusion. It is limited to 30 minutes and encompasses only the Scrum Master asking the team members the three questions above. Any other issues that arise and require discussion will be handled at later single-issue meetings with only the interested parties attending. This consistency of meeting place, time and format allows developers to get into a comfortable "groove," allowing them to be more productive, with an appropriate amount of supervision and support to take care of issues and impediments that may arise. [Schwaber, 06]

3.2.4 – Getting Real

Getting Real (Real) was developed by 37signals in Chicago, IL. It is a "smaller, faster, better way to build software." [37signals] This methodology focuses primarily on developing web applications, but many of the ideas it encompasses can be applied to other categories of software design. The primary philosophy of 37signals is that software is too complex and offers too many features. Their products are very simple with a limited number of features that are very easy to use and extremely simple to understand. Help files and tutorials are not required when using these applications because the applications are self-explanatory and elegant.

Real begins with user interface design. The user interface is the only part of the system that users will ever see, so Real suggests getting it right and building the back end around

the working and customer-approved front end. Real advocates that formal specifications are an illusion and producing anything that simply represents reality should be avoided in favor of producing real software. [37signals]

Real encourages designers and programmers to embrace constraints. No project ever has sufficient resources (people, money, time, etc.), so accept what you have to work with and find creative solutions to problems. Working within constraints breeds focus and innovation. "Constraints are often advantages in disguise. Forget about venture capital, long release cycles, and quick hires. Instead, work with what you have." [37signals]

Focusing on the big idea of a project allows Real to focus on development and delay many decisions until they absolutely have to be made. Specifically, working from large to small problems ensures that user interfaces have necessary components and features in place while allowing time for small user interface (UI) tweaks and small corrections to the functionality once the "heavy lifting" is done. [37signals]

With respect to feature selection, Real advises taking your initial product and cutting it in half. Focus only on what is truly essential, the absolute core of your application; get that working and then add functionality to the solid foundation that you have built. Real says that the alternative to building half a product is building a "half-assed" product. When the stakeholder is interested in adding features to the system, the designers' default answer should be "no." This is a difficult concept at first but essential to develop a

product that does exactly what the system requires and does not include many useless features. [37signals]

All these ideas and nuggets of advice are incorporated using a straightforward process. The Real software development process is not as rigid as many other methodologies, but clearly outlines how to manage projects. First do everything in your power to produce running software quickly. Running software builds momentum and builds the confidence of the team. Second, work in iterations and deploy non-perfected software as soon as it is running. Getting real-time feedback from actual end users will lead to better decisions and reveal errors as the project matures. Iterations also allow developers the freedom to not get it working perfectly on the first try. Third, determine what the big questions are and how these ideas will be implemented. Start with brainstorming, move to sketches on paper or white-boards, and create a working UI prototype for each feature. Then after iterating the UI prototype until it is acceptable, add the application code to the back end. Fourth, avoid offering preferences to the end user. Go ahead and make decisions for the user rather than offering choices unless it is absolutely necessary. Fifth, become comfortable with the word "done" and accustomed to moving on to the next challenge. Don't worry about making a bad decision or leaving something unimplemented because going back and making a change shouldn't be a big deal. Sixth, test your application in the real world with real users. This is the fastest way to get a lot of feedback very quickly. Obviously there are many software projects where this approach would not be acceptable, but it is invaluable when your project can allow for it. Lastly, don't plan

large increments that will require long periods of time. For example, if a portion of a project will take ten weeks, break it into ten one-week projects that are easier to manage.

4- Cowboy – How Agile Practices can help

Programmers working by themselves do not typically adopt formal methodologies, but rather bang away at a solution until they feel they have finished. Designed to be lightweight, Cowboy borrows heavily from the core agile practices as well from XP, Scrum, AUP and Real. It is an attempt to integrate the best of those practices and creates a process that allows programmers to stay focused while creating software customers want, on time and inside budget constraints. No specific metrics were utilized to select the features included and excluded from Cowboy; rather components that could be applied by a solo programmer and that seemed to mesh and complement each other were identified. The goal is a process easy enough to use and helpful enough to be a more attractive option for programmers working alone than using nothing at all. As a methodology, it is divided into four major sections: overall practices, customer interaction and requirements specification, iteration management, and deployment.

4.1- Core Elements

Certain elements of Cowboy are simply good practices and rules of thumb that a developer should keep in mind while creating software solo. Borrowing from the core agile practices, Cowboy is iterative with each cycle adding features and fixing bugs of the previous cycles. Each cycle should have specific requirements that will be implemented during the cycle.

XP-recommended use of spike solutions, writing small pieces of code to investigate unknown or rough coding terrain, is applicable to this methodology as well. The potential pitfall, however, is using a spike solution as part of the final product. Spike solutions, like all prototypes, should be discarded when final development starts.

Another good practice is keeping a Scrum-like backlog of work to be done for the entire project, with a detailed sub-list for the current iteration. The backlog is really a to-do list with items crossed off when finished and new items added when issues (including bugs) arise.

Finally, Cowboy suggests you keep your artifacts simple and just barely good enough, as suggested by AUP. The artifacts should encapsulate the core ideas and system architecture, but are subject to change (or even elimination) as the requirements change, so not much time should be lost perfecting them.

4.2- Customer Relationship

Since agile practices discourage spending a lot of time solidifying requirements before the project begins, Cowboy relies heavily on having a customer (or at least someone knowledgeable to represent the customer's interests) available to discuss and specify requirements as they arise during development. The customer's input should weigh heavily in determining the order in which features are added to the system. The user stories and glossary that are created in XP to gather requirements are also good tools for the solo programmer. Customers can understand the concept of user stories and relay how specific users will use the system to accomplish tasks without having to understand

or think about the layers of code and design. A glossary is necessary to clarify what words mean in the context of the project.

Dealing with customers frequently will require compromise and explanation of constraints. Cowboy applies the Real methodology of customer interaction. Embracing constraints like time, money, and experience (for example) and understanding the limits in scope these constraints will enforce before meeting with the customer can help keep the project smaller and more manageable. Also, the developer is encouraged to make his default answer "no" when customers ask for features to be added to the system. Customers may get excited and ask for more and more small features that will, in reality, hardly be used in the final product and detract valuable time from the development of more important features. If customers have to fight to get a feature added, it makes them think more deeply about how it will be used and why it should be included, making it a better feature if it is accepted. Developers should also try to offer the customer as few preference choices as possible. This holds true in requirements gathering as well as in the final product. The developer should feel empowered to make decisions, and, if this results in the occasional wrong decision from the viewpoint of the customer, be comfortable with refactoring to make the necessary changes.

4.3- Iterations

During every iteration, as code is written in Cowboy, the following guidelines should be followed. A cowboy should try to restrict the maximum iteration length to no more than two weeks. Tackling the large problems first and moving on to smaller issues is

conceptually described by Real and would also imply that spike solutions should be developed at the beginning of the project. Refactoring code written in this and previous cycles should be an ongoing task and something that the developer does not hesitate to do. As code is written, sufficiently detailed comments should be included and updated during refactoring. Comments should be in a format that allows a tool such as JavaDoc or RDoc to automatically extract and create formal documentation for the code. The developer should keep in mind that these documents will largely replace the formal requirements documents and artifacts of a conventional software development methodology.

4.4- Code

Code should be kept organized and preferably in a code repository that is capable of recording notes in keeping with AUP's requirement of a staging area. If this is not a practical or desirable process for the developer, then at the very least, backups should be maintained, and comments at the top of each file should indicate what known bugs exist in the file.

4.5- Test Driven Development

Test-driven development is core to agile practices and should be practiced in Cowboy. Unit and functional tests should be developed up front and run often to ensure the quality of the current code base. This attention to testing while developing facilitates two components of Real that are central to Cowboy. The developer should focus on building and deploying non-perfected software as quickly as possible. Running software builds

momentum and makes it easier to get feedback from your customer/stakeholder on what direction to take next. If working on a project that may be deployed before it is finished (especially if web-based), a lone programmer can take advantage of his/her connections in the software development community for additional feedback and end-user testing in a real-world environment.

4.6- Outline of the Cowboy methodology

The following outline describes all of the basic components of the Cowboy methodology. As with any agile methodology, this is a template that can be modified to work for a specific project.

I. Customer / Developer Relationship
    a. If the customer is not available for meetings, then designate a customer proxy to stand in at meetings
    b. Use a questionnaire to ensure that the customer is comfortable and happy with the progress on the project and the meeting format
    c. Use tools such as e-mail, instant messaging and phone calls for the developer to ask questions of the customer during development

II. Meetings
    a. Conduct an initial design meeting to determine the initial features
        i. Provide a comfortable room with sufficient table and whiteboard space
        ii. Define overall goals for different classes of users
        iii. Define an initial, prioritized task list for the developer
        iv. Develop a glossary for possibly ambiguous terms
        v. Design a preliminary user interface
    b. Build-review meetings

i. Begin meetings by providing the customer with a walk-through testing script of the newest release

ii. The developer should take notes on the customer's oral comments and observations on problems the customer encounters to identify refactoring tasks

iii. Customer and developer revise goal list

iv. Customer and developer revise and re-prioritize task list

v. Customer and developer add to or edit glossary as needed

vi. Customer and developer discuss user interface changes

vii. Customer responds to questionnaire

III. Artifacts

a. Goal list

i. Goals should be kept as high level as possible, identifying needs of the end user that the system will satisfy

ii. One, or preferably more, tasks will be derived from each goal

iii. Categorize goals according to end-user group

iv. Each individual goal should be a phrased as a complete sentence and use verbs like "allow, provide, enable"

b. Task list

i. Tasks are the actions the system allows an end-user to perform

ii. Each task is prefaced with the phrase, "A (user class) should be able to…"

iii. Group tasks into categories based on the goal or major feature they satisfy or support

c. Glossary

i. Create entries in the glossary to define both important terms and terms that may be interpreted incorrectly

ii. Ensure that the customer and developer both agree that the definition is clear, concise, and correct within the context of the project

iii. Keep glossary handy for all meetings

d. Code

   i. Use Integrated Development Environment (IDE) , preferably with code completion support

   ii. Follow conventions for formatting of code to make code consistent

   iii. Choose meaningful, and if necessary verbose, names for classes, methods, and variables to improve readability

   iv. Comment classes and methods appropriately

   v. Use unit testing to verify each class's behavior

   vi. Maintain a to-do list that mirrors the task list, including refactoring tasks and user interface changes

4.7- Cowboy, Ready to Test

While not a finely-honed software development methodology, Cowboy is specified clearly enough for a lone programmer to use. Section five outlines a specific test application of Cowboy to a real software development project.

5- Plans for the Application of Cowboy

As my personal interest in agile development grew, and as I began incorporating agile fundamentals into my school and work projects, I began to search for a large project to which I could apply this solo programmer methodology. In the fall of 2005 I began meeting with Dr. Pam Taylor of the Virginia Commonwealth University Art Education Department to discuss a software product she wanted to have developed. In the Spring of 2006 I realized that Dr. Taylor's VLP project would be an excellent vehicle for testing the core concepts of Cowboy.

5.1- Project Overview

Dr. Pam Taylor wanted a prototype to demonstrate her concept of a "Virtual Learning Portfolio" (VLP). The underlying goal of this project was to develop a working prototype system that would allow Dr. Taylor to apply for patents of her concepts of interactive learning using hyper-linking technology. VLP is envisioned as an interactive, online tool that supports art education in grades K-12. Students create portfolios by uploading digital versions of art, usually their own, and linking these to other items in the portfolio or to items on the web. Teachers supervise students' portfolios as they grow over a given semester or school year, adding comments and concretely assessing (via rubrics) the media and links to internal and external resources added by the students. After this prototype is developed, the end goal is to create a system that would be able to also track Standards of Learning for Art Education as well as have a cross-platform, three-dimensional or "Virtual Reality" interface to replace the web-based interface being implemented for this project.

5.2- Customer/Programmer Relationship

Katie Helms, one of Dr. Taylor's Graduate Assistants, agreed to act as proxy for the customer, Dr. Taylor. Dr. Taylor's schedule was not open enough to be involved with the meetings, but Ms. Helms felt comfortable in representing the wishes of Dr. Taylor. I proposed to work with two-week build cycles, and Ms. Helms agreed. We planned to meet at the end of each build cycle to review the project and plan the next phase.

5.3- Meetings

Meetings were designed to be as short as possible with very clear goals that included making sure that Ms. Helms felt comfortable with both the project's progress and her role in the project. Before each meeting the developer prepared and printed a written agenda and hard copies of all artifacts for both the customer and developer. Designing software can be a daunting process, even for trained professionals, so much care was taken to make sure feelings weren't hurt and that customer and developer were both happy and productive during the meetings.

All meetings were to be held in a room with plenty of table space and a whiteboard. The initial design meeting was organized into four parts. First, the high-level goals of the final product were defined. These goals were written with verbs like "allow, provide, and enable" to create active and precise sentences. Second, a task list was derived using the goals as jumping-off points. For this project, the list was divided into a list for student tasks and another for teacher tasks. Tasks are written in subject-verb-object sentences where each started with the fragment, "Using the system, a student/teacher should be able to…" As potentially confusing terminology was discovered, notes were made for a glossary of terms that will be created with definitions agreed upon by Ms. Helms and me. Third, the task list was prioritized into things to do first, second and third. Last, a preliminary user interface was sketched using the white board so development could begin.

Subsequent meetings began with usability tests. Ms. Helms followed a testing script to perform tasks inside the current system while I took notes. She was asked to speak up when she encountered bugs or anything that she didn't understand. After completing the script, she was free to "play" with the software and demonstrate changes she wanted or new features that should be added. Once this testing and review process was over, I had created a list of refactoring tasks needed by the system. These fixes were categorized as bugs, unexpected behavior, missing behavior, and user interface design issues. Next the goal and task list artifacts were reviewed, and items that were mutually decided to be 100% complete were crossed off. Individual goals and tasks were re-worded, added or deleted at this time. The remaining tasks were prioritized again with the understanding that the refactoring tasks would be completed first.

A questionnaire was given to Ms. Helms at the end of each meeting to track her concerns, observations and feelings about the project, and attempt to collect useful data about programmer-customer interaction. The questionnaire was comprised of questions that required ratings on a scale of one to five and short answer questions. An example of these questionnaires can be found in Appendix C.

5.4- Artifacts

While the code was being written, the only required artifacts were the goal and task lists and the code. The goal list and task list were stored electronically and printed at each meeting. Changes made to the hard copy during meetings were also made in the electronic version and saved as a new version.

The code was to follow certain coding guidelines. Class, method and variable names were to be self explanatory and unambiguous. Ruby on Rails formatting standards, including capitalization, scope, and indentation standards, were also adopted. Every source file, method and variable was to be well commented in a format that supports an automated comment tool, in this case RDoc. Unit and interaction testing were to be built in and run before each iteration was deemed complete. A statistics gathering program was to be used to track the lines of program, test and comment code at each build.

5.5- Cowboy Summary as Applied to VLP

By slightly modifying the previously presented outline, VLP was implemented using the following plan.

I. Customer / Developer Relationship
    a. Use a proxy for the customer
    b. Use questionnaire to ensure that the customer's proxy is comfortable and happy with progress
    c. Use tools like e-mail, instant messaging and phone calls for the developer to ask questions of the customer's proxy during development

II. Meetings
    a. Initial design meeting
        i. Provide comfortable room with sufficient table and whiteboard space
        ii. Define overall goals for different classes of users
        iii. Define initial, prioritized task list for developer
        iv. Develop glossary for possibly ambiguous terms
        v. Design preliminary user interface
    b. Build-review meetings

i. Customer's proxy walks through software with a testing script, articulating concerns encountered

ii. Developer takes notes on customer's comments to build refactoring tasks

iii. Revise goal list

iv. Revise and re-prioritize task list

v. Add to or edit glossary as needed

vi. Discuss user interface changes

vii. Customer's proxy responds to questionnaire

III. Artifacts

a. Goal list

i. Goals should be kept as high level as possible, identifying needs of the end user that the system will satisfy

ii. One, or preferably more, tasks will be derived from each goal

iii. Categorize goals according to end-user group

iv. Each individual goal should be a phrased as a complete sentence and use verbs like "allow, provide, enable"

b. Task list

i. Tasks are the actions the system allows an end-user to perform

ii. Each task is prefaced with the phrase, "A (teacher or student) should be able to…"

iii. Group tasks into categories based on the goal or major feature they satisfy or support

c. Glossary

i. Create entries in the glossary to define both important terms and terms that may be interpreted incorrectly

ii. Ensure that the customer and developer both agree that the definition is clear, concise, and correct within the context of the project

iii. Keep glossary handy for all meetings

d. Code

i. Use Integrated Development Environment (IDE) , preferably with code completion support
ii. Follow conventions for formatting of code to make code consistent
iii. Choose meaningful, and if necessary verbose, names for classes, methods, and variables to improve readability
iv. Comment classes and methods appropriately
v. Use unit testing to verify object's behavior
vi. Maintain to-do list that mirrors the task list, including refactoring tasks and user interface changes

5.6- Summary

The goals of being lightweight and simple appeared to be well outlined in the project plan. Both the customer's proxy and developer agreed at the first development meeting that the meeting schedules, artifacts, and questionnaires seemed reasonable and easy to stay on top of.

6- Final Product Analysis

Overall, this project was highly successful. The prototype met almost every goal and was developed and deployed on time. A couple of the "nice to have" features were not implemented, but their absence should not affect the patent application process, so customer was satisfied.

6.1- Customer's Impression of the Final Prototype

Dr. Taylor was periodically updated on the status of the project, and she saw an informal demonstration shortly before the final delivery date. At this time, she made it clear that she "didn't see the metaphor" the system was based on, but Ms. Helms helped to

communicate the similarities between VLP and the hyper-linking software Dr. Taylor was used to. Ms. Helms's explanations seemed to clear things up significantly. However, special care was taken while developing a final walkthrough script to clearly describe the task the user was trying to do.

After the system was deployed onto a live server, Dr. Taylor was provided with a comprehensive walkthrough script to demonstrate the functionality of the software. Overall, Dr. Taylor was pleased with the prototype. As an art educator, she was unhappy about the look of the interface, but this was not surprising. She was informed before development began that graphic design and the ability to create great-looking web pages were not strengths that this developer possessed. As far as functionality was concerned, Taylor was pleased with what was supported and only saw a couple of features missing. However, she did express concerns about how well the application would scale for both teachers and students when the portfolios contained a lot of data.

Dr. Taylor was also concerned with the "speak" of the user interface. She felt that the language used to describe the metaphor could have been clearer and more focused on education. Some specific ideas were expressed to help resolve this issue, but further work with Dr. Taylor, Ms. Helms and the developer will be needed to adjust the language used in the graphical interface to correct these issues.

6.2- Developer's Impression of the Final Prototype

Overall I was very pleased with the quality and functionality of the final prototype that was delivered. I was relatively new to Ruby on Rails; also this was one of the largest projects I had ever worked on and the largest I had worked on by myself as a cowboy.

Midway through the project, the code was migrated from a Windows platform using RadRails (a special, rather buggy, build of Eclipse) to Mac OS X using TextMate as an IDE. Ruby on Rails (Rails) was originally developed on a Macintosh, and TextMate has many helpful functions built into it (by the author of Rails) that made development much easier. The Macintosh operating system proved to be a lot friendlier for Rails development than Windows.

I agreed with Dr. Taylor's final assessment of how the VLP prototype looks. It is plain and a very simple. However, I focused on creating an easy-to-use website with the desired functionality, leaving the design of the final site to a graphic designer to be hired later. The flexibility to allow after-the-fact changes to the user interface was achieved using Cascading Style Sheets (CSS). CSS allows the separation of the format of the rendered pages from the content. Separating the content and formatting promotes agility by making future changes to the interface easier. A designer can polish the user interface by editing the CSS without requiring many changes to the content-generating code.

6.4 Source Code Analysis

The source code looks very clean and well formatted. Both IDEs used in the authoring supported tabbed browsing of source files, indentation assistance, code completion, and

syntax highlighting. This support from the authoring tool made it much easier to produce clean code. Ruby on Rails encourages keeping class names simple, and the developer strove to name methods very clearly. For example, `Cluster.resources_for_student(student)` returns an array of resources from the cluster for a given student. `Student.unseen_comments` returns all the comments for a student that have not been displayed previously.

Statistical tools were not used to analyze the code during development because very few comments ended up being required, and unit testing was not completed. (See section 7.2 for further discussion of this point)

7- Process Analysis

This project was an experiment that tested the hypothesis that a proposed development methodology would work well. By adhering to the process much was learned about what was good and what didn't work well with the application of Cowboy to the development of VLP.

7.1- Cowboy's Successes

Overall, the developer, final customer and her proxy thought that the final product produced using the Cowboy methodology was a success. The process itself also proved to be successful.

7.1.1- Customer/Developer Relationship

Ms. Helms and I got along very well for the four months we worked together. We felt very comfortable with each other, and meetings went very smoothly. Many e-mails and

phone messages were sent between meetings to clear up my questions or to communicate thoughts Ms. Helms had that had not occurred to her during the meetings.

7.1.2 - Meetings

Meetings ran very smoothly. Having hard copies of all artifacts and an agenda prepared before they started allowed meetings to stay on task and be as short as possible. During these meetings, there was much compromise between the developer and the customer's proxy. The order of feature development, the actual features to be developed during the next iteration and changes to the requirements all had to be decided upon. Ms. Helms and I had a very professional and friendly rapport, and worked very well together even when compromises were necessary. Cowboy's use of a meeting agenda and customer questionnaire would help keep this relationship on track if personality conflicts did exist.

After the first couple of meetings schedule conflicts made it difficult for meetings to be held on campus, so subsequent meetings were held in my living room. This was not seen to be a problem, since the software was working quite well, and the whiteboard was no longer necessary to help flesh out ideas.

7.1.3- Testing Scripts

The testing scripts described tasks for Ms. Helms to perform, rather than telling her exactly what buttons to press and what to type. This tested the intuitiveness of the user interface. I took notes on my observation of problems and Ms. Helms' comments during the tests. Conversation about the software was kept to a minimum during the tests, and issues raised were discussed after the testing was completed.

7.1.4- Artifacts

The goal and task lists proved to be a very handy form of communication between developer and customer. One significant strength was that both parties could enforce the standards that had been set forth at the beginning of the project.

At the conclusion of each meeting, I would update the artifacts to reflect the decisions reached and would then update a to-do list housed in the source code for quick access. A to-do list was added to the artifacts because I found it to be a great aid while writing code. This list of bug fixes, refactoring tasks, features to add, and user interface tweaks was very handy since it was only a single click away while development was going on. It could also be added to when bugs were discovered while authoring without cluttering up the artifacts themselves.

7.1.5- Customer Questionnaires

The customer questionnaires were an easy and reliable tool in reassuring the developer that the customer felt comfortable and that she understood what was being accomplished. Ms. Helms was encouraged to be as honest and open as possible because any constructive criticism would be used to improve the process. Information obtained from the questionnaires ranged from user interface issues, program bugs and how Ms. Helms was feeling about the process as a whole.

Customer questionnaires were incorporated into Cowboy primarily so that the developer can be assured that the customer is comfortable and regards the progress being made as satisfactory. The questionnaires received during this experiment did just that: reassure on

paper that the developer was doing a good job and that a happy, comfortable customer appreciated his efforts.

7.1.6- Iterative Builds

Adding prioritized features in successive build cycles accommodated the developer and customer very well. Each iteration met the goal of developing all of the features assigned to it. The developer didn't feel overwhelmed, and it was easy to ask questions of the customer's proxy in the middle of a development cycle since Ms. Helms could easily remember what features were being worked on during that cycle.

7.2- Cowboy Shortcomings in VLP Project

Despite all interested parties being happy with the final result, not all aspects of the VLP prototype project worked out exactly as planned. Specific issues are outlined below, and the limitations that led to these shortcomings are investigated in Section 7.3.

7.2.1- Source Code

The areas where this application of Cowboy deviated from the initial plan were manifested in the lack of comments and unit testing. Although the source code is not barren of comments, commenting classes and methods whose functions were very obvious from their carefully-chosen names never became a priority. Code written in Ruby on Rails normally relies on many very short functions. The longest method in VLP is about twenty lines and did not, in the developer's opinion, require a comment to describe it. The html documentation produced by RDoc lists the methods of a class and the values passed into the methods. Every Ruby method returns a value, so return type is

not automatically documented. The name of each method should describe the value that is expected to be returned. The following screen-capture is representative of the documentation RDoc generates for a class with no comments and serves as an example of why I decided adding additional comments would have not been particularly helpful.

As seen in Appendix G, the `in_course?` method returns a Boolean value which corresponds to whether or not a student is in a given course. The other three methods return arrays, so the method names imply plurality. The arrays contain types indicated by the name; for example, `unseen_comments()` returns an array of comment objects that have not been seen by the student.

There are three reasons that unit testing was not included. First, the developer did not have any understanding of or experience with building unit tests. Second, unit testing finds rarely-encountered, potentially lingering problems that may not appear until later in development or even after system deployment. Since VLP was developed as a prototype, rarely-encountered bugs are acceptable, and the primary functionality of the system has been proven through end-user testing. In a production system, unit testing would help assure that the quality of the system was being maintained as development moved forward. A third reason relates to the fact that unit testing is critical in allowing parallel development of modules in a team development environment. Obviously this reason for performing unit tests does not exist for the cowboy programmer.

Since comments and testing code did not become a major part of the VLP system, tracking the ratio of application, testing and commented lines of code was not performed.

7.2.2- Missing Features

There were two classes of features identified by the customer as missing from the final VLP prototype: features from the initial goal list that were not implemented and features revealed in Dr. Taylor's final comments that were unknown to the developer and customer's proxy. Neither the proxy nor the developer considered the features from the initial goal list that were not implemented a priority during the development process. Adding them on to the final prototype or incorporating them into the next incarnation of VLP will not require major changes in the system and their absence will not affect the patentability of the ideas illustrated by the prototype

The missing features previously unknown to the developer and customer's proxy were larger in scope and would require much more development effort to implement. As stated previously, the end goal of a final working VLP incorporates a three-dimensional interface. Many of the concepts and metaphors that a two-dimensional version of this VLP implementation is missing could, in Dr. Taylor's opinion, be illustrated by adding graphical elements and extendibility that weren't seen as priorities by the proxy or developer for the prototype.

7.3- Limitations

The primary limitation of this project was that both the developer and customer worked on this project part-time. Furthermore, both were working two jobs and working on other

major projects. The scheduling limitations of Dr. Taylor also prevented her from taking a more active role. Better planning and better communication with Dr. Taylor could have led to the incorporation of the missing features.

Another side effect of developing VLP part-time was the length of the build cycles. Although the amount of work planned and completed did not increase, the number of weeks between meetings averaged about three to four instead of two. Also, the developer left the country for a week and a half in the middle of the project, cutting out almost all of the time allocated for an entire build cycle. These factors did not appear to have any serious side effects, although it is possible that the two remaining initial features, standards of learning and teacher templates, might have been implemented if two week build cycles had been possible.

7.4- Having a Customer Proxy

In this project the customer was represented by a proxy who worked with the developer to communicate the project requirements. It is very likely that customer proxies would also be used frequently in real-world applications of Cowboy. Customers are very likely to be too busy to spend too much time in development meetings. It is also possible that there will be many end users and stakeholders, so a single customer or proxy may represent many interests.

Ms. Helms worked out very well as a customer proxy. However, it would be best if the actual customer or stakeholder could have performed the role she assumed for this project. Many decisions had to made in development meetings that had to be approved

after the fact by Dr. Taylor. Even more serious, though, were omissions that were not discovered until the end of the project because Ms. Helms was entirely unaware of Dr. Taylor's desires and expectations. A solution may have been to periodically schedule meetings with the developer, proxy and customer all attending. The customer could walk through the software with a testing script just as the proxy did. This would also be an opportunity for the proxy to verify that her impressions of the customer's requirements were on track.

8- Cowboy 3.0 - How to Improve the Process

Cowboy was a good idea. While teams develop most software, solo development does occur and should also follow an appropriate methodology. Based on my experience in applying this methodology in the VLP trial, I would propose several specific improvements.

8.1- Problems in the Application of Cowboy to the VLP Project

As outlined previously, the development of the VLP prototype was successful. Not unexpectedly, the Cowboy methodology did have some limitations. Some of these limitations are restricted to this particular project, while others suggest areas of improvement in the methodology itself.

8.1.1- Customer / Developer Relationship

One of the more significant issues with the VLP project was the lack of meetings with the final customer. Although the proxy concept worked out well, the input of the actual customer at every second or third meeting could have significantly altered some of the

decisions made by the developer and proxy in her absence. In general this is a good practice that should be adopted in later adaptations of Cowboy. The input of the end customer can give significant insight and help keep the project on track.

8.1.2- Time Commitments

Determining set weekly time commitments or even a daily schedule of work for the developer to follow would have increased productivity and might have kept iteration lengths consistent. During this process, the developer worked on VLP when time was available, and in retrospect, this was a poor plan. The shortcomings of the code, not having class-level comments or automated testing features, and missing features could be attributed to the lack of a disciplined schedule that allowed time for these activities to occur.

Cowboy could possibly benefit from the adoption of time management or time commitments as part of the agreement between the developer and the customer. While it is possible that the issues that arose during this experiment may not always occur, cowboys work without direct social pressure and may very well benefit from having structured work schedules.

8.1.3- Test Driven Development

The VLP experiment did not incorporate Test Driven Development (TDD), a core part of the Agile Manifesto. This element is core to any methodology claiming to be agile and was left out of this particular project due to poor scheduling and the inexperience of the developer with TDD and unit testing in general.

In future Cowboy trials, the developer should be knowledgeable about and schedule adequate time for unit testing. TDD requires unit tests to be written at the class level before the classes themselves are written, and Cowboy should take advantage of this. Unit tests force a programmer to clearly understand the specific issues a class is being written to solve and ensure that new code does not introduce new errors.

8.2- Additional Standards for Professional Adaptation

Research did not uncover any evidence of how programmers currently working alone stay organized or if following a methodology is popular for these developers, but if Cowboy is applied in a professional setting some additions would be desirable.

8.2.1- Code Repository

Code repositories are an invaluable tool of the professional software developer. In team development settings, they are indispensable in facilitating parallel development of software components by many developers working simultaneously. They also offer advantages in maintaining version control and periodically making backups of the code base, useful to the solo developer as well as team developers. Developers should frequently check code into the repository, usually after implementing a feature. Repositories allow a developer to "roll back" the code to a certain point if a large bug or issue is discovered. Since checking in a version usually requires comments to be associated with it, the developer is not burdened with having to remember which version has which working feature. Artifacts such as goal and task lists can also be checked into

the repository, allowing the developer to recover historical copies of documents without manually handling version control.

8.2.2- Contracts

Professional software developers, by definition, write software for money. This inevitably requires a contract between developer and customer before work begins. Cowboy, like all agile processes, doesn't pretend to be able to set all requirements before work begins. Customers, however, tend to be uncomfortable signing a contract that doesn't fully specify the work to be performed as well as the price and delivery date.

One solution is to have an iterative contract arrangement. This concept requires the customer to sign off on a small, core set of features that may be developed quickly. This initial iteration should be no longer than six to eight weeks. At the end of the iteration, the customer has two choices: to continue, signing a contract for the next iteration, or choose to take what has been done and walk away. This process has advantages for both developer and customer. The developer can more accurately estimate the time and skills required for future iterations and never has too many features in development at the same time. The customer's risk is significantly reduced and the customer is very aware of the progress of the software. [Subramaniam]

9- Conclusion

Cowboy programmers write a lot of code in many applications. They come in all shapes and sizes and have different levels of success. Despite the ubiquity of the solo developer,

little software engineering research has focused on practices appropriate for the solo developer.

Before I developed and tried the Cowboy methodology I followed no formal methodology at all. I would elicit requirements from customers at the beginning of a project and then proceed to design and development. Input from the customer after the initial requirements elicitation would be extremely limited. No concept of iterative development was considered and artifacts like goal and task lists were never created or maintained. In all, my previous efforts of creating software resulted in working products that were satisfactory upon completion. However, the process used to create these products was much more hectic than the application of Cowboy to the VLP project. Production systems were often built directly on top of prototypes. This practice compromised the understandability and maintainability of old projects because the code was not always easy to understand.

The implementation of the VLP prototype, utilizing the Cowboy methodology was much more organized and comfortable. Taking advantage of frequent customer meetings and reviews provided constant feedback and adjustments to the project right after each section of the code was written. Simple, well-maintained artifacts kept system requirements orderly and easy for both customer and developer to understand. Despite the missing features in the final product, what was developed was considered very successful. Constant refactoring have made the code base very easy to read and maintain. While there is room for improvement, this first experiment with Cowboy was largely successful.

Its success implies that adopting agile practices can increase the quality of the software and overall success of the projects developed by cowboys. .

Appendix A: Sources Used

[37signals] 37signals. 2006. *Getting Real*, Chicago, IL: 37signals.

[Ambler, 02] Ambler, Scott. 2002. *Agile Modeling*, New York: John Wiley and Sons, Inc.

[Ambler, 06] Ambler, Scott. 2006. "Agile Unified Process (AUP) Home Page," http://www.ambysoft.com/unifiedprocess/agileUP.html, accessed 31 Jul 2006, Ambysoft, Inc.

[Bach, 95] Bach, James. Mar 1995. "Enough about process: what we need are heroes," IEEE Software, Volume 12, Issue 2, pp96-98.

[Bach, 99] Bach, James. Dec 1999. "What Software Reality is Really About" Computer, Volume 32, Issue 12, pp148-149.

[Beck, 99] Beck, Kent. 1999. *Extreme Programming Explained: Embrace Change.* Boston, MA, Addison-Wesley.

[Beck, 01] Beck, Kent. et all. 2001. Manifesto for Agile Software Development. http://agilemanifesto.org, accessed 28 Jul 2006.

[Curtis] Curtis, Tex. Mar/Apr 2001. "So You Wanna Be a Cowboy?" IEEE Software, Volume 18, Issue 2, pp. 112, 110-111.

[Hodges] Hodges, A. 1983. *Alan Turing: the Enigma.* London: Burnett; New York: Simon & Schuster.

[Mateosian] Mateosian, Richard. Feb 1996. "The road ahead." *IEEE Micro*, vol. 16, no. 1, pp. 5-6,72.

[McBeen] McBreen, Pete. 2003. *Questioning Extreme Programming.* Boston, MA: Addison-Wesley.

[McConnell] McConnell, Steven C. Nov/Dec 2001. "Raising Your Software Consciousness." IEEE Software, Volume 18, Issue 6, p7-9.

[Schwaber, 96] Schwaber, Ken. 1996. "Controlled Chaos: Living on the Edge," http://www.controlchaos.com, accessed 31 Jul 2006, Advanced Development Methods, Inc.

[Schwaber, 06] Schwaber, Ken. 2006. "Scrum: it's about common sense," http://www.controlchaos.com, accessed 31 Jul 2006, Advanced Development Methods, Inc.

[Smialek] Smialek, Michal. 2005. "From User Stories to Code in One Day?" *Extreme Programming and Agile Processes in Software Engineering*, Proceedings of the 6th International XP Conference, Sheffield, UK, 18-23 Jun 2005, p38-47, Heidelberg, Germany: Springer.

[Subramaniam] Subramaniam, Venkat. Andy Hunt. 2006. Practices of an Agile Developer, Pragmatic Bookshelf, Raleigh, NC.

[Wells] Wells, Don. 2006. "Extreme Programming: A Gentile Introduction," http://www.extremeprogramming.org, accessed 31 Jul 06.

Appendix B: Artifacts

B-1: Representative Goal List

- Provide students an environment in which to build VLPs – a collection of organized multi-media resources.
- Allow student to link resources to other resources and/or external web documents.
- Enable teachers to review student VLPs.
- Provide teachers with the means to assess VLPs.
- Make VLP accessible to a wide range of teacher/students by maximizing (and focusing on) ease of use.
- Allow students to associate tags with resources.
- The system should be cross-platform and web-based.
- There should be a way to align SOL requirements with tags.
- Allow teachers to observe/identify themes in a VLP.
- Allow teachers to create templates and place them into VLPs for a given course.

B-2: Representative Student Task List

Using VLP, STUDENT will be able to...

manage resources

- 1 by adding resources
- 1 uploading files
- 1 adding/editing title
- 1 comments
- 3 icons
- 3 review archived VLP

linking

- 1 add/edit links from resource comment to other resource
- 1 click on icon or thumbnail and open resource file or URL in browser
- 3 link to archived VLP resource

tagging

- 2 add/remove tag to resource
- 2 see a list of their own tags
- 3 view a tag pool of all their tags
- 3 cluster their own tags

assessment

- 2 view comments by teacher for assignments
- 3 view assessment rubrics

clouds

- 3 have clouds

B-3: Representative Teacher Task List

Using VLP, TEACHER will be able to...

review student portfolios

- 2 review all student resources for each tag via couds, individual tags, groups of tags, etc.
- 2 comment on individual resources
- 3 review archived VLP

tagging

- 3 assign global tags to all VLPs associated with a course

assessment

- 2 create rubrics and associate with a tag for the assignment
- 2 store and reuse rubrics
- 2 assess student performance on assignment using rubric

clouds

- 3 implement clouds

assignments

- 3 create templates
- 3 create rubrics
- bundle and deploy templates and rubrics as assignment to students in a course

B-4: Representative Glossary

Assignment – consists of a template, grading rubric and one assignment tag.

Assignment tag – a tag associated with a resource that corresponds to a particular assignment. If placed there by the teacher on a template resource, neither the resource nor tag may be removed by the student.

Cluster- a group of tags that all have the same "parent" – tags may be in 0..* clusters e.g.- the color cluster may include red, blue and green

External resource- a resource stored outside the VLP server accessible via URL

Internal resource- a resource stored on the VLP server

Resource- an object with below properties associated with a single file of any format stored on a server and having the following properties date entered, title, comments, tags, icon (maybe thumbnail), links.

Tag- a single word that is associated with a resource and may be used in a search or to group similar resources

Template – a series of resources created by a teacher w/ partial information (questions/tasks for students in text areas) that are pre-tagged with an assignment tag. They are designed to be the jumping off point for a student for a given assignment.

Appendix C: Example Proxy Customer Questionnaire

Please answer the following questions honestly and openly. Any negative responses are not a reflection on you the customer, but indicate to the developer areas that need work in this development process. Thank you for your time!

If questions ask you to rate and you rank less than 5, please explain why...

1) Meeting purpose (as you understood it):

---

2) Did you feel that the purpose of the meeting was achieved? 1 2 3 4 5

3a) Did you feel lost or confused at any point? Y / N

3b) If yes, did the developer pick up on your confusion and clarify? Y / N

3c) What feedback do you have that experience and if not satisfied, please explain.

---

4) In this particular meeting: to what extent do you feel your input affected the meeting's course, development decisions made, and overall design of the program? 1 2 3 4 5

5a) Were there tradeoffs (differences in what was wanted by you and what the developer would commit to deliver) made in this meeting? Y / N

5b) If yes, please rank how well explained the tradeoffs were. 1 2 3 4 5

5c) Now that the meeting is over, please rank how happy you are with the tradeoffs made. 1 2 3 4 5

---

6) Is there anything from today's meeting that you would have preferred to be organized or handled differently? Y / N (if yes, please explain)

---

7) Overall, how did this meeting make you feel about the project?

Appendix D: Example Meeting Agenda

Meeting: 8 Jun 2006
Who: Hollar and Helms
Objective: Initial Design Meeting
Agenda:
Define high-level goals
Use verbs like "allow, provide, enable"

Define initial requirements the form of a core task list
Identify needs the system will satisfy
Tasks are grouped into categories
Tasks are the actions end users will perform
each can be prefaced with: "You should be able to…"

Develop user stories for core tasks
Keep to simple Subject-Verb-Object sentences to describe events.
Maintain notation description (common vocabulary) in a separate list.
(in other words, make user stories simple and define potentially confusing words in a "data-dictionary" of sorts)

Attempt to prioritize tasks into 3 categories
This list will persist and allow the developer know where to focus for the next iteration – Core/Important/Nice to have

Sketch initial user interface ideas on whiteboard in conjunction with 1-3
Create very primitive model that allows developer to start coding
Use list from 4 to determine what features to account for in initial design…

Appendix E: Example Proxy Customer Test Script

VLP Testing Script
There are two known major errors in the current build of the system. As a result, some functionality will not be tested.
Cannot create new users
Editing resources and links not functional – messes up the internal/external resources

Today you will be starting a VLP that tracks interesting things about Brooks's back yard. Many fun things happen in Brooks's back yard, and there is much media that we want to make available via VLP.

1. Log in:
    Username: brooks
    Password: p0ck3td0g

2. Add internal resources and tag them:
    You will be adding 7 resources to the VLP. For each one, you will be given the title, text, and filename to upload. (all media in My Pictures/VLP folder) Once the resource is created, you may add tags.

    1:
    Title: Smoking Pork
    Text: These are a couple of pork loins about ready to come off the grill and be eaten.
    Filename: pork_loin.jpg
    Tags: Food, Summer, Fire

    2:
    Title: Pocket running in circles
    Text: This is pocket running around in circles. This move cracks me up!
    Filename: pocket_movie.avi
    Tags: Pocket, Summer

3:
Title: Moonflower on mailbox
Text: This is the moonflower I gave to my neighbor. It never got that big because it was left in a pot.
Filename: moon_flower.jpg
Tags: Plants, Summer

4:
Title: Baby watermellon
Text: This is one of the watermelons that grew in the garden last year. It ended up getting fullsize and being quite tasty.
Filename: watermelon.jpg
Tags: Plants, Food, Summer

5:
Title: Summertime view from shed
Text: This is the whole yard from the back corner where the shed is. The garden hasn't become completely overgrown yet.
Filename: yard_summer.jpg
Tags: Yard, Summer

6:
Title: Snow covered fire pit
Text: This is the strange snow fall that we got in February. It was enough to knock over the apple tree.
Filename: fire_pit_snow.jpg
Tags: Fire, Winter, Snow

7:
Title: Winter view from shed
Text: This is the whole yard from the back corner during the snow fall.
Filename: yard_snow.jpg
Tags: Yard, Winter, Snow

3. Link the resources to each other and to outside web pages. Link each of the following resources to the indicated resources, or to the indicated web pages. You are provided with a title to use in the link, but must think of a title for the link.

INTERNAL LINKS:
Moonflower on mailbox
Link to "Baby watermelon" – more things that grow

Summertime view from shed
Link to "Winter view from shed" – Same picture, different season
Link to "Baby watermelon" – You can see the garden

Winter view from shed
Link to "Snow covered fire pit" – You can see the fire pit

EXTERNAL LINKS:
Smoking Pork
Link to "http://www.smoking-meat.com"
Link to "http://www.bbqsauceofthemonth.com"

Pocket running in circles
Link to "http://www.boxer-dog.org/"
Link to "http://www.boxerworld.com/

Baby watermelon
Link to "http://www.watermelon.org"

4. All done! Now you can click on tags to see all the items tagged with that tag, follow the links and open resources in new windows!

Appendix F: Final Customer Test Script

VLP Testing Walkthru for Dr. Pam Taylor.

Please remember- any issues you come across, please note them. Most issues fall into the following categories:

1) UI - User interface problems - colors, placement size of text, images or links, etc.
2) Behavior - Usually unexpected behavior (you click and something "odd" happens) or missing behavior (this isn't a way to do "this thing").
3) Bugs - Things that are actually broken. Hitting the "back button" will usually allow you to back up and skip the steps that are broken.

This test will ask you to perform tasks inside the VLP system. The instructions will outline small goals, then step you through completing those goals. You will create a teacher and a student account, set up coursework as a teacher, complete the assignments as a student, grade them as a teacher, then view the grading as a student.

Notes to you, Dr. Taylor, will be wrapped in {braces}. They are not steps to be followed, just information for you.

Also, do not feel bad for not understanding something or getting confused! ;) This is a weakness in the software or this document, not you!

Step 1: Create the teacher account

a) Open Firefox and go to: http://67.62.202.249:3000/ (does not currently work with Safari – I discovered a bug tonight…)
b) Click "Register for an account"
c) Create a teacher by filling in the fields as follows
    a. First: Pam
    b. Last: Taylor
    c. Login ID: pam
    d. Email: ptaylor@vcu.edu {not used for anything yet}
    e. Grade: 1
    f. Type: Teacher
    g. Password: password
    h. Password conf: password
d) Click "Submit"

Step 2: Create the student account

a) Click "Register for an account"
b) Create a teacher by filling in the fields as follows
    a. First: Student
    b. Last: One
    c. Login ID: student_one
    d. Email: student@vcu.edu {not used for anything yet}
    e. Grade: 1
    f. Type: Student
    g. Password: password
    h. Password conf: password
c) Click "Submit"

Step 3: Login as the teacher, and setup course, and students

{The "home screen" is pretty much blank right now, please ignore it…}

a) Click "Manage Courses" – where you add and edit courses and enrollment
b) Add a course named "Dogs" to Grade "1"
c) Once created, click on "edit enrollment"
d) Add Student One to the course by clicking on the name

Step 4: Create clusters and tags for a course

{Many tags are associated with a cluster. Clusters are associated with courses so that the students enrolled will see that they need to use them. One or two clusters would most likely constitute an assignment.}

a) Create a cluster named "Cluster One" for course "Dogs"
b) Once created, click "edit tags"
c) Add the following tags: "boxer greyhound agility"

Step 5: Create rubric to assess the Dogs Cluster

a) Click on "Rubrics"
b) Create a rubric named "Rubric one" for course "Dogs"
c) Click "edit line items" once the rubric is created
d) Add "Has at least 3 resources" for 5 points
e) Add "Found dogs that were cute" for 10 points
f) Add "Intelligent comments used" for 10 points

Step 6: Log the teacher out

a) Click on "logout" – upper right-hand corner

Step 7: Log in as the student

a) Login as "student_one" / "password"
b) Notice the list of unused tags on the left hand side

Step 8: Create a few resources for the assigned tags

{Internal resources are files that are uploaded, and External resources are web-pages. Once resources are created, they may be tagged and linked to other resources.}

a) Click on "Add resource"
b) Add first resource
    a. Title: Boxers
    b. Text: Boxers are great dogs.
    c. Click: External Resource
    d. URL: http://www.boxerworld.com
c) Once created, click "Edit tags" and add "boxer" {notice it drop off the list at left}

d) Add another resource
    a. Title: Senator Boxer
    b. Text: This is a senator from CA.
    c. Click: External Resource
    d. URL: http://boxer.senate.gov

e) Once created, add the tag "boxer"
f) Link this resource to the first boxer one
    a. Click on "add link"
    b. Title: A link
    c. Text: This is a link from a dog to a senator.
    d. Click: Internal Link
    e. Select: Boxers

g) Add another resource
    a. Title: Agile dogs
    b. Text: These dogs can do crazy tricks
    c. Click: External Resource
    d. URL: http://www.agiledogs.net
h) Once created, add the tag "agile"

i) Add another resource {just to demonstrate uploading}
    a. Title: Not a greyhound
    b. Text: This is a picture
    c. Click: Internal Resource
    d. File: Select any image file stored on you computer
j) Once created, add the tag "greyhound"

Step 9: Look over student's portfolio

a) Click on "Cloud"
b) Notice how the font for boxer is larger – that's because it has been used more than the others.
c) You can click on a tag and see all the resources tagged with that.

Step 10: Logout as student

a) Click on "Logout"

Step 11: Log back in as the teacher

a) "pam" / "password"

Step 12: Review the Student work

a) Click on "Review students"
b) Click on the student's name.
c) Click on the name of the first resource.
d) At the bottom of the resource, click on "add comment"

e) Type: "This resource needs more content" and click "Create"
f) Click on "Review Students" again
g) Click on "Cluster One"
h) The resources to be graded are listed. Click "Grade with this rubric"
i) Fill the rubric in with comments and points for each line item.

Step 13: Logout the teacher and login the student

a) Logout teacher
b) Login student "student_one" / "password"

Step 14: Look over comments and rubrics

a) Click on "View" under Unseen comments
b) The resource is opened and the new comment marked.
c) Click on "Rubrics"
d) Click on "Rubric One"
e) See that the completed rubric is there.

So endeth the testing....

Appendix G: RDoc Generated Documentation Example for Student Class

Class **Student**
In: app/models/student.rb
Parent: User

Methods

in_course? rubric_templates_for_course rubrics_for_course unseen_comments unused_tags

Public Instance methods

in_course?(*course*)

[Source]

rubric_templates_for_course(*course*)

[Source]

rubrics_for_course(*course*)

[Source]

unseen_comments()

[Source]

unused_tags()

[Source]

Appendix H: Source Code for Student Class

```
class Student < User
  has_and_belongs_to_many :courses
  has_many :student_rubrics
  has_many :resources, :foreign_key=>'user_id'

  def in_course?(course)
    a = course.students
    a.include?(self)
  end

  def unseen_comments
    a = Array.new
    resources = Resource.find(:all, :conditions=>['user_id = ?',
self.id])
    resources.each {|x|
      x.comments.each {|y|
        if  y.viewed_by_owner_at == nil { a << y }
      }
    }
    return a
  end

  def unused_tags
    tags = Tag.tags_for_user(:user_id => self.id).collect{|x| x.name}
    a = Array.new
    courses.each {|course|
      course.clusters.each {|cluster|
        cluster.tags.each {|tag|
          a << tag.name
        }
      }
    }
    a = a - tags
    a.sort.uniq
  end

  def rubrics_for_course(course)
    a = Array.new
    student_rubrics.each {|x|
      a << x if x.rubric_template.course == course
    }
    a.uniq
  end

  def rubric_templates_for_course(course)
    a = Array.new
    student_rubrics.each {|x|
      a << x.rubric_template if x.rubric_template.course == course
    }
    a.uniq
  end
end
```

Appendix I: Completed Proxy Questionnaires

Customer meeting survey Meeting Date: 10/15/02

Please answer the following questions honestly and openly. Any negative responses are not a reflection on you the customer, but indicate to the developer areas that need work in this development process. Thank you for your time!

If questions ask you to rate and you rank less than 5, please explain why...

1) Meeting purpose (as you understood it):

Part 1	Part 2
~~Design~~ set definitions + goals	review goals/definitions discuss UI

2) Did you feel that the purpose of the meeting was achieved? 1 2 3 4 (5)

3a) Did you feel lost or confused at any point? (Y) N
3b) If yes, did the developer pick up on your confusion and clarify? (Y) N
3c) What feedback do you have that experience and if not satisfied, please explain.

this process was great I feel it was much more productive than previous meetings :)

Actually this is a little weird cause most of the time I just told you if I didn't understand but that's mostly just my personality

4) In this particular meeting: to what extent do you feel your input affected the meeting's course, development decisions made, and overall design of the program? 1 2 3 (4-5)

Much of what you had in mind as far as the UI went worked for me — I feel I affected the verbage of the task list and goals you prioritized without much input from me but I think that was okay → I would have said something if I disagreed.

Customer meeting survey Meeting Date: 6/15/06

5a) Were ~~their~~ there tradeoffs (differences in what was wanted by you and what the developer would commit to deliver) made in this meeting? Y (N)
5b) If yes, please rank how well explained the tradeoffs were. 1 2 3 4 5

Mostly we were just wondering if Pam's priorities & goals were the same as before... I don't feel like any compromising has happened yet.

5c) Now that the meeting is over, please rank how happy you are with the tradeoffs made. 1 2 3 4 5

6) Is there anything from today's meeting that you would have preferred to be organized or handled differently? Y / (N) (if yes, please explain)

7) Overall, how did this meeting make you feel about the project?

I'm pleased with the fact that you seem to know now how to ask questions that will get you the information that you need. The process has been smooth, interesting and painless (a great improvement over the first few meetings we had with Pam). I'm happy to be working with you!

Customer meeting survey Meeting Date: 7/4/06

Please answer the following questions honestly and openly. Any negative responses are not a reflection on you the customer, but indicate to the developer areas that need work in this development process. Thank you for your time!

You should put the scale on here somewhere 1=☹ 5=☺ etc

If questions ask you to rate and you rank less than 5, please explain why..

1) Meeting purpose (as you understood it):

Review first build - review goals + reprioritize

2) Did you feel that the purpose of the meeting was achieved? 1 2 3 4 (5)

3a) Did you feel lost or confused at any point? (Y) / N

3b) If yes, did the developer ~~pick up~~ on your confusion and clarify? Y / N

3c) What feedback do you have that experience and if not satisfied, please explain.

You said to use your pics

Are there some words missing here?

When you put the computer in front of me & said go I didn't know what to do because I didn't have my files — but then I realized I just needed to make stuff up and that you wanted me to use your images (plus I was a little distracted by the TV) ☺

4) In this particular meeting: to what extent do you feel your input affected the meeting's course, development decisions made, and overall design of the program? 1 2 (3) 4 5

I helped find that bug about the pic not being associated...

I felt like you wanted color advice but your choices were fine

Customer meeting survey Meeting Date: 7/4/6

5a) Were ~~their~~ there tradeoffs (differences in what was wanted by you and what the developer would commit to deliver) made in this meeting? Y / (N)
5b) If yes, please rank how well explained the tradeoffs were. 1 2 3 4 5

5c) Now that the meeting is over, please rank how happy you are with the tradeoffs made. 1 2 3 4 5

6) Is there anything from today's meeting that you would have preferred to be organized or handled differently? Y / (N) (if yes, please explain)

7) Overall, how did this meeting make you feel about the project?

good. still excited. looking forward to getting to use it for my own stuff.

Customer meeting survey Meeting Date: 23Jul6

Please answer the following questions honestly and openly. Any negative responses are not a reflection on you the customer, but indicate to the developer areas that need work in this development process. Thank you for your time!

For questions with ranking, 5 is best and 1 is unsatisfactory. If questions ask you to rate and you rank less than 5, please explain why...

1) Meeting purpose (as you understood it):

try out the newest build

2) Did you feel that the purpose of the meeting was achieved? 1 2 3 4 (5)

Yes

3a) Did you feel lost or confused at any point? Y / (N)
3b) If yes, did the developer pick up on your confusion and clarify? Y / N
3c) What feedback do you have about that experience and if not satisfied, please explain.

4) In this particular meeting: to what extent do you feel your input affected the meeting's course, development decisions made, and overall design of the program? 1 2 (3) 4 5

I asked a lot of questions & commented on alignment problems + font issues but that's sort of small beans.

Customer meeting survey Meeting Date: 23Jo16

5a) Were their tradeoffs (differences in what was wanted by you and what the developer would commit to deliver) made in this meeting? Y /(N)
5b) If yes, please rank how well explained the tradeoffs were. 1 2 3 4 5

5c) Now that the meeting is over, please rank how happy you are with the tradeoffs made. 1 2 3 4 5

6) Is there anything from today's meeting that you would have preferred to be organized or handled differently? Y /(N) (if yes, please explain)

7) Overall, how did this meeting make you feel about the project?

still great!

Customer meeting survey Meeting Date: 9/11/00

Please answer the following questions honestly and openly. Any negative responses are not a reflection on you the customer, but indicate to the developer areas that need work in this development process. Thank you for your time!

If questions ask you to rate and you rank less than 5, please explain why...

1) Meeting purpose (as you understood it):

follow script for newest build

2) Did you feel that the purpose of the meeting was achieved? 1 2 3 4 (5)

3a) Did you feel lost or confused at any point? (Y) / N
3b) If yes, did the developer pick up on your confusion and clarify? (Y) / N
3c) What feedback do you have that experience and if not satisfied, please explain.

There were just minor things that left me confused but once clarified it wouldn't happen again (edit tag vs edit clustered confusion)

4) In this particular meeting: to what extent do you feel your input affected the meeting's course, development decisions made, and overall design of the program? 1 2 3 (4) 5

I can see that watching me use the program is helpful. Following a "do this then do that" script is good for me because I can see how a teacher using it for the first time would interact.

Customer meeting survey Meeting Date: 9/11/06

5a) Were their tradeoffs (differences in what was wanted by you and what the developer would commit to deliver) made in this meeting? Y / (N)
5b) If yes, please rank how well explained the tradeoffs were. 1 2 3 4 5

5c) Now that the meeting is over, please rank how happy you are with the tradeoffs made. 1 2 3 4 5

6) Is there anything from today's meeting that you would have preferred to be organized or handled differently? Y / (N) (if yes, please explain)

7) Overall, how did this meeting make you feel about the project?

Still good. It's going well.

What if there were 2 teachers team teaching a course? Could they both have access to the course info?

Appendix J: Vita

Vita

Ashby Brooks Hollar was born on the 23rd of May, 1977 in Richmond, VA and is an American citizen. Brooks graduated from Robert E. Lee High School, Staunton, VA in 1995. He received a Bachelor of Science in the field of Computer Science in 1999 from James Madison University in Harrisonburg, VA. From 1999 through 2001 he was adjunct faculty in the Computer Science Department at James Madison University. The completion of this thesis meets the final requirement for his Master of Science in Computer Science from Virginia Commonwealth University, Richmond, VA, 2006.