International Journal of Business and Management Invention
ISSN (Online): 2319 – 8028, ISSN (Print): 2319 – 801X
www.ijbmi.org Volume 2 Issue 7‖ July. 2013‖ PP.89-95

Extreme Programming beyond Adoption: A Longitudinal Case Study of a Software Start-up

Akbar Saeed[1], Peter Tingling[2]
[1]*Assistant Professor, School of Business and Economics, Wilfrid Laurier University, Canada*
[2]*Associate Professor, Beedie School of Business, Simon Fraser University, Canada*

ABSTRACT: *Rapid Application Development (RAD) has captured interest as a solution to problems associated with traditional systems development. Describing the adoption of agile methods and Extreme Programming (XP) by a software start-up, we find that all XP principles were not adopted equally and were subject to temporal conditions. Small releases, on site customer, continuous integration and refactoring were most vigorously advanced by management and adopted by developers. Paired programming on the other hand was culturally avoided.*

KEYWORDS: *Agile Methods, Extreme Programming, Rapid Application Development*

I. INTRODUCTION

The speed and quality with which systems are delivered continues to concern both practitioners and academics. Traditional methodologies, while praised for their rigor, are often criticized as non responsive, bloated, bureaucratic, or contributing to late and over budget systems that when delivered solve problems that are no longer relevant.

Various solutions have been proposed. Frequently combined under the rubric of Rapid Application Development (RAD), these include extensive user involvement, Joint Application Design, prototyping, integrated CASE tools, and more recently, agile methods such as Extreme Programming (XP). Despite the popularity of agile methods, there is a noted paucity of research that goes beyond the adoption stage [1]. Following a qualitative longitudinal case study of how agile methods were implemented and employed in a start-up software organization, we conclude that adoption and extent of agile principle appropriation are affected temporally and by institutional culture. Coding standards for example were initially excluded in a search for creativity and flexibility. Similarly, in addition to the continuous improvement of refactoring, bursts of intense focus also seemed to occur.

II. RAD AND AGILE METHODS

The need for software quality and reliability is a mainstay of application development [2, 3]. While recognizing there is no "silver bullet" solution [4-6], the Systems Development Life Cycle is a well adopted 'systematic, disciplined, quantifiable approach to the development, operation and maintenance of software' [2, 6]. However, with increasing backlogs; some high profile development failures; and the need to adapt to emerging business conditions; the SDLC has been subject to criticism that it is constraining, heavyweight and results in projects that are outdated before they are finished [8]. Consequently, many organizations have adopted alternates that emphasize incremental development with constant customer feedback (Rapid Application Development); structured processes where constituents collectively and intensely review requirements (Joint Application Development); construct partial systems to demonstrate operation, gain acceptance or technical feasibility (Prototyping); and tools that assist in software development and business analysis (Computer Aided Systems Engineering).

In 2001, a group of programmers created a landmark manifesto that embodied the core principles of a new methodology [10]. An extreme application of RAD, agile methods capitalize on member skill; favor individuals and interactions over process and tools; working software over comprehensive documentation; customer collaboration over negotiation; and change rather than plans and requirements. Dynamic, context specific, aggressive and growth oriented [11, 12], agile methods favor time boxing and iterative development over long or formal development cycles. The most widely adopted agile development methodology, eXtreme Programming is a generative set of guidelines that consisting of twelve inter-related principles. These are described in TABLE 1

.

TABLE 1. **Agile Principles of Extreme Programming**

XP Principle	Rationale and Description
40-Hour Work Week	Alert programmers are less likely to make mistakes. XP teams do not work excessive hours.
Coding Standards	Co-operation requires clear communication. Code conforms to standards.
Collective Ownership	Decisions about the code are made by those actively working on the modules. All code is owned by all developers
Continuous Integration	Frequent integration reduces the probability of problems. Software is built and integrated several times per day.
Continuous Testing	Test scripts are written before the code and used for validation. Ongoing customer acceptance ensures features are provided.
On-Site Customer	Rapid decisions on requirements, priorities and questions reduce expensive communication. A dedicated and empowered individual steers the project.
Pair Programming	Two programmers using a single computer write higher quality code than individual programmers.
Planning Game	Business feature value is determined by programming cost. The customer decides what needs is done or deferred.
Refactoring	The software is continually improved
Simple Design	Programs are simple and meet current rather than future evolving requirements.
Small Releases	Systems are updated frequently and migrated on a short cycle.
System Metaphor	Communication is simplified and development guided by a common system of names and description.

Source: Adapted from [8]

III. METHODOLOGY AND DATA COLLECTION

For this study, we used a case oriented approach which is an "empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident" [13].

Site selection was opportunistic, the result of an ongoing relationship with Semper Corporation, a young start-up firm developing an interactive software product. Data was collected over a sixteen month period and consisted of interviews with employees, observation of the environment and work practices, and retrospective examination of documents and email [14] These are generally described in TABLE 2.

TABLE 2. **Data Collection Activities**

Data Type	Description
Interviews	The development staff and company principals were regularly interviewed throughout the year long data gathering.
Observation	Programming and development staff was observed at least weekly. This was both active (at the development offices) and passive (by remote viewing of video cameras).
Artifact examination	Employment and programming records, progress and bug reports, and copies of each build and version of the product were reviewed as was email correspondence.

The main steps in analysis involved identification of concepts and definition,followed by the theorizing and write up of ideas and relationships. Content relating to agile methods and extreme programming content were separated and categorized according to discrete principles. Illustrative yet concise examples were then selected. Direct quotations have been italicized and placed within quotation marks.

IV. EXTREME PROGRAMMING AT SEMPER CORPORATION

This section reviews principles described in Table 1.Although these principles were meant to be generative rather than all-inclusive, typical recommendations recognize their inter-relatedness and suggest that implementation be done in entirety with adaptation encouraged only when use and familiarity is established [7]. Findings are summarized in TABLE 3.

40 Hour Work Week.

Company policy was one of flexible work hours. Other than core hours between 10:00 to 15:00, developers were free to set their own schedule. While there were work weeks longer than 40 hours (during customer testing or resolving production problems) this was the exception rather than the norm. There was no overtime compensation. Another factor affecting the schedule was the young (average 21) age of the developers who adopted nocturnal habits because of their social schedule. For example, advising when he might be in the office, one of the developers noted "*I will be in early tomorrow -around 10:00-10:30".* Email conversations (where a message was sent and a response received) between the developers and managers declined during the core hours from 45% to 31% and increased from 6% to 37% between 22:00 and 04:00.

Coding Standards

Coding standards were initially avoided. For example, rather than conventionally declaring all variables at the beginning of a module, one programmer simply added them wherever they were needed. Requests to impose standards were generally ignored by management until the program became sufficiently complex as to require tighter control and the CEO realized development teams continually rewrote the variables when they refactored or changed modules. Staff attrition later resulted in de facto standards.

Collective Code Ownership

With the exception of a few core modules, module decisions were made by the active developers. As a consequence, different ideas about modules were continually rewritten according to individual preferences. Although modules had multiple authors – one team tended to write the analytic modules while another wrote the graphically intense reporting component. While officially shared and located on a common storage medium, developers were reluctant to adapt code written by others and continued to speak of "their code".

Continuous Integration

The programming environment was Visual Basic in a Microsoft .Net framework. Modules were tested in isolation and embedded into the program several times a day adding to the formal build schedule with weekly integration. During the sixteen months of observation, more than 35 complete formal versions of the product and 225 integrations were compiled. In addition, internal and external users were given replacement Dynamical Link Libraries (DLLs) that encouraged up-to-date testing. Despite a preference for working code, there were several occasions when changes to the data model required extensive rewrites and the code was broken for up to two weeks as modules were re-written and tested.

TABLE 3. **Adoption Faithfulness of XP Principles**

eXtreme Programming Principle	**Adoption Level***	**Temporal Effects**	**Summary**
40-Hour Work Week	Full	N	Developers worked flexible but regular workdays.
Coding Standards	Low to Partial	Y	Standards were initially avoided but later implemented.
Collective Code Ownership	Partial	Y	Code was officially shared but developers exhibited possessiveness.
Continuous Integration	Full	N	Code was rarely broken and was continually linked and compiled.
Continuous Testing	Partial to Full	Y	Testing was continuous but advance scripts were not created. Black box testing was phased.
On-Site Customer	Full	N	The CEO and Analytic Director acted as customers.
Pair Programming	Low	N	Programmers were independent except when difficulties or interdependencies existed.
Planning Game	Full	N	Value engineering balanced features against time and budget.
Refactoring	Full	Y	Modules were constantly improved. Periodic bursts of and dramatic improvement occurred.
Simple Design	Full	N	Working software was favored.
Small Releases	Full	N	Frequent (weekly) build cycles
System Metaphor	Full	N	Communication was simple and informal but unambiguous.

*Adoption is considered Low, Partial or Full

Continuous Testing

Test scripts were not written in advance of coding (as recommended by XP) and were frequently developed in parallel. Ongoing functional and compatibility testing used standardized and ad hoc test scripts. Because the design was modular and addressed a specific rather than generic problem, the majority of the code could be tested in isolation. Integration testing was completed after each weekly build and was conducted by management and external users. Black box testing was conducted using a combination of end user and test samples. HCI and usability aspects were the most dynamic with the majority of the changes immediately accepted or rejected by the onsite customer. The few exceptions to this occurred when the developers were given free rein to creatively design new ideas or when previously adopted choices were abandoned. The CEO often challenged the developers to present complex information simply and intuitively rather than providing them with a design to be implemented. After reviewing the work, he frequently commented that they seemed to anticipate what he wanted or were able to implement what he had been unable to imagine. In addition to a comprehensive series of test scripts that were developed and executed, the program was also provided to industry professionals. Two beta tests involving early customer experience programs were used by the company for acceptance testing and both of these surfaced unanticipated areas for attention. Semper used formal bug and feature tracking software for major or outstanding problems but generally the developers tended to simply immediately fix problems once identified. Often the first indication that management had of a problem was when a fix was provided or noted in the change log. Discussing the need to document bugs, the programmers opined that judgment was used to determine if a bug report should be completed after the fact and that this was only done for difficult or particularly complex solutions.

Onsite Customer

Because Semper was an early-stage pre-market company, they did not have customers in the traditional sense. Instead, the product vision was provided by the CEO and the Director of Analytics. Originally trained as a mainframe programmer, the CEO was empathetic to technical problems but was not familiar with modern systems development and did not get involved in construction details. He would often jokingly describe programming and analytic modules as *"it is just a sort and a print right - what is the big deal – three to four hours programming tops*!" and would often laugh and offer to write some code himself if he thought some simple aspects were taking too long. He would challenge developers by reminding them that they learned little by programming simple tasks. A developer response to his question about a particularly complex change provides an example *"This is possible but will be hard to do. This is because* [text redacted]. *Anyway, I'm not going to start talking about the how-to parts. I know your response will be 'if it were easy, why would you want to do it?' ".* The Director of Analytics on the other hand, had current technical skills and would often interact directly with the developers and offer suggestions. Generally developers worked interactively with the management team and demonstrated prototypes for immediate feedback. Where planned requirements or changes necessitated extensive coding and development work, Unified Modeling Language use cases, conceptual sketches and data models were used as scaffolding to be discarded in favor of a prototype. A great deal of the management and developer communication was oral but the fact that offices were physically separated meant that email and instant messenger were used a great deal. The main design artefacts were the data model and build reports that identified progress and what was planned for or deferred to the next iteration.

Pair Programming

Pair programming was not adopted. Developers were dyadic but each within their own workstations. Modules were coded by one person although complex or difficult problems were shared. Although management discussed paired programming as an option with developers when they were hired (new applicants were interviewed by the programming staff and in addition to technical competency had to *"fit in"*) it was not pursued. Developers, hired directly from university where assignments and evaluations were competitive and individual; did not embrace collective approaches. While the environment was co-operative, developers would occasionally compete to see who could write the most efficient and effective code. Further exacerbating the difficulties with paired programming were work schedules, staff turnover, and personalities. Two of the development staff for example preferred to listen to iPods and to be isolated. Although programmers would often compete to see who could develop the better module they were reluctant to comment on code written by co-workers except in a joking manner. However, once a programmer left the company or was assigned to a different capacity they immediately became part of the out group and their code would often be referred to as *"strange"* , *"poorly written"* or *"in need of a re-write"*. Although developers would blame problems on former co-workers they would laugh when reminded that they may ultimately be subject to the same criticism. After one developer had been gone for six months another noted it was *"too late to blame* [redacted] *now".*

Planning Game.

Management realized that development had aspects of both art and science. Nevertheless the planning game was used extensively and trade-offs between time and features were routine. Estimates were almost exclusively provided by the developers and once established were treated as firm deadlines against which they were evaluated. Development was categorized into Structural Requirements, Differentiating Features, Human Computer Interaction, and Cosmetic changes.

Structural Requirements. Features and capabilities outlined in the business plan, considered core and treated as priority and foundational items.

Differentiating Features. Provided differentiating or competitive capabilities and were further grouped into *"must haves","nice to have"* and *"defer".* The majority of the *"must haves"* differentiated the product. Additions to this list resulted from competitive reviews or extensions to existing capabilities suggested by users. Typically a few *"must haves"* were included each week and developers knew that these could delay the build (there were two or three occasions where a deadline was missed). *"Nice to have"* items were optional. There were between eight to twenty of these each week although they were added to a cumulative list. Approximately three-quarters of these were included in each time box. *"Defer"* items were a combination of large and small features or changes that could be moved over time into the *"must have"* or *"nice to have"* group. Examples included the tutorial to complex encryption requirements that were included in subsequent builds.

*Human Computer Interaction.*Although management realized that HCI was important it was considered secondary to programming and design staff were not hired until the first version of the product had been completed. The main proponent of a more expanded view of usability was the Director of Analytics. Rather than criticize the existing product, he would usually make his point by identifying other products that he believed exemplified good design. The result of these comparisons was a complete re-write from the existing traditional Window's-based interface (Icons, Menu's and Pointers) to one that was much more intuitive and conversational. Despite the fact that Human Computer Interface issues were later seen as critical to the system and a great deal of time was spent in design, HCI was considered technically minor by the CEO.

*Cosmetic Changes.*Semper viewed all non programming changes as important to customers and use but mainly *"cosmetic".* There were numerous evolutions and changes to text, font, color, position and alignment. These were continuous and, in the words of a developer, were "*tedious but not hard".*

The frequency and approach used to manage these changes are described in TABLE 4.

TABLE 4. **Development Taxonomy**

Type	**Description**	**Number of Changes**	**Approach**
Structural	Fundamental aspects or product core.	<12	Simple Design & System Metaphor
Feature	Market and competitive requirements. Grouped into *"must have", "nice to have"* and *"defer"*	>100	On Site Customer, Planning Game, Simple Design, & Refactoring.
HCI	Usability issues such as placement of glyphs, screen dialogue and presentation.	>250	On Site Customer, Small Releases, Continuous testing, Refactoring.
Cosmetic	Icons, glyph, color, dialogue and position changes (not all simple).	>1,000	Onsite Customer, Refactoring, & Small Releases

Refactoring

Code focused on functionality and was continually refined and improved. The first product build, created after just two weeks, was essentially a shell program but was designated version 1.0.0. Substantive changes incremented the second order digit and minor changes usually incremented the low order identifier. In addition there were several major changes. For example a complete change in system interface required that all of the modules be re-written simultaneously and the main analytic engine (over 6,000 lines of code) was completely re-written over a two month period. As such, in addition to continuous improvement through refactoring there were periods of intense improvement in function, usability, reliability, speed and stability.

Simple Design

Development was guided by simple principles but trying to avoid architectural constraints or what the CEO called *"painting themselves into a corner".* Problems were designated BR or AR. BR were those that impacted customers and had to be fixed before revenue. AR were those could be solved with the increased resources provided after revenue. The planning game arbitrated between the cost of desired features and refactoring delivered functionality that was later improved. Conceptually developers were told to consider the metaphor of a *'modern digital camera',* where a high level of complexity and functionality was behind a simple interface that users could employ in a myriad of sophisticated ways.

Small Releases.

Time boxing was part of the discipline. Consequently, developers released a new version almost every second week. This was relaxed during major revisions and accelerated to almost daily versions when approaching a major deadline. In addition, management and users were also given replacement modules (DLLs) that delivered specific functionality, fixed problems or generally improved the code. Despite periods where developers complained that the ongoing short-term focus impeded delivery of a more systematic and quality oriented product, management remained committed to the concept of small releases. In a twelve month period developers delivered approximately 35 complete versions, with almost two dozen non-developer compiles and more than 150 replacement DLLs over and above the build cycle. Working through the planning game, management and the developers laid out a build schedule that was tracked using basic project management tools and rarely modified.

System Metaphor

Communication was simple and directly facilitated most often by the data model, the program itself, and the fact that with the exception of the Director of Finance and two junior business analysts all employees had been formally trained in systems analysis or computer programming. Design of the products was handled through a combination of strategic and tactical adjustments. Joint Application Design (JAD) sessions were used to begin product development and after each of the beta programs and before each of the three program redesigns. Tactically, designers and management met twice a week to receive the weekly build and to review progress, bug status and planned revisions to the upcoming version schedule.

We next draw conclusions about the degree and extent of appropriation, discuss limitations and suggest future research and implications.

V. CONCLUSION

Semper's partial adoption of agile principles reinforce other findings that indicate up to two thirds of large companies have adopted 'some form' of agile methods [9] which are then blended with more traditional practices. Practitioners have not adopted XP in an all or none action and faithful appropriation of all principles seems to be a rarity.

Initially Semper implemented only eight principles. Interestingly, three of the remaining four (continuous testing, shared code and coding standards) did later become more fully and faithfully appropriated. At first, it would appear that Semper should have applied more diligence in following agile principles from the outset. Alternatively, we suggest that these principles may have required a certain level of maturity not present in the organization's employees and processes. Coding standards were initially eschewed by management in favor of creativity, until a basic level of code had been developed. While the programming staff themselves favored standards, they were unable to agree on the specifics, until staff turnover and management support of a standard pressured them to do so. Similarly, developers still sought code ownership despite a concerted effort by management to curb such behavior. Paired programming, the only principle that did not manage to gain any momentum continues to be supported by management but has yet to be embraced by the developers. Other studies have found a similar lack of acceptance [15]. Therefore, we find that temporal conditions and institutional maturity affect the extent to which extreme programming principles are adopted and that both management and developer cultures are salient considerations. Consequently, future research should consider both cultural conditions and managerial preferences.

VI. ACKNOWLEDGEMENTS

This research was supported by a grant from Simon Fraser University. The authors appreciate the cooperation of Octothorpe Software Corporation.

REFERENCES

[1]. P. Abrahamsson, K. Conboyand X. Wang, 'Lots done, more to do': the current state of agile systems development research,*European Journal of Information Systems, 18*, 2009, 281-284

[2]. E. Geogiandou, Software Process and Product Improvement, *A Historical Perspective, Cybernetics and Systems Analysis, 39(1),* 2003, 125-142

[3]. W. Gibbs, Software's Chronic Crises, *Scientific American, 271(3)*, 1994, 89-96
[4]. D. Berry, M. Wirsingm, A. Knapp and B. Simonetta, The Inevitable Pain of Software Development: Why there is no silver bullet in *Radical Innovations of Software and Systems Engineering in the Future*. (Venice, 2002)
[5]. F. Brooks, *The Mythical Man Month* (Addison-Wesley, 1975)
[6]. E. Duggan, Silver Pellets for Improving Software Quality, *Information Resources Management Journal, 17(2)*, 2004, 1-21
[7]. 7. K. Beck, *Extreme Programming Explained: Embrace Change*,(Reading, Mass.: Addison -Wesley, 2000)
[8]. 8. J. HighSmith, Agile Software Development Ecosystems in A. Cockburn and J. HighSmith (Eds), *Agile Software Development Series*, ed. Boston: Addison-Wesley (2002)
[9]. 9. S. Ambler (2007) Survey says … Agile has crossed the chasm. Dr. Dobb's Journal 32(8) [WWW document] http://www.ddj.com/architect/ 200001986 (accessed 23 July 2013)
[10]. 10. AgileManifesto: The Agile Manifesto (2001)
[11]. 11. S. Goldman, N. Nagal and K. Preiss,*Agile Competitors and Virtual Organizations*(NY: Van Nostrand Reinhold, 1995)
[12]. 12. L. Williams and A. Cockburn, Agile Software Development: IT's About Feedback and Change, *Computer, 36(6)*, 2006, 39-43
[13]. 13. R. Yin, *Case Study Research: Design and Methods* (Thousand Oaks, CA: Sage Publications, 1994)
[14]. 14 J. Spradley, *The Ethnographic Interview* (New York, NY: Holt, Rinehart and Winston, 1979)
[15]. 15. G. Mangalaraj, R. Mahapatra and S. Nerur, Acceptance of software process innovations- the case of extreme programming, *European Journal of Information Systems, 18*, 2009, 344-354