Association for Information Systems
AIS Electronic Library (AISeL)

AMCIS 2001 Proceedings

Americas Conference on Information Systems (AMCIS)

December 2001

A Comparison of Requirements Engineering in Extreme Programming (XP) and Conventional Software Development Methodologies

Thomas Cohn
Managed Health Care Associates

Ravi Paul
New Jersey Institute of Technology

Follow this and additional works at: http://aisel.aisnet.org/amcis2001

Recommended Citation

Cohn, Thomas and Paul, Ravi, "A Comparison of Requirements Engineering in Extreme Programming (XP) and Conventional Software Development Methodologies" (2001). *AMCIS 2001 Proceedings*. 256.
http://aisel.aisnet.org/amcis2001/256

This material is brought to you by the Americas Conference on Information Systems (AMCIS) at AIS Electronic Library (AISeL). It has been accepted for inclusion in AMCIS 2001 Proceedings by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.

A COMPARISON OF REQUIREMENTS ENGINEERING IN EXTREME PROGRAMMING (XP) AND CONVENTIONAL SOFTWARE DEVELOPMENT METHODOLOGIES

Thomas M. Cohn
Managed Health Care Associates
tcohn@mhax.com

Ravi C. Paul
New Jersey Institute of Technology
paulr@cis.njit.edu

Abstract

Extreme Programming (XP) is the latest software development methodology to hit the stands. XP is being touted as ***the*** *methodology of choice for creating business applications in "Internet time". One of the ways XP supports rapid application development is by collapsing the requirements phase. How is this done and how is requirements engineering in XP different from the conventional methodologies? What issues does this raise for research? This paper addresses these questions. First, a brief introduction to requirements engineering in conventional development methodologies is provided. Next, the Extreme Programming development cycle is presented. Finally, the differences in requirements engineering processes between XP and conventional methodologies are explored and research questions presented.*

Requirements Engineering Overview

Organizations that do not use a methodology to develop software have been shown to constantly operate in crisis mode, to have difficulty meeting commitments and to depend on the heroics of individuals (Paulk 1995). In other words, it makes sense to develop software systems in an orderly, planned and repeatable way using a standard methodology. The following are some of the more popular life cycle methodologies: Waterfall, Prototyping, Incremental Development, Evolutionary, and Spiral (Dorfman 1997). The requirements phase of these methods is generally implemented by the requirements engineer, who is ideally not also involved in the design or development phase of the system.

Requirements Engineering can be divided into two main activities – Requirements development and Requirements management. The requirements development phase typically includes elicitation, analysis, specification and verification & validation. The requirements management phase extends throughout the life cycle and is mainly focused on change management and impact analysis.

Faulk (1997) describes the requirements phase as having two goals.

Problem Analysis: This is the process of the requirements engineer understanding the customer's problem, the purpose of the system being developed and the constraints on the system.

Requirements Specification: The product of this goal is a document known as the Software Requirements Specification (SRS) that details what a system should do without saying how to do it. This document captures everything learned in problem analysis. This is the document that will guide the software design, development and testing phases.

There are many techniques to assist in the requirements phase of development. These include interviews, prototypes, JAD/RAD sessions, Protocol Analysis, and Use Cases (Gougen and Linde 1993, Gause 1989, Dorfman 1997, Rumbaugh 1994). The development of specification documents using one or more of these techniques is the "traditional" way of developing requirements.

An Introduction to Extreme Programming

Extreme Programming (XP) was developed as a response to some of the problems with conventional methodologies such as lack of adaptability and flexibility. The creators of XP were looking for an alternative to the traditional methodologies, which they considered "heavy". Proponents of the XP methodology claim that the combination of the XP development process along with the XP programming principles produces an effective and efficient development life cycle. McGregor (2001) describes an XP project in the following way: "The goal of an Extreme Project seems to be producing a software product with acceptable quality in the least amount of time possible." Beck (2000) originally described XP as a method with the following lifecycle (Figure 1).

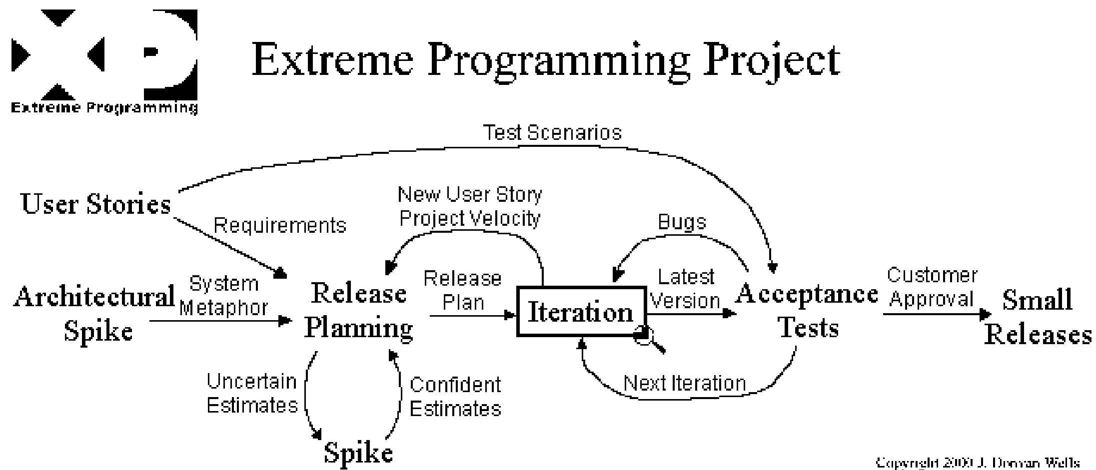

Figure 1. The Extreme Programming Lifecycle

A customer writes what are known as "User Stories" on numbered index cards. These user stories are one-paragraph descriptions in plain English of what they want the system to do. Next, an iteration planning meeting is held. At this meeting, the customer reads the user stories and the development team asks questions until they understand the story. Based on this, the development team determines what tasks are required to meet the needs of the story. They then estimate the time required to complete the development of a story. These time estimates, along with a risk ranking, are written on the back of each user story.

The risk ranking is based on the developer's confidence in being able to implement a solution. If the developers are not sure about how to implement a solution to the user story, they create a "Spike Solution". The spike solution is an exploratory exercise that gives the developer the domain or technical knowledge required to create a time estimate.

The next step is a release planning meeting which is held with the customer. At this meeting, the customer prioritizes the user stories taking into account the time estimate for each one. The team then develops a release plan, which details the stories that will be included in the next three software releases. Subsequently, developers create unit tests along with the actual code. At completion, the code for each individual story is released for acceptance testing with the rest of the baseline code. This happens, minimally, once per day. The customer plans the acceptance test. After customer approval of the acceptance tests, a release is created.

XP and Requirements Engineering Activities

The following section highlights the main differences between XP and conventional development methodologies across the primary requirements engineering activities.

Elicitation

One of the primary artifacts of interest and the sole method for requirements gathering in XP are user stories. As the name suggests, user stories are simply descriptions of what the system needs to do, written by the actual user as a concise, free-flowing, story on index cards. They are similar, but not the same, as a very popular tool in traditional requirements elicitation – the Use Case. The following table (Table 1) compares these two techniques.

Table 1. Comparison of Use Cases and User Stories (XP)

Use Cases	User Stories
Consistent, complete, do not overlap, do not contradict	Are just stories – they may break these rules
One cannot contain other use cases that are not related	Do not distinguish between different scenarios
Structured	Unstructured – A set of cards written in plain English
Do not have any measurements	Measured by risk, estimated time and priority
Can only be from the perspective of one actor	Have no perspective constraints

User stories do not need to be decomposed until they reveal every detail of a process. In fact, it is difficult to get a customer to write down all of the details of a story. In this case, the user story functions as the starting point for a future conversation between the customer and the developer. For this reason the user story should contain enough information to remind the customer of all the details that need to be discussed when the story is explored later.

The following is an example of a user story: "When a customer orders an item they are charged based on either a discount off of list price or a markup above cost depending on how the customer is setup. Discounts off of list price are maintained by the marketing department while markups from cost are maintained by the sales department. In either case the customer may be eligible for a quarterly rebate that is negotiated by the buying department." In the traditional development scenario, this user story would be represented by multiple use cases from the perspective of at least five actors. Additional elicitation would need to occur before parts of this user story could be converted into a use case.

Analysis

The goal of requirements analysis is to develop a complete understanding of the problem to be solved and to uncover conflicting needs. Some of the popular, traditional techniques used to analyze requirements are: structured analysis, object-oriented analysis and formal methods. In an XP project the programmers have continuous feedback with one customer who is the voice of all requirements. Having a single voice for all requirements helps eliminate conflicting needs that are typically uncovered during requirements analysis. XP does not have a separate requirements analysis phase, requirements go from the elicitation phase directly into the development phase. For software development, XP favors object-oriented languages because of their adaptability to change. In XP, writing code is an additional analysis technique. Beck (2000) maintains that the simplest object-oriented designs occur after some test cases and code has been written and refactored. Only after reflection upon the problem to be solved in conjunction with the knowledge and insight gained by programming is the final object-oriented design developed. In XP the phases of software engineering are: requirements, test, code then design.

Specification

The complete and accurate specification of requirements is the ultimate goal of the requirements development process. In XP, there is no requirement to develop a formal SRS to record all the requirements under strict guidelines. Instead, a set of informal, user stories is expected to take its place. Looking at user stories collectively from a standards perspective clearly suggests that they are not a unique set, normalized, a linked set, complete or consistent. They do, however, satisfy the properties of being abstract, bounded, modifiable, traceable, testable, configurable and granular. The concept of User Stories in XP also satisfies most of the categorization requirements (Identification, Priority, Criticality, Feasibility, Risk, and Source) of the IEEE guide. Furthermore, user stories help avoid some of the pitfalls described in the IEEE Guide, such as the mixing of design and implementation details during requirements elicitation, overspecification, etc. As pointed out by Scharer (1981), traditional requirements typically suffer from overspecification problems such as "The Kitchen Sink" (including everything that might possibly be needed) and "The Smoke Screen" (veiling true requirements with extras that are not needed). XP's solution to this problem is a principle known as "You aren't going to need it". Jefferies (2000) describes the principle as: "Don't build for tomorrow. When you hear yourself elaborating or generalizing a design, stop. Implement the simplest design that could possibly work to do what you have to do right now. When you say 'We're gonna need it', you're wasting precious time, and you`re usurping the customer's right to set priorities."

Verification and Validation

In projects using traditional methodologies, this phase involves the testing of the final software product based on the System Requirement Specification (SRS), to ensure accuracy and adequacy and thus improve quality. Wallace (2000) describes this as "Software verification examines the products of each development activity (or increment of the activity) to determine if the software development outputs meet the requirements established at the beginning of the activity". In XP there is no SRS. The testing responsibility is divided between developers and customers. Developers write test cases as part of code development and testing (both unit and integration) is completely integrated into the development process. In order to test that all requirements are included in the final product, XP depends on the customer conducting acceptance tests. The customer conducts the acceptance tests by checking the software against the user stories that they have written.

Requirements Management

Change is inevitable. System and software requirements will evolve and change. Requirements management (RM) is primarily interested in helping firms manage these changes effectively. This includes "monitoring the addition of new requirements, the deletion of old requirements, and the changes made to existing requirements." (Yourdon 1998). Although the credo of XP is "Embrace Change", XP is not well suited for projects that require documented traceability of all changes. When a change to a requirement is needed in XP a new user story is written. The requirement is also encapsulated in two other entities: the unit test and the customer scripted acceptance test. XP manages change by ensuring that all unit test and acceptance test for the entire system pass every time a new software version is created. Changes start as a user story but are only recorded in the application software code, the unit tests and the acceptance tests. By dealing with change in a flexible lightweight manner, Beck (2000) maintains that the cost of change no longer rises exponentially over time.

Research Issues

Is the promise of XP real? Under what circumstances? Some of the fundamental "problems" frequently attributed to project failures and the reason a formal requirements engineering process is stressed are addressed by the key tenets of XP such as simplicity, user involvement, continuous testing, and small and frequent releases. However, whether XP really addresses the problems and the extent to which it does, are still not clear. The following are some key questions related to requirements engineering that need to be addressed.

- What kind of projects can benefit from XP? What projects are "too big" and therefore outside the scope of XP?
- How do we reconcile past research, which suggests that the lack of a formal specification is a recipe for project disaster? What is the effect of the lack of focus, in general, with a formal requirements engineering process?
- XP requires customers that can write user stories, assist in planning releases, perform acceptance tests, and be available to answer questions on-site with the developers. Ideally, this is one person who can represent a group of users. What are the "ideal" characteristics that this person must possess? How are different, and potentially conflicting, stories from multiple "customers" handled?
- What are the implications of not having a formal document (such as an SRS) to serve as a binding contract?
- How is change managed after release? How is the potential impact of changes assessed? Can the "cost of change" curve truly be flattened with XP?
- Does the use of XP indeed result in project "success"? Are the success factors for XP implementation different than for the traditional methodologies?

A variety of quantitative and qualitative methods and measures will need to be used to address these questions. An appropriate framework with multiple dimensions of IS project success should be used to measure the "effectiveness" and "success" of XP projects in the real world. For example, Saarinen (2000) proposed a model for project success that included measures assessing satisfaction along four dimensions. 1 – success of development process; 2 – success of use; 3 – quality of application; and 4 – impact of IS on organization. Data for these measures should be collected from various stakeholders through interviews and surveys. In addition, data such as time to complete project, number of errors/lines of code, etc. should be collected to quantitatively assess performance. Some of the other questions - for example the ones about customer characteristics - will need to be answered using qualitative methods such as observation and protocol analysis. Controlled experiments with groups working on the same project – some using XP and others a traditional methodology – would provide opportunities for comparison of process and results.

Conclusions

In today's fast economy, getting your product to market in the least amount of time is critical to the success of a business venture. Additionally, the time and expense for developing formal software specifications may not be justified for smaller projects. XP would seem to be the solution in these cases. XP also seems to be particularly well suited to projects where the requirements are in a constant state of change. On the other hand, since every requirement is not scrutinized to the lowest level of detail before design and development, certain constraints may be overlooked. As a result, XP may not be well suited to projects that produce high-reliability software, or projects that are primarily governed by constraints such as a missile guidance systems or projects that have strong security requirements.

Extreme Programming saves time in the requirements phase by setting up a structure that minimizes the amount of time the same problem must be explored by different individuals. Unfortunately, the saving in time comes at the expense of increased risk. On the other hand XP strives to take a user-centric approach. It is designed to prevent lack of user involvement as being a risk that can lead to project failure. Continual feedback and dialog take the place of documentation. While Extreme Programming, as a methodology, seems to have a lot of promise, it is also clear that it requires much further study. The successes of XP thus far have been largely anecdotal. In order to fully take advantage of the potential possibilities of XP, there are several issues that must be addressed. Some of the potential research topics related to requirements development and management were presented in this paper.

References

Beck, K., *Extreme Programming Explained: Embrace Change*, Addison Wesley, 2000.

Dorfman, M., "Requirements Engineering", *Software Requirements Engineering,* IEEE, 1997, pp. 7-22.

Faulk, S., "Software Requirements: A Tutorial", *Software Requirements Engineering,* IEEE, 1997, pp. 158-179.

Gause, D., *Exploring Requirements*, Dorset House, 1989, pp. 249.

Goguen, J. A. & Linde, C., "Techniques for Requirements Elicitation", *Proceedings of the International Symposium on Requirements Engineering,* 1993.

IEEE Std 1233, "IEEE Guide for Developing System Requirement Specifications", *Software Requirements Engineering,* IEEE, New York, 1998.

Jefferies, R., *Extreme Programming Installed,* Addison Wesley, 2000, p. 125.

McGregor, J., "Quality Assurance Taking Testing to the Extreme", *Journal of Object Oriented Programming*, February 2001.

Paulk, M., *The Capability Maturity Model: Guidelines for Improving the Software Process,* Addison-Wesley, 1995.

Rumbaugh, J., "Getting Started: Using Use Cases to Capture Requirements", *Journal of Object Oriented Programming*, September 1994.

Saarinen, T., "An expanded instrument for evaluating information systems success," *Information & Management*, 1996, pp. 103-118.

Scharer, L., "Pinpointing Requirements", *Datamation,* April 1981.

Wallace, D. & Ippolito, L., "Verifying and Validating Software Requirement Specifications", *Software Requirements Engineering*, IEEE, 2000, p. 437.

Yourdon, E., "Requirements Management: A New Look", *Cutter IT Journal*, 1998