The Quality of Requirements in Extreme Programming

Richard Duncan
Mississippi State University

Extreme Programming (XP) is a software process methodology that nominates writing code as the key activity throughout the development process. While at first glance this sounds chaotic, a disciplined group utilizing XP performs sufficient requirements engineering. This paper describes and evaluates the quality of requirements generated by an ideal group using XP and discusses how the XP process can assist or hinder proper requirements engineering.

Extreme Programming (XP) is a hot, new software process methodology for medium to small sized organizations. It is designed with requirements drift as a fundamental occurrence to be embraced, rather than dealing with it as a necessary evil. XP nominates coding as the key activity throughout the development process, yet the methodology is based on economics [1].

Dr. Barry Boehm presented that the cost of change grows exponentially as the project progresses through its lifecycle [2], Stuart Faulk reiterates this by stating that the relative repair cost is 200 times greater in the maintenance phase than if it is caught in the requirements phase [3]. XP challenges that this is no longer the case. While it is more expensive to modify code than to modify a prose description, with modern languages and development techniques it is not an exponential increase.

Instead, Beck asserts that the cost for change levels out. Rather than spend extra effort in the requirements analysis phase to nail down all requirements (some of which will become obsolete through requirements drift anyway), accept that changes due to incomplete requirements will be dealt with later. XP assumes that lost resources in rework will be less than the lost resources in analyzing or developing to incomplete requirements [1].

The primary vehicle for requirements elicitation in XP is adding a member of the customer's organization to the team. This customer representative works full time with the team, writing stories – (similar to Universal Markup Language (UML) Use Cases) – developing system acceptance tests, and prioritizing requirements [4]. The specification is not a single monolithic document; instead, it is a collection of user stories, the acceptance tests written by the customer, and the unit tests written for each module. Since the customer is present throughout the development, that customer can be considered part of the specification since he or she is available to answer questions and clear up ambiguity.

The XP life cycle is evolutionary in nature, but the increments are made as small as possible. This allows the customer (and management) to see concrete progress throughout the development cycle and to respond to requirements changes faster. There is less work involved in each release, therefore the time-consuming stages of stabilization before releases take less time. With a longer iteration time it may take a year to incorporate a new idea: with XP this can happen in less than a week [2].

A fundamental of XP is testing. The customer specifies system tests, the developers write unit tests. This test code serves as part of the requirements definition – a coded test case is an unambiguous medium in which to record a requirement. XP calls for the test cases to be written first, and then the simplest amount of code to be written to specify the test case. This means that the test cases will exercise all relevant functionality of the system, and irrelevant functionality should not make it into the system [1].

This paper describes and evaluates the requirements engineering processes associated with the XP paradigm.

The XP Requirements Engineering Process

Harwell et al. break requirements into two types – product parameters and program parameters. A product parameter applies to the product under development, while a program parameter deals with the managerial efforts that enable development to take place [5]. The customer who becomes a member of the XP team defines both product and program parameters. The product parameters are defined through stories and acceptance tests, while the program parameters are dealt with in release and iteration planning.

The product parameters are chiefly communicated through *stories*. These stories are similar to Use Cases defined in UML, but are much simpler in scope [4]. Developing a comprehensive written specification is a very costly process, so XP uses a less formal approach. The requirements need not be written to answer every possible question, since the customer will always be there to answer questions as they come up. This technique would quickly spiral out of control for a large development effort, but for small- to medium-sized teams (teams of fewer than 20 people are most often reported) it can offer a substantial cost savings. It should be noted, however, that an inexperienced customer representative would jeopardize this property.

The programmers then take each story and estimate how long they think it will take to implement it. Scope is controlled at this point – if a programmer thinks that the story, in isolation, will take more than two weeks to implement, the customer is asked to split the story. If the programmers do not understand the story they can always interact directly with the customer. Once the stories are estimated, the customer selects which stories will be implemented for the upcoming release, thereby driving development from business interests. At each release, the customer can evaluate if the next release will bring business value to the organization [1].

Each story to be implemented is broken up into tasks. A pair of programmers will work to solve one task at a time. The first step in solving a task (after understanding, of course) is to write a test case for it. The test cases will define exactly what needs to be coded for this task. Once the test cases pass, the coding is complete [1]. Thus the unit tests may be considered

a form of requirements as well. Every test (across the entire system) must pass before new code may be integrated, so these unit-test requirements are persistent. This is not to say that simple unit testing counts as an executable specification – but XP's test-driven software development does record the specific requirements of each task into test cases.

The final specification medium for product requirements is the customer acceptance tests. The customer selects scenarios to test when a user story has been correctly implemented. These are black-box system tests, and it is the customer's responsibility to ensure that the scenarios are complete and that they sufficiently exercise the system [6]. These acceptance tests serve as an unambiguous determiner as to when the code meets the customer's expectations.

How XP Rates

The XP requirements engineering process can be analyzed by considering the 24 quality attributes for software requirements specification (SRS) proposed by [7]. Davis et al. propose that a quality SRS is one that exhibits the 24 attributes listed in Table 1. Rather than applying these metrics to a given document, they are used here to measure the requirements that theoretically come out of the XP process. Of course, a quality SRS is mostly dependent on the discipline used by the people associated with the project, but specific features of XP can influence the quality of a SRS.

A specification created with XP would appear to score very well across most of these attributes, but fare poorly on others. Those qualities with a "+" symbol indicate that the subsequent paragraphs argue the XP process can lead to an improvement in the area: a "-" that XP detracts from the quality. The "+/-" annotation indicates that XP partially helps and partially harms a specification in achieving the quality. Many of the qualities are not addressed by XP and are hence annotated with a "?," for these qualities a group's organization, discipline, and specific project needs will decide. It should be noted that to religiously follow XP requires a great deal of discipline: This discipline should be expected to carry over into the other qualities. Following is a look at some of the quality attributes.

Unambiguous, Correct, and Understandable: Since the customer is present, ambiguity and problems understanding the requirements are generally minimal and easily solvable [1] . Requirements are correct if and only if each represents an actual requirement of the system to be built. Since the customer writes the stories from business interests, the requirements should all be correct. With so much responsibility and freedom, clearly the selection of an appropriate customer representative is crucial to the success of the project. Even if the customer does not know exactly what he or she desires at the start of the project, the evolutionary nature of XP development leads to a system more in line with the customer's needs.

Modifiable: The XP lifecycle allows changes to the requirements specification at nearly any point in system development. The specification exists as a collection of user stories, so the customer can switch out one future story for another with little impact on existing work. Since the planning, tests, and integration are all performed incrementally, XP should receive highest marks in modifiability. Of course, work may be lost in this changeover, but with XP the programmers should be able to estimate how much a change will cost.

Unambiguous, Verifiable: Since the customer writes acceptance tests (with the assistance of programmers), it could be argued that the functional specification is recorded in an unambiguous format. Furthermore, the first activity performed by a programming pair to solve a task is to write test cases for it. These test cases become a permanent part of the specification/test suite. Customers (with the help of the XP coach) will also make sure that the specification is verifiable, since they know that they will have to write test cases for it.

Annotated by Relative Importance: The customer defines which user stories they wish implemented in each release. Hence, each requirement is annotated by relative importance at this time – the customer should for ask the highest-priority stories to be implemented first and the programmers are never left guessing priorities.

Achievable: Since each release provides some business value, a portion of the system found to be unachievable should not leave the customer with a very expensive yet unusable piece of technology. If the high-risk piece is important, it will be implemented first, in which case the unachievable component should be found quickly and the project aborted relatively inexpensively. If it is less important, then the system may be delivered in useful form without it

Design Independent: Design independence is a classic goal for requirements, but today's object-oriented development methods recognize that design independent requirements are often impractical. Portions of the requirements (such as the user stories) can be very design independent, but the unit tests that are archived as part of the requirements and used to cross-check new modules may depend heavily

Table 1: *The 24 Quality Attributes [7]*

1. Unambiguous	+	13. Electronically Scored	+/-
2. Complete	-	14. Executable/Interpretable	+/-
3. Correct	+	15. Annotated by Relative Importance	+
4. Understandable	+	16. Annotated by Relative Stability	?
5. Verifiable	+	17. Annotated by Version	+
6. Internally Consistent	+/-	18. Not Redundant	-
7. Externally Consistent	+/-	19. At the Right Level of Detail	?
8. Achievable	+	20. Precise	?
9. Concise	+	21. Reusable	?
10. Design Independent	+/-	22. Traced	?
11. Traceable	?	23. Organized	?
12. Modifiable	+	24. Cross-Referenced	?
13.		25.	

+/- indicates XP both assists and degrades.
? indicates XP has little bearing on the area.
+ indicates XP may assist in this area.
- indicates XP degrades this area.

on the actual system.

Electronically Stored: XP calls for the stories to be written on index cards, so this portion of the requirements is not electronically stored. While the stories could be placed in a word processor, Jeffries et al. assert that handwritten index cards produce less feelings of permanence and allow the customer to more freely change the system [4]. The customer is also available as a requirements resource, obviously not electronically stored. However, the requirements are written on individual cards so modifications can often be localized to a single card if rewriting is necessary. Furthermore, the customer codifies the system requirements with acceptance tests, so it could be argued that the most important part of the specification is stored.

Complete, Concise: XP stresses programming as the most important development activity, hence little effort is spent on creating documents, therefore the specification is very concise. The cost may be a lack of completeness, however. Since little up front analysis takes place, there may very well be holes in the system. Yet the customer drives what functionality is implemented and in what order, so true functionality should not be left out. Furthermore, since the XP process accommodates change, it should be possible to compensate for these holes later in the development lifecycle.

Security Assurances

Since the XP development methodology does not progress from a verified requirements document, how might a system developed with XP rate on a security evaluation? The Common Criteria has seven evaluation assurance levels (EAL1-EAL7). For EAL5 and above the Common Evaluation Methodology calls for the system to be semi-formally designed and tested [8]. This leaves two questions to be addressed. First, can a project use formal methods with XP? Second, without formal methods, how trusted can a system developed under XP be?

The XP process screams informality in many respects. The name alone conjures images of snowboarders with laptops, and even the books about XP are written in a conversational tone. Nevertheless, what would happen if the customer writes stories and they are annotated with a formal specification? Clearly, this would entail a large cost in training personnel, writing the specifications, and verifying the specifications. This also reduces the agility of the XP product – since more money is spent on specification, the cost of change will increase. But if each story were rewritten in a formal notation it would be possible to formally verify the specification and design.

Formal methods aside, the way an XP project progresses does offer many assurances of trust. First, all code is written directly from the user stories (the specification). All functionality is tested in the unit tests and all integrated code is required to pass all tests all the time. While testing does not guarantee the absence of errors, many security holes come from poorly tested software. Hence, the test-oriented nature of XP may be a great step forward.

A strong security feature of XP is pair programming. The observer in a pair constantly evaluates the code being written by his or her partner. This programmer can help reduce the probability of coding errors that might later be exploited (e.g., buffer overruns). XP also adds counterbalances to reduce the impact of a single malicious coder (either in a truly malevolent sense or inadvertently opening holes as Easter Eggs[2] side effects) through the pairing process. Rather than just inserting code into the system, one programmer would have to convince the other of a rationale for why the code was being inserted. Due to collective code ownership, it is entirely possible that the next pair in the course of re-factoring would catch malicious code. Pair programming and collective code ownership add further assurance that the code is written exactly to the specification.

Conclusions

XP performs requirements engineering throughout the life cycle in small informal stages. The customer joins the development team full time to write user stories, develop system acceptance tests, set priorities, and answer questions about the requirements. The stories are simpler in scope to use cases because the customer need not answer every conceivable question. The informal stories are then translated into unit and system acceptance tests, which have some properties of an executable specification.

Of the 24 quality attributes of a software specification, the XP process leads to higher points in nine attributes and lowers the score in two. The most noteworthy gains are in ambiguity and understandability, since the customer is always present to answer questions and clear up problems. Furthermore, since the customer is also responsible for developing test scenarios he or she will create more verifiable requirements. The discipline enforced by the XP process should also carry over into other areas of requirements engineering.◆

Notes

1. XP has been used on several projects. Beck mentions a campaign management database, a large-scale payroll system, a cost analysis system, and a shipping and tariff calculation system in (9). Yet there is little objective data available for analysis at this time. More data should be made available through the upcoming XP Universe Conference, Raleigh, NC, July 2001, www.xpuniverse.com
2. An unsolicited, undocumented piece of code a programmer inserts into software, generally for his or her own amusement.

References

1. Beck, Kent, *Extreme Programming Explained: Embrace Change*, Boston, Addison Wesley, 2000.
2. Boehm, Dr. Barry, *Software Engineering Economics*, Prentice Hall, 1981.
3. Faulk, Stuart, Software Requirements: A Tutorial, *Software Engineering*, ***[What Month?]*** 1996, pp. 82-103.
4. Jeffries, Ron; Anderson, Ann; and Hendrickson, Chet, *Extreme Programming Installed*, Boston, Addison Wesley, 2001.
5. Harwell, Richard; Aslaksen, Erik; Hooks, Ivy; Mengot, Roy; and Ptack, Ken, What is a Requirement? Proceedings of the Third Annual International Symposium National Council Systems Engineering, 1993, pp. 17-24.
6. Extreme Programming: A Gentle Introduction, www.ExtremeProgramming.org

7. Davis, Alan; Overmyer, Scott; Jordan, Kathleen; Caruso, Joseph; Dandashi, Fatma; Dinh, Anhtuan; Kincaid, Gary; Ledeboer, Glen; Reynolds, Patricia; Sitaram, Pradhip; Ta, Anh; and Theofanos, Mary, Identifying and Measuring Quality in Software Requirements Specification, Proceedings of the First International Software Metrics Symposium, 1993, pp. 141-152.

8. Common Criteria, International Standard (IS) 15408, csrc.nist.gov/cc/ccv20/ccv2list.htm, September 2000.

9. Beck, Kent, Embracing Change with Extreme Programming, *Computer*, October 1999, pp. 70-79.

About the Author

Richard Duncan is pursuing a master's degree in computer science at Mississippi State University (MSU) with emphasis in software engineering. He has held summer internships at Microsoft, AT&T Labs Research, and NIST. His current research interests involve applying software-engineering process to the development of a public domain speech recognition system at the Institute for Signal and Information Processing at MSU.

Mississippi State University
P.O. Box 9571
Mississippi State, MS 39762
Voice: 662-325-8335
Fax: 662-325-2292
E-mail: Richard.Duncan@ieee.org

Letters to the Editor

Editor:

Just a note to let you know the address for the SPI EGroups web site you reference on page 19 of March's CrossTalk needs to be updated. Yahoo recently took over eGroups so the address is now
http://groups.yahoo.com/group/spi

I am the creator, owner, and moderator of this group, and just out of curiosity, was wondering how you came about including it in this issue. I'm very pleased to see it there!

Thanks,
Maj. Andrew D. Boyd
Chief, Software Quality Assurance Section
USAF

Have We Lost Our Focus?

Having recovered from a wonderful case of laryngitis, I just got back from a great Software Engineering Process Group conference in New Orleans. A great time was had by all. Between the SEPG conference and reviewing papers for this issue of CrossTalk, I am totally up-to-date on the latest and greatest in methodologies.

Back in 1969 I started my life as a programmer/developer/computer scientist/software engineer. (As you get promotions, you don't just get pay raises. You get neater and spiffier job titles, too!). I first learned to program a Wang programmable calculator in junior high – a gift to the school from a local company that needed a tax break. At that time the calculator was the size of a desk and filing cabinet with (a whopping) 256 bytes of memory, a paper tape reader, and a single punch card input. Not a *deck* of cards, mind you, but a single card: punched by hand, eighty columns, 12 rows.

There were only 61 possible instructions in the Wang instruction set, so each row in the punched card could encode two six-bit operations. Therefore, each card could contain 160 instructions. To gain admission to the "Computer Science Club," you had to be able to program the quadratic equation on a single card. (Remember $ax^2 + bx + c = 0$, given a, b, and c, solve for x?) Once you accomplished this Herculean task, why, you were considered a programmer!

I remember how *smart* I felt after accomplishing this task. I was a hacker, a member of the brotherhood/sisterhood! I could code! We didn't use a methodology – design was for wimps! I remember trying to explain to my girlfriend about what I had accomplished. (This might explain why I didn't date much during school).

Fortunately, I matured in my profession. I joined the Air Force and had to learn how to design. As my first design methodology, I flowcharted my code. Of course – as a true hacker – I knew that if I finished the code first, the flowcharting was much easier. I was lucky – I had a wise and tolerant boss who showed me that flowcharting was a requirements and design tool. That when the problem got too big for me to understand, I had to use some tool to help me understand and partition the problem.

I eventually became both a computer scientist and later, a software engineer. Over the years, I went from flowcharting to Top Down Structured Design (TDSP); Hierarchical Input, Process, Output (HIPO); Structured Analysis and Structured Design (SASD); Transform Analysis (TA); Object-Oriented Modeling and Design (OOMD); Unified Modeling Language (UML); and Unified System Development Process (USDP). I tried to become a Certified Data Processor (CDP). I earned degrees from institutions that are Computer Science Accreditation Board-compliant (CSAB). I studied the Software Engineering Body of Knowledge (SWEBOK). I learned lots of acronyms and new methodologies – new methodologies? You know, when I first saw activity diagrams and sequence diagrams in UML and USDP, I felt right at home. I had come full circle. I was flowcharting again!

It's never been about the methodology. Methodologies are simply techniques and tools to help you partition and understand the problem. Here's a kernel of truth – you can't design a system (let alone code it) unless you understand what the system is supposed to do. The methodologies are there to assist me. They are NOT ends in themselves – they are simply a means to the end. You have to focus on the larger picture – getting the system "out the door" on time. It's not enough to be an expert in the latest and greatest methodology, unless you can also use it to help produce a system on time and under budget. The methodology has to make you more productive. If a methodology helps, use it! If it doesn't, get a better methodology! Focus on the ends, not the means.

And – if all else fails – try drawing a flowchart. It might make you feel SMART again!

David A. Cook
Principal Engineering Consultant
Shim Enterprises, Inc.
david.cook@hill.af.mil