EXTREME PROGRAMMING

2.1 Extreme programming (**XP**) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development,[1][2][3] it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.

Other elements of extreme programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, simplicity and clarity in code, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers.[2][3][4] The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels. As an example, Code reviews are considered a beneficial practice; taken to the extreme, code can be reviewed *continuously*, i.e. the practice of Pair programming.

Critics have noted several potential drawbacks,[5] including problems with unstable requirements, no documented compromises of user conflicts, and a lack of an overall design specification or document.

2.2 History

Extreme Programming was created by Kent Beck during his work on the Chrysler Comprehensive Compensation System (C3) payroll project.[5] Beck became the C3 project leader in March 1996 and began to refine the development methodology used in the project and wrote a book on the methodology (in October 1999, *Extreme Programming Explained* was published).[5] Chrysler cancelled the C3 project in February 2000, after seven years, when the company was acquired by Daimler-Benz.[6]

Although extreme programming itself is relatively new, many of its practices have been around for some time; the methodology, after all, takes "best practices" to extreme levels. For example, the "practice of test-first development, planning and writing tests before each micro-increment" was used as early as NASA's Project Mercury, in the early 1960s (Larman 2003). To shorten the total development time, some formal test documents (such as for acceptance testing) have been developed in parallel (or shortly before) the software is ready for testing. A NASA independent test group can write the test procedures, based on formal requirements and logical limits, before the software has been written and integrated with the hardware. In XP, this concept is taken to the extreme level by writing automated tests (perhaps inside of software modules) which validate the operation of even small sections of software coding, rather than only testing the larger features.

Origins

Software development in the 1990s was shaped by two major influences: internally, object-oriented programming replaced procedural programming as the programming paradigm favored by some in the industry; externally, the rise of the Internet and the dot-com boom emphasized speed-to-market and company growth as competitive business factors. Rapidly changing requirements demanded shorter product life-cycles, and were often incompatible with traditional methods of software development.

The Chrysler Comprehensive Compensation System (C3) was started in order to determine the best way to use object technologies, using the payroll systems at Chrysler as the object of research, with Smalltalk as the language and GemStone as the data access layer. They brought in Kent Beck,[5] a prominent Smalltalk practitioner, to do performance tuning on the system, but his role expanded as he noted several problems they were having with their development process. He took this opportunity to propose and implement some changes in their practices based on his work with his frequent collaborator, Ward Cunningham.

Current state

XP generated significant interest among software communities in the late 1990s and early 2000s, seeing adoption in a number of environments radically different from its origins.

The high discipline required by the original practices often went by the wayside, causing some of these practices, such as those thought too rigid, to be deprecated or reduced, or even left unfinished, on individual sites. For example, the practice of end-of-day integration tests for a particular project could be changed to an end-of-week schedule, or simply reduced to mutually agreed dates. Such a more relaxed schedule could avoid people feeling rushed to generate artificial stubs just to pass the end-of-day testing. A less-rigid schedule allows, instead, for some complex features to be more fully developed over a several-day period. However, some level of periodic integration testing can detect groups of people working in noncompatible, tangent efforts before too much work is invested in divergent, wrong directions.

Meanwhile, other agile development practices have not stood still, and XP is still evolving, assimilating more lessons from experiences in the field, to use other practices. In the second edition of *Extreme Programming Explained* (November 2004), five years after the first edition, Beck added more values and practices and differentiated between primary and corollary practices.

Concept

Goals

Extreme Programming Explained describes extreme programming as a software-development discipline that organizes people to produce higher-quality software more productively.

XP attempts to reduce the cost of changes in requirements by having multiple short development cycles, rather than a long one. In this doctrine, changes are a natural, inescapable and desirable aspect of software-development projects, and should be planned for, instead of attempting to define a stable set of requirements.

Extreme programming also introduces a number of basic values, principles and practices on top of the agile programming framework.

Activities

XP describes four basic activities that are performed within the software development process: coding, testing, listening, and designing. Each of those activities is described below.

Coding

The advocates of XP argue that the only truly important product of the system development process is code – software instructions that a computer can interpret. Without code, there is no working product.

Coding can also be used to figure out the most suitable solution. Coding can also help to communicate thoughts about programming problems. A programmer dealing with a complex programming problem, or finding it hard to explain the solution to fellow programmers, might code it in a simplified manner and use the code to demonstrate what he or she means. Code, say the proponents of this position, is always clear and concise and cannot be interpreted in more than one way. Other programmers can give feedback on this code by also coding their thoughts.

Testing

Extreme programming's approach is that if a little testing can eliminate a few flaws, a lot of testing can eliminate many more flaws.

- Unit tests determine whether a given feature works as intended. A programmer writes as many automated tests as they can think of that might "break" the code; if all tests run successfully, then the coding is complete. Every piece of code that is written is tested before moving on to the next feature.
- Acceptance tests verify that the requirements as understood by the programmers satisfy the customer's actual requirements.

System-wide integration testing was encouraged, initially, as a daily end-of-day activity, for early detection of incompatible interfaces, to reconnect before the separate sections diverged widely from coherent functionality. However, system-wide integration testing has been reduced, to weekly, or less often, depending on the stability of the overall interfaces in the system.

Listening

Programmers must listen to what the customers need the system to do, what "business logic" is needed. They must understand these needs well enough to give the customer feedback about the technical aspects of how the problem might be solved, or cannot be solved. Communication between the customer and programmer is further addressed in the *planning game*.

Designing

From the point of view of simplicity, of course one could say that system development doesn't need more than coding, testing and listening. If those activities are performed well, the result should always be a system that works. In practice, this will not work. One can come a long way without designing but at a given time one will get stuck. The system becomes too complex and the dependencies within the system cease to be clear. One can avoid this by creating a design structure that organizes the logic in the system. Good design will avoid lots of dependencies within a system; this means that changing one part of the system will not affect other parts of the system.

Values

Extreme programming initially recognized four values in 1999: communication, simplicity, feedback, and courage. A new value, respect, was added in the second edition of *Extreme Programming Explained*. Those five values are described below.

Communication

Building software systems requires communicating system requirements to the developers of the system. In formal software development methodologies, this task is accomplished through documentation. Extreme programming techniques can be viewed as methods for rapidly building and disseminating institutional knowledge among members of a development team. The goal is to give all developers a shared view of the system which matches the view held by the users of the system. To this end, extreme programming favors simple designs, common metaphors, collaboration of users and programmers, frequent verbal communication, and feedback.

Simplicity

Extreme programming encourages starting with the simplest solution. Extra functionality can then be added later. The difference between this approach and more conventional system development methods is the focus on designing and coding for the needs of today instead of those of tomorrow, next week, or next month. This is sometimes summed up as the "You aren't gonna need it" (YAGNI) approach.[8] Proponents of XP acknowledge the disadvantage that this can sometimes entail more effort tomorrow to change the system; their claim is that this is more than compensated for by the advantage of not investing in possible future requirements that might change before they become relevant. Coding and designing for uncertain future requirements implies the risk of spending resources on something that might not be needed, while perhaps delaying crucial features. Related to the "communication" value, simplicity in design and coding should improve the quality of communication. A simple design with very simple code could be easily understood by most programmers in the team.

Feedback

Within extreme programming, feedback relates to different dimensions of the system development:

- Feedback from the system: by writing unit tests,[5] or running periodic integration tests, the programmers have direct feedback from the state of the system after implementing changes.
- Feedback from the customer: The functional tests (aka acceptance tests) are written by the customer and the testers. They will get concrete feedback about the current state of their system. This review is planned once in every two or three weeks so the customer can easily steer the development.
- Feedback from the team: When customers come up with new requirements in the planning game the team directly gives an estimation of the time that it will take to implement.

Feedback is closely related to communication and simplicity. Flaws in the system are easily communicated by writing a unit test that proves a certain piece of code will break. The direct feedback from the system tells programmers to recode this part. A customer is able to test the system periodically according to the functional requirements, known as *user stories*.[5] To quote Kent Beck, "Optimism is an occupational hazard of programming. Feedback is the treatment."[9]

Courage

Several practices embody courage. One is the commandment to always design and code for today and not for tomorrow. This is an effort to avoid getting bogged down in design and requiring a lot of effort to implement anything else. Courage enables developers to feel comfortable with refactoring their code when necessary.[5] This means reviewing the existing system and modifying it so that future changes can be implemented more easily. Another example of courage is knowing when to throw code away: courage to remove source code that is obsolete, no matter how much effort was used to create that source code. Also, courage means persistence: A programmer might be stuck on a complex problem for an entire day, then solve the problem quickly the next day, but only if they are persistent.

Respect

The respect value includes respect for others as well as self-respect. Programmers should never commit changes that break compilation, that make existing unit-tests fail, or that otherwise delay the work of their peers. Members respect their own work by always striving for high quality and seeking for the best design for the solution at hand through refactoring.

Adopting the four earlier values leads to respect gained from others in the team. Nobody on the team should feel unappreciated or ignored. This ensures a high level of motivation and encourages loyalty toward the team and toward the goal of the project. This value is very dependent upon the other values, and is very much oriented toward people in a team.

Rules

The first version of rules for XP was published in 1999 by Don Wells[10] at the XP website. 29 rules are given in the categories of planning, managing, designing, coding, and testing. Planning, managing and designing are called out explicitly to counter claims that XP doesn't support those activities.

Another version of XP rules was proposed by Ken Auer[11] in XP/Agile Universe 2003. He felt XP was defined by its rules, not its practices (which are subject to more variation and ambiguity). He defined two categories: "Rules of Engagement" which dictate the environment in which software development can take place effectively, and "Rules of Play" which define the minute-by-minute activities and rules within the framework of the Rules of Engagement.

Principles

The principles that form the basis of XP are based on the values just described and are intended to foster decisions in a system development project. The principles are intended to be more concrete than the values and more easily translated to guidance in a practical situation.

Feedback

Extreme programming sees feedback as most useful if it is done frequently and promptly. It stresses that minimal delay between an action and its feedback is critical to learning and making changes. Unlike traditional system development methods, contact with the customer occurs in more frequent iterations. The customer has clear insight into the system that is being developed, and can give feedback and steer the development as needed. With frequent feedback from the customer, a mistaken design decision made by the developer will be noticed and corrected quickly, before the developer spends much time implementing it.

Unit tests contribute to the rapid feedback principle. When writing code, running the unit test provides direct feedback as to how the system reacts to the changes made. This includes running not only the unit tests that test the developer's code, but running in addition all unit tests against all the software, using an automated process that can be initiated by a single command. That way, if the developer's changes cause a failure in some other portion of the system that the developer knows little or nothing about, the automated all-unit-test suite will reveal the failure immediately, alerting the developer of the incompatibility of his change with other parts of the system, and the necessity of removing or modifying his change. Under traditional development practices, the absence of an automated, comprehensive unit-test suite meant that such a code change, assumed harmless by the developer, would have been left in place, appearing only during integration testing – or worse, only in production; and determining which code change caused the problem, among all the changes made by all the developers during the weeks or even months previous to integration testing, was a formidable task.

Assuming simplicity

This is about treating every problem as if its solution were "extremely simple". Traditional system development methods say to plan for the future and to code for reusability. Extreme programming rejects these ideas.

The advocates of extreme programming say that making big changes all at once does not work. Extreme programming applies incremental changes: for example, a system might have small releases every three weeks. When many little steps are made, the customer has more control over the development process and the system that is being developed.

Embracing change

The principle of embracing change is about not working against changes but embracing them. For instance, if at one of the iterative meetings it appears that the customer's requirements have changed dramatically, programmers are to embrace this and plan the new requirements for the next iteration.

Practices

Extreme programming has been described as having 12 practices, grouped into four areas:

Fine-scale feedback

- Pair programming[5]
- Planning game
- Test-driven development
- Whole team

Continuous process

- Continuous integration
- Refactoring or design improvement[5]
- Small releases

Shared understanding

- Coding standards
- Collective code ownership[5]
- Simple design[5]
- System metaphor

Programmer welfare

- Sustainable pace

Coding

- The customer is always available
- Code the unit test first
- Only one pair integrates code at a time
- Leave optimization until last
- No overtime

Testing

- All code must have unit tests
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created before the bug is addressed (a bug is not an error in logic, it is a test that was not written)
- Acceptance tests are run often and the results are published

Controversial aspects

The practices in XP have been heavily debated.[5] Proponents of extreme programming claim that by having the on-site customer[5] request changes informally, the process becomes flexible, and saves the cost of formal overhead. Critics of XP claim this can lead to costly rework and project scope creep beyond what was previously agreed or funded.

Change-control boards are a sign that there are potential conflicts in project objectives and constraints between multiple users. XP's expedited methods are somewhat dependent on programmers being able to assume a unified client viewpoint so the programmer can concentrate on coding, rather than documentation of compromise objectives and constraints. This also applies when multiple programming organizations are involved, particularly organizations which compete for shares of projects.[*citation needed*]

Other potentially controversial aspects of extreme programming include:

- Requirements are expressed as automated acceptance tests rather than specification documents.
- Requirements are defined incrementally, rather than trying to get them all in advance.
- Software developers are usually required to work in pairs.
- There is no Big Design Up Front. Most of the design activity takes place on the fly and incrementally, starting with "the simplest thing that could possibly work" and adding complexity only when it's required by failing tests. Critics compare this to "debugging a system into appearance" and fear this will result in more re-design effort than only re-designing when requirements change.
- A customer representative is attached to the project. This role can become a single-point-of-failure for the project, and some people have found it to be a source of stress. Also, there is the danger of micro-management by a non-technical representative trying to dictate the use of technical software features and architecture.
- Dependence upon all other aspects of XP: "XP is like a ring of poisonous snakes, daisy-chained together. All it takes is for one of them to wriggle loose, and you've got a very angry, poisonous snake heading your way."[12]

Scalability

Historically, XP only works on teams of twelve or fewer people. One way to circumvent this limitation is to break up the project into smaller pieces and the team into smaller groups. It has been claimed that XP has been used successfully on teams of over a hundred developers.[*citation needed*] ThoughtWorks has claimed reasonable success on distributed XP projects with up to sixty people.[*citation needed*]

In 2004, industrial extreme programming (IXP)[13] was introduced as an evolution of XP. It is intended to bring the ability to work in large and distributed teams. It now has 23 practices and flexible values. As it is a new member of the Agile family, there is not enough data to prove its usability; however it claims to be an answer to what it sees as XP's imperfections.

Severability and responses

In 2003, Matt Stephens and Doug Rosenberg published *Extreme Programming Refactored: The Case Against XP*, which questioned the value of the XP process and suggested ways in which it could be improved.[6] This triggered a lengthy debate in articles, Internet newsgroups, and web-site chat areas. The core argument of the book is that XP's practices are interdependent but that few practical organizations are willing/able to adopt all the practices; therefore the entire process fails. The book also makes other criticisms, and it draws a likeness of XP's "collective ownership" model to socialism in a negative manner.

Certain aspects of XP have changed since the publication of *Extreme Programming Refactored*; in particular, XP now accommodates modifications to the practices as long as the required objectives are still met. XP also uses increasingly generic terms for processes. Some argue that these changes invalidate previous criticisms; others claim that this is simply watering the process down.

Other authors have tried to reconcile XP with the older methodologies in order to form a unified methodology. Some of these XP sought to replace, such as the waterfall methodology; example: Project Lifecycles: Waterfall, Rapid Application Development, and All That. JPMorgan Chase & Co. tried combining XP with the computer programming methods of capability maturity model integration (CMMI), and Six Sigma. They found that the three systems reinforced each other well, leading to better development, and did not mutually contradict.[14]

Criticism

Extreme programming's initial buzz and controversial tenets, such as pair programming and continuous design, have attracted particular criticisms, such as the ones coming from McBreen[15] and Boehm and Turner.,[16] Matt Stephens and

Doug Rosenberg.[17] Many of the criticisms, however, are believed by Agile practitioners to be misunderstandings of agile development.[18]

In particular, extreme programming has been reviewed and critiqued by Matt Stephens's and Doug Rosenberg's *Extreme Programming Refactored*.[6][19]

Criticisms include:

- a methodology is only as effective as the people involved, Agile does not solve this
- often used as a means to bleed money from customers through lack of defining a deliverable product
- lack of structure and necessary documentation
- only works with senior-level developers
- incorporates insufficient software design
- requires meetings at frequent intervals at enormous expense to customers
- requires too much cultural change to adopt
- can lead to more difficult contractual negotiations
- can be very inefficient; if the requirements for one area of code change through various iterations, the same programming may need to be done several times over. Whereas if a plan were there to be followed, a single area of code is expected to be written once.
- impossible to develop realistic estimates of work effort needed to provide a quote, because at the beginning of the project no one knows the entire scope/requirements
- can increase the risk of scope creep due to the lack of detailed requirements documentation
- Agile is feature-driven; non-functional quality attributes are hard to be placed as user stories.