2. Develop an acceptance plan:
   - Project description.
   - User responsibilities
   - Acceptance description.
   - Execute the acceptance test plan.

Agile development

as a basis but advocates a lighter and more people-centric viewpoint than traditional approaches. Agile processes use feedback, rather than planning, as their primary control mechanism. The feedback is driven by regular tests and releases of the evolving software.

There are many variations of agile processes:

- In Extreme Programming (XP), the phases are carried out in extremely small (or "continuous") steps compared to the older, "batch" processes. The (intentionally incomplete) first pass through the steps might take a day or a week, rather than the months or years of each complete step in the Waterfall model. First, one writes automated tests, to provide concrete goals for development. Next is coding (by a pair of programmers), which is complete when all the tests pass, and the programmers can't think of any more tests that are needed. Design and architecture emerge out of refactoring, and come after coding. Design is done by the same people who do the coding. (Only the last feature — merging design and code — is common to *all* the other agile processes.) The incomplete but functional system is deployed or demonstrated for (some subset of) the users (at least one of which is on the development team). At this point, the practitioners start again on writing tests for the next most important part of the system.

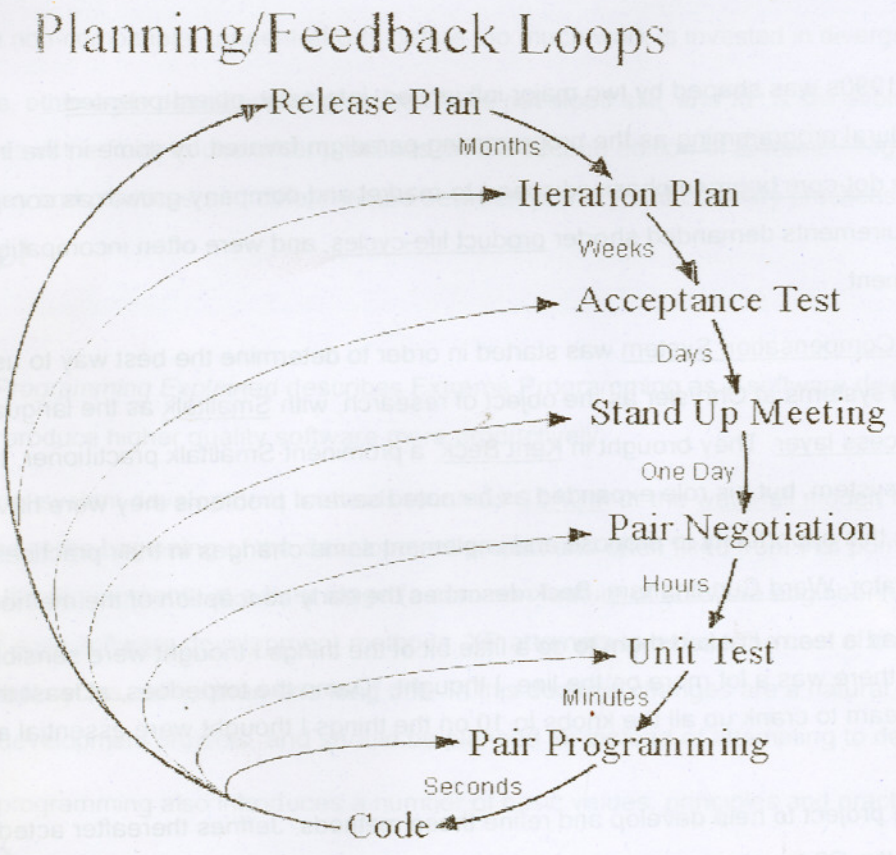
- Scrum

Extreme Programming

Extreme Programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type ofagile software development, it advocates frequent "releases" in short development cycles (timeboxing), which is intended to improve productivity and introduce checkpoints where 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. The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels, on the theory that if some is good, more is better. It is unrelated to "cowboy coding", which is more free-form and unplanned. It does not advocate "death march" work schedules, but instead working at a sustainable pace.

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



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 method used in the project and wrote a book on the method (in October 1999, *Extreme Programming Explained* was published).[6] Chrysler cancelled the C3 project in February 2000, after the company was acquired by Daimler-Benz.[7] 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 1960 (Larman 2003). To shorten the total development time, some formal test documents (such as for acceptance testing) hav 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. Some other XP practices, such as refactoring, modularity, bottom-up design, and incremental design were described by Leo Brodie in his book published in 1984.

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 boomemphasized 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 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, 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. Beck describes the early conception of the methods:

The first time I was asked to lead a team, I asked them to do a little bit of the things I thought were sensible, like testing and reviews. The second time there was a lot more on the line. I thought, "Damn the torpedoes, at least this will make a good article," [and] asked the team to crank up all the knobs to 10 on the things I thought were essential and leave out everything else.

Beck invited Ron Jeffries to the project to help develop and refine these methods. Jeffries thereafter acted as a coach to instill the practices as habits in the C3 team.

Information about the principles and practices behind XP was disseminated to the wider world through discussions on the original Wiki, Cunningham's WikiWikiWeb. Various contributors discussed and expanded upon the ideas, and some spin-off methodologies resulted (seeagile software development). Also, XP concepts have been explained, for several years, using a hypertext system map on the XP website at "http://www.extremeprogramming.org" circa 1999.

Beck edited a series of books on XP, beginning with his own *Extreme Programming Explained* (1999, ISBN 0-201-61641-6), spreading his ideas to a much larger, yet very receptive, audience. Authors in the series went through various aspects attending XP and its practices. Even a book was written, critical of the practices.

Current state

XP created quite a buzz 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 a: those thought too rigid, to be deprecated or reduced, or even left unfinished, on individual sites. For example, the practic 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 jus 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 non-compatible, 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 fron experiences in the field, to use other practices. In the second edition of *Extreme Programming Explained*, 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.

In traditional system development methods (such as SSADM or the waterfall model) the requirements for the system are determined at the beginning of the development project and often fixed from that point on. This means that the cost of changing the requirements at a later stage (a common feature of software engineering projects[citation needed]) will be high. Like other agile software development methods, XP attempts to reduce the cost of change by having multiple short development cycles, rather than one long one. In this doctrine changes are a natural, inescapable and desirable aspect c 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 programmin 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 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 and finding it hard to explain the solution to fellow programmers might code it 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

One can not be certain that a function works unless one tests it. Bugs and design errors are pervasive problems in software development. 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. These occur in the exploration phase of release planning.

A "testathon" is an event when programmers meet to do collaborative test writing, a kind of brainstorming relative to software testing.

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. A new value was added in the second edition of *Extreme Programming Explained*. The five values are:

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 ain't gonna need it" (YAGNI) approach. 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. 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*. To quote Kent Beck, "Optimism is an occupational hazard of programming, feedback is the treatment."

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. 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, if only 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 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 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.

In the APSO workshop at ICSE 2008 Conference, Mehdi Mirakhorli proposed a new and more precise and comprehensive version of the Extreme Programming Rules, more independent of the practices, and intended to be more "agile".

Rules of engagement

According to Mehdi Mirakhorli, these are:[citation needed]

- Business people and developers do joint work: Business people and developers must work together daily throughout the project.
- Our highest priority is customer satisfaction: The customer must set and continuously adjust the objectives and priorities based on estimates and other information provided by the developers or other members of the team. Objectives are defined in terms of what not how.
- Deliver working software frequently: Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale (timeboxing).
- Working software: Working software is the primary measure of progress.

- Global awareness: At any point, any member of the team must be able to measure the team's progress towards the customer's objectives and the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.
- The team must act as an effective social network, which means:
  - Honest communication leading to continuous learning and an emphasis on person-to-person interaction, rather than documentation
  - Minimal degrees of separation from what is needed by the team to make progress and the people/resources that can meet those needs.
  - Alignment of authority and responsibility.

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 rapidly and expresses that the time 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. He or she can give feedback and steer the development as needed.

Unit tests also contribute to the rapid feedback principle. When writing code, the unit test provides direct feedback as to how the system reacts to the changes one has made. If, for instance, the changes affect a part of the system that is not in the scope of the programmer who made them, that programmer will not notice the flaw. There is a large chance that this bug will appear when the system is in production.

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[6]
- Planning game
- Test-driven development
- Whole team

Continuous process

- Continuous integration
- Refactoring or design improvement[6]
- Small releases

Shared understanding

- Coding standards
- Collective code ownership[6]
- Simple design[6]
- 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 till 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 you forgot to write)
- Acceptance tests are run often and the results are published

Extreme programming practices

Extreme programming (XP) is a popular agile software development methodology used to implement software projects. This article details the practices used in this methodology. Extreme programming has 12 practices, grouped into four areas, derived from the best practices of software engineering.

Fine scale feedback

Pair programming

Pair programming means that all code is produced by two people programming on one task on one workstation. One programmer has control over the workstation and is thinking mostly about the coding in detail. The other programmer is more focused on the big picture, and is continually reviewing the code that is being produced by the first programmer. Programmers trade roles regularly.

The pairs are not fixed: it's recommended that programmers try to mix as much as possible, so that everyone knows what everyone is doing, and everybody can become familiar with the whole system. This way, pair programming also can enhance team-wide communication. (This also goes hand-in-hand with the concept of Collective Ownership).

Planning game

The main planning process within extreme programming is called the Planning Game. The game is a meeting that occurs once per iteration, typically once a week. The planning process is divided into two parts:

- Release Planning: This is focused on determining what requirements are included in which near-term releases, and when they should be delivered. The customers and developers are both part of this. Release Planning consists of three phases:
  - Exploration Phase: In this phase the customer will provide a shortlist of high-value requirements for the system. These will be written down on user story cards.
  - Commitment Phase: Within the commitment phase business and developers will commit themselves to the functionality that will be included and the date of the next release.
  - Steering Phase: In the steering phase the plan can be adjusted, new requirements can be added and/or existing requirements can be changed or removed.

- Iteration Planning: This plans the activities and tasks of the developers. In this process the customer is not involved. Iteration Planning also consists of three phases:
  - Exploration Phase: Within this phase the requirement will be translated to different tasks. The tasks are recorded on task cards.
  - Commitment Phase: The tasks will be assigned to the programmers and the time it takes to complete will be estimated.
  - Steering Phase: The tasks are performed and the end result is matched with the original user story.

The purpose of the Planning Game is to guide the product into delivery. Instead of predicting the exact dates of when deliverables will be needed and produced, which is difficult to do, it aims to "steer the project" into delivery using a straightforward approach.[2]

Release planning

Exploration phase

This is an iterative process of gathering requirements and estimating the work impact of each of those requirements.

- Write a Story: Business has come with a problem; during a meeting, development will try to define this problem and get requirements. Based on the business problem, a story (user story) has to be written. This is done by business, where they point out what they want a part of the system to do. It is important that development has no influence on this story. The story is written on a user story card.
- Estimate a Story: Development estimates how long it will take to implement the work implied by the story card. Development can also create spike solutions to analyze or solve the problem. These solutions are used for estimation and discarded once everyone gets clear visualization of the problem. Again, this may not influence the business requirements.
- Split a Story: Every design critical complexity has to be addressed before starting the iteration planning. If development isn't able to estimate the story, it needs to be split up and written again.

When business cannot come up with any more requirements, one proceeds to the commitment phase.

Commitment phase

This phase involves the determination of costs, benefits, and schedule impact. It has four components:

- Sort by Value: Business sorts the user stories by Business Value.
- Sort by Risk: Development sorts the stories by risk.
- Set Velocity: Development determines at what speed they can perform the project.
- Choose scope: The user stories that will be finished in the next release will be picked. Based on the user stories the release date is determined.

Sort by value

The business side sorts the user stories by business value. They will arrange them into three piles:

- Critical: stories without which the system cannot function or has no meaning.
- Significant Business Value: Non-critical user stories that have significant business value.
- Nice to have: User stories that do not have significant business value.

Sort by risk

The developers sort the user stories by risk. They also categorize into three piles: low, medium and high risk user stories. The following is an example of an approach to this:

- Determine Risk Index: Give each user story an index from 0 to 2 on each of the following factors:
  - Completeness (do we know all of the story details?)
    - Complete (0)
    - Incomplete (1)
    - Unknown (2)
  - Volatility (is it likely to change?)
    - low (0)
    - medium (1)
    - high (2)
  - Complexity (how hard is it to build?)
    - simple (0)
    - standard (1)
    - complex (2)

All indexes for a user story are added, assigning the user stories a risk index of low (0–1), medium (2–4), or high (5–6).

Steering phase

Within the steering phase the programmers and business people can "steer" the process. That is to say, they can make changes. Individual user stories, or relative priorities of different user stories, might change; estimates might prove wrong. This is the chance to adjust the plan accordingly.

Iteration planning

Exploration phase

The exploration phase of the iteration planning is about creating tasks and estimating their implementation time.

- Translate the requirement to tasks: Place on task cards.
- Combine/Split task: If the programmer cannot estimate the task because it is too small or too big, the programmer will need to combine or split the task.
- Estimate task: Estimate the time it will take to implement the task.

Commitment phase

Within the commitment phase of the iteration planning programmers are assigned tasks that reference the different user stories.

- A programmer accepts a task: Each programmer picks a task for which he or she takes responsibility.
- Programmer estimates the task: Because the programmer is now responsible for the task, he or she should give the eventual estimation of the task.
- Set load factor: The load factor represents the ideal amount of hands-on development time per programmer within one iteration. For example, in a 40-hour week, with 5 hours dedicated to meetings, this would be no more than 35 hours.
- Balancing: When all programmers within the team have been assigned tasks, a comparison is made between the estimated time of the tasks and the load factor. Then the tasks are balanced out among the programmers. If a programmer is overcommitted, other programmers must take over some of his or her tasks and vice versa.

Steering phase

The implementation of the tasks is done during the steering phase of the iteration planning.

- Get a task card: The programmer gets the task card for one of the tasks to which he or she has committed.
- Find a Partner: The programmer will implement this task along with another programmer. This is further discussed in the practice Pair Programming.
- Design the task: If needed, the programmers will design the functionality of the task.
- Write unit test: Before the programmers start coding the functionality they first write automated tests. This is further discussed in the practice Unit Testing.
- Write code: The programmers start to code.
- Run test: The unit tests are run to test the code.
- Refactor: Remove any code smells from the code.
- Run Functional test: Functional tests (based on the requirements in the associated user story and task card) are run.

Test driven development

Unit tests are automated tests that test the functionality of pieces of the code (e.g. classes, methods). Within XP, unit tests are written before the eventual code is coded. This approach is intended to stimulate the programmer to think about conditions in which his or her code could fail. XP says that the programmer is finished with a certain piece of code when he or she cannot come up with any further condition on which the code may fail.

Whole team

Within XP, the "customer" is not the one who pays the bill, but the one who really uses the system. XP says that the customer should be on hand at all times and available for questions. For instance, the team developing a financial administration system should include a financial administrator.

Continuous process

Continuous integration

The development team should always be working on the latest version of the software. Since different team members may have versions saved locally with various changes and improvements, they should try to upload their current version to the code repository every few hours, or when a significant break presents itself. Continuous integration will avoid delays later on in the project cycle, caused by integration problems.

Design improvement

Because XP doctrine advocates programming only what is needed today, and implementing it as simply as possible, at times this may result in a system that is stuck. One of the symptoms of this is the need for dual (or multiple) maintenance: functional changes start requiring changes to multiple copies of the same (or similar) code. Another symptom is that changes in one part of the code affect lots of other parts. XP doctrine says that when this occurs, the system is telling you to refactor your code by changing the architecture, making it simpler and more generic.

Small releases

The delivery of the software is done via frequent releases of live functionality creating concrete value. The small releases help the customer to gain confidence in the progress of the project. This helps maintain the concept of the whole team as the customer can now come up with his suggestions on the project based on real experience.

Shared understanding

Coding standard

Coding standard is an agreed upon set of rules that the entire development team agree to adhere to throughout the project. The standard specifies a consistent style and format for source code, within the chosen programming language, as well as various programming constructs and patterns that should be avoided in order to reduce the probability of defects. The coding standard may be a standard conventions specified by the language vendor (e.g. The Code Conventions for the Java Programming Language, recommended by Sun), or custom defined by the development team.

Collective code ownership

Collective code ownership means that everyone is responsible for all the code; this, in turn, means that everybody is allowed to change any part of the code. Pair programming contributes to this practice: by working in different pairs, all the programmers get to see all the parts of the code. A major advantage claimed for collective ownership is that it speeds up the development process, because if an error occurs in the code any programmer may fix it.

By giving every programmer the right to change the code, there is risk of errors being introduced by programmers who think they know what they are doing, but do not foresee certain dependencies. Sufficiently well defined unit tests address this problem: if unforeseen dependencies create errors, then when unit tests are run, they will show failures.

Simple design

Programmers should take a "simple is best" approach to software design. Whenever a new piece of code is written, the author should ask themselves 'is there a simpler way to introduce the same functionality?'. If the answer is yes, the simpler course should be chosen. Refactoring should also be used, to make complex code simpler.

System metaphor

The system metaphor is a story that everyone - customers, programmers, and managers - can tell about how the system works. It's a naming concept for classes and methods that should make it easy for a team member to guess the functionality of a particular class/method, from its name only. For example a library system may create loan_records(class) for borrowers(class), and if the item were to become overdue it may perform a make_overdue operation on a catalogue (class). For each class or operation the functionality is obvious to the entire team.

Programmer welfare

Sustainable pace

The concept is that programmers or software developers should not work more than 40 hour weeks, and if there is overtime one week, that the next week should not include more overtime. Since the development cycles are short cycles of continuous integration, and full development (release) cycles are more frequent, the projects in XP do not follow the typical crunch time that other projects require (requiring overtime).

Also, included in this concept is that people perform best and most creatively if they are rested.

A key enabler to achieve sustainable pace is frequent code-merge and always executable & test covered high quality code. The constant refactoring way of working enforces team members with fresh and alert minds. The intense collaborative way of working within the team drives a need to recharge over weekends.

Well-tested, continuously integrated, frequently deployed code and environments also minimize the frequency of unexpected production problems and outages, and the associated after-hours nights and weekends work that is required.

Controversial aspects

The practices in XP have been heavily debated. Proponents of extreme programming claim that by having the on-site customer 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 methodology is 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.

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."

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. ThoughtWorks has claimed reasonable success on distributed XP projects with up to sixty people.

In 2004 Industrial Extreme Programming (IXP) 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. 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 book *Extreme Programming Refactored* (2003) was published; 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.

RDP Practice is a technique for tailoring extreme programming. This practice was initially proposed as a long research paper in a workshop organized by Philippe Kruchten and Steve Adolph(See APSO workshop at ICSE 2008) and yet it is the only proposed and applicable method for customizing XP. The valuable concepts behind RDP practice, in a short time provided the rationale for applicability of it in industries. RDP Practice tries to customize XP by relying on technique XP Rules.

Other authors have tried to reconcile XP with the older methods in order to form a unified methodology. Some of these XP sought to replace, such as the waterfall method; example: Project Lifecycles: Waterfall, Rapid Application Development, and All That. JPMorgan Chase & Co tried combining XP with the computer programming methodologies 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.

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 and Boehm and Turner. Many of the criticisms, however, are believed by Agile practitioners to be misunderstandings of agile development.

In particular, extreme programming is reviewed and critiqued by Matt Stephens's and Doug Rosenberg's *Extreme Programming Refactored*.

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
- 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

References

1. ^ "Human Centred Technology Workshop 2005", 2005, PDF webpage: Informatics-UK-report-cdrp585.
2. ^ a b "Design Patterns and Refactoring", University of Pennsylvania, 2003, webpage: UPenn-Lectures-design-patterns.
3. ^ a b "Extreme Programming" (lecture paper), USFCA.edu, webpage: USFCA-edu-601-lecture.
4. ^ "Manifesto for Agile Software Development", Agile Alliance, 2001, webpage: Manifesto-for-Agile-Software-Dev
5. ^ a b "Everyone's a Programmer" by Clair Tristram. *Technology Review*, Nov 2003. p. 39
6. ^ a b c d e f g h i j k l m "Extreme Programming", *Computerworld* (online), December 2001, webpage: Computerworld-appdev-92.
7. ^ *Extreme Programming Refactored: The Case Against XP*. ISBN 1590590961.