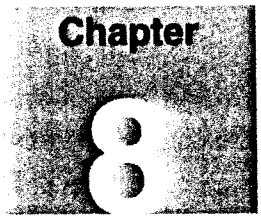



# EXTREME PROGRAMMING

*It's easy to have a complicated idea.*
*It's very very hard to have a simple idea.*
*—Carver Mead*

## OVERVIEW

- Classification of XP.
- Workproducts, roles, and practices.
- Common mistakes, adoption and process mixtures, strengths and weaknesses.

**Extreme Programming** (**XP**) is a well-known agile method; it emphasizes collaboration, quick and early software creation, and skillful development practices. It is founded on four values: communication, simplicity, feedback, and courage. In addition to IID, it recommends 12 core practices:

1. *Planning Game*
2. *small, frequent releases*
3. *system metaphors*
4. *simple design*
5. *testing*
6. *frequent refactoring*
7. *pair programming*
8. *team code ownership*
9. *continuous integration*
10. *sustainable pace*
11. *whole team together*
12. *coding standards*

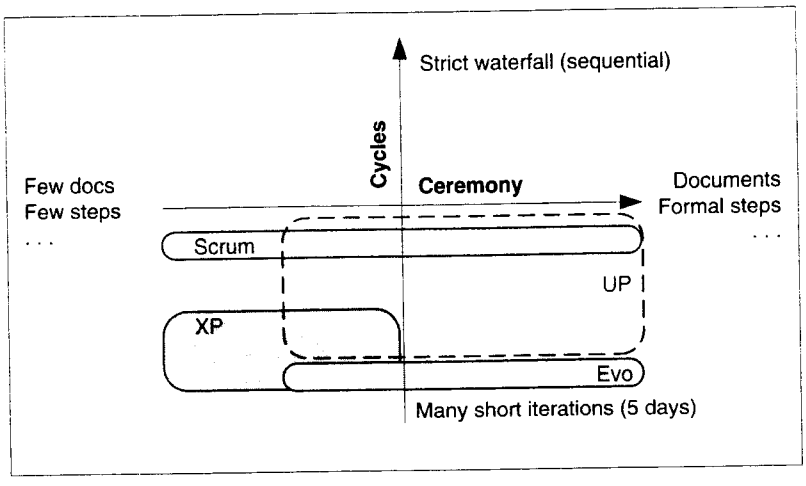
## METHOD OVERVIEW

### Classification

*cycles and ceremony p. 26*

In terms of cycles and ceremony, XP classification is illustrated in Figure 8.1. For average projects, the recommended length of a timeboxed iteration is between one and three weeks—somewhat shorter than for UP or Scrum.

Figure 8.1 XP on the cycle and ceremony scale.




XP is low on the ceremony scale; it has only a small set of a predefined, informal workproducts, such as paper index cards for summarizing feature requests, called **story cards**.

A refreshing quality of the original XP description was the statement of known applicability: It had been proven on projects involving roughly 10 developers or fewer, and not proven for safety-critical systems. Nevertheless, it has been more recently applied with larger teams. Consequently, in terms of the Cockburn scale, XP perhaps covers the cells shown in Figure 8.2.

*Cockburn scale p. 36*

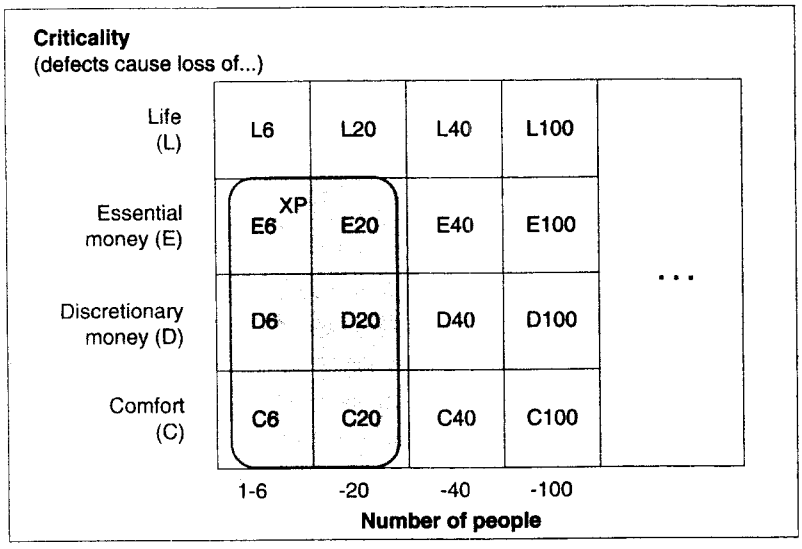



Figure 8.2 XP on the Cockburn scale

## Introduction

XP[1] [Beck00], created by Kent Beck, is an IID method that stresses customer satisfaction through rapid creation of high-value software, skillful and sustainable software development techniques, and flexible response to change. It is aimed at relatively small team projects, usually with delivery dates under one year. Iterations are short—usually one to three weeks.

As the word *programming* suggests, it provides explicit methods for programmers, so they can more confidently respond to changing requirements, even late in the project, and still produce quality code. These include test-driven development, refactoring, pair programming, and continuous integration, among others. In contrast to most methods, some XP practices truly are adopted by

1. Some writers capitalize the full name as "eXtreme Programming" but Beck does not.

developers—they sense its practical programmer-relevant techniques.

XP is very communication- and team-oriented; customers, developers, and managers form a team working in a common project room to quickly deliver software with high business value.

XP is distinctive in not *requiring* detailed workproducts except for program code and tests. However, it doesn't disallow other detailed workproducts. Although all evolutionary methods avoid detailed up-front specifications and plans that span the entire release cycle, most of these methods encourage writing down details for at least the next iteration. In contrast, XP emphasizes *oral communication* for requirements and design. For example, a feature is summarized "Find lowest fare" on a handwritten paper index **story card**. Then, when work starts on the feature, the programmers learn details by talking with the customers working full-time in the project room. This may sound disorganized or naive, but Beck is experienced and well aware of the implications of sloppy requirements. Instead, XP is posing this interesting question:

> *Is there a sane and disciplined way to quickly succeed—on typical small projects—by focusing on code and tests, while avoiding most other documentation overhead?*

XP's premise isn't hacker code-and-fix programming; rather, its premise is that there is a new, structured, and sustainable way to succeed with a focus on rapid code production and oral communication, while avoiding most other overhead. To reiterate, XP is not hacking. Quite the contrary, an XP project involves constant practice of highly disciplined—yet agile—software development practices and values.

XP consultant Don Wells explains the influence of the XP values [Wells01]:

*XP improves a software project in four essential ways; communication, simplicity, feedback, and courage. XP programmers communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software starting on day one. They deliver the system to the customers as early as possible and implement changes as suggested. With this foundation XP programmers are able to courageously respond to changing requirements and technology.*

There is a considerable set of practices in XP: 12 core practices and many ancillary ones. Speaking of these, Wells writes [Wells01]:

*It is a lot like a jig saw puzzle. There are many small pieces. Individually the pieces make no sense, but when combined together a complete picture can be seen.*

Many of these practices work *in synergy*, and thus *it is risky to customize XP by removing some elements.* For example, XP aims to produce software quickly by—in part—avoiding detailed requirements documentation. But, this is compensated by the practice of onsite customers sitting in the project room to fill in the details.

The word *extreme* in XP comes from Beck's conviction that it is hard to have too much of a good thing in software development. That is, take known good practices and "turn the dial up to 10," or to extreme levels. For example:

- Testing is good, so write unit tests for (almost) all code, and acceptance tests for all features.
- Code reviews are good—even better close to creation date—so do code reviews in real time and all the time via pair programming.
- Frequent integration of code across all team members is good, so do it 24/7 with an automated, continuous integration process on a dedicated build machine.

- ❑ Short iterations and early feedback are good, so make iterations one or two weeks long, if possible.
- ❑ More customer involvement is good, so bring customers into the project full-time, sitting in the common project room.
- ❑ Communication is good, so have everyone sit together, pair program, include onsite customers, and involve the customer frequently in planning, steering, and evaluation.

## LIFECYCLE

| EXPLORATION | PLANNING | ITERATIONS TO FIRST RELEASE | PRODUCTIONIZING | MAINTENANCE |
|---|---|---|---|---|
| **Purpose:**<br>- Enough well-estimated story cards for first release.<br>- Feasibility ensured. | **Purpose:**<br>- Agree on date and stories for first release. | **Purpose:**<br>- Implement a tested system ready for release. | **Purpose:**<br>- Operational deployment | **Purpose:**<br>- Enhance, fix.<br>- Build major releases |
| **Activities:**<br>- prototypes<br>- exploratory proof of technology programming<br>- story card writing and estimating | **Activities:**<br>- Release Planning Game<br>- story card writing and estimating | **Activities:**<br>- testing and programming<br>- Iteration Planning Game<br>- task writing and estimating | **Activities:**<br>- documentation<br>- training<br>- marketing<br>- . . . | **Activities:**<br>- May include these phases again, for incremental releases. |

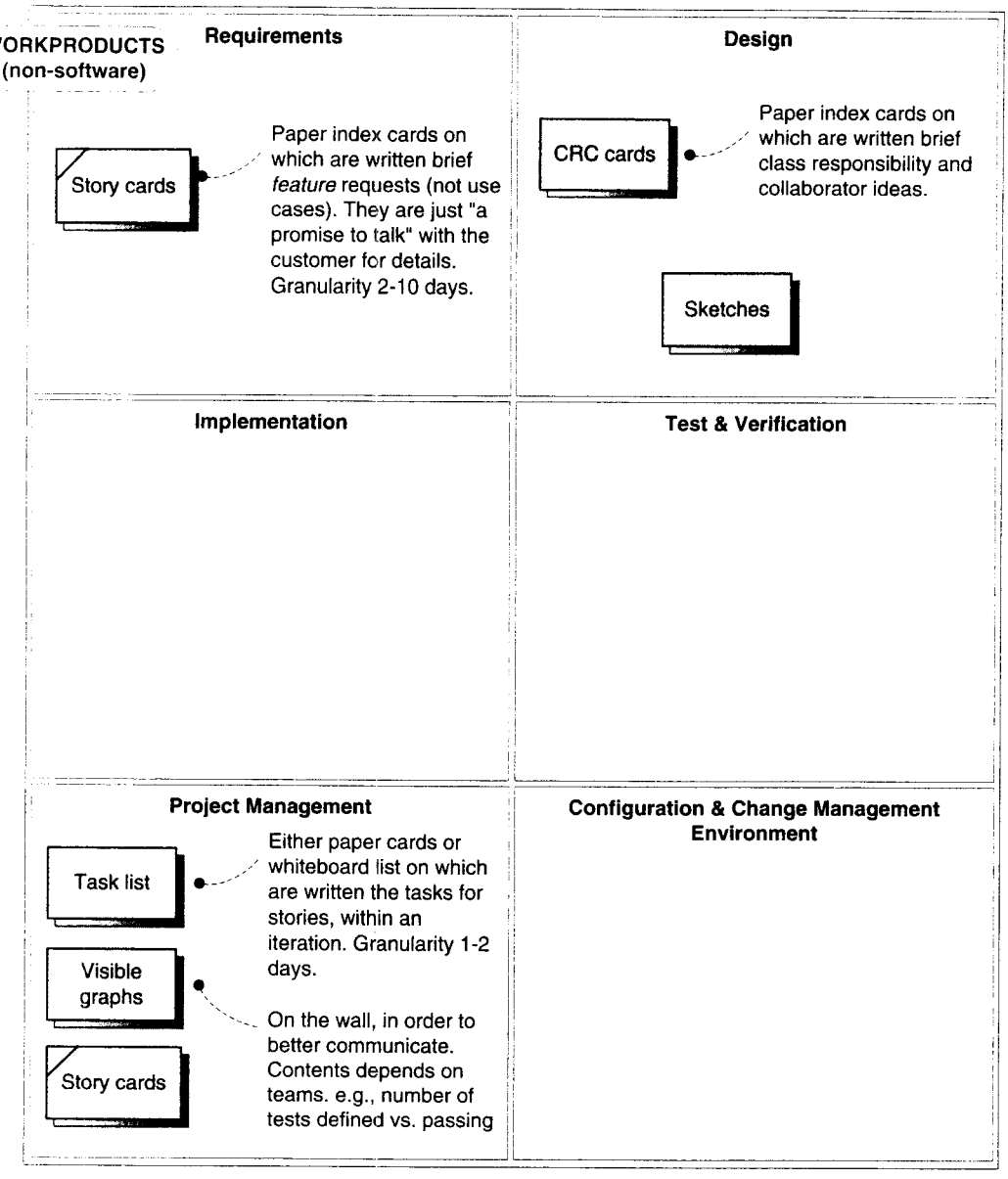
Some comments on the XP lifecycle phases defined by Beck:

1. Like many projects, XP can start with exploration. Some story cards (features) may be written, with rough estimates.
2. In the Release Planning Game, the customers and developers complete the story cards and rough estimates, and then decide what to do for the next release.
3. For the next iteration, in the Iteration Planning Game, customers pick stories to implement. They choose stories—and

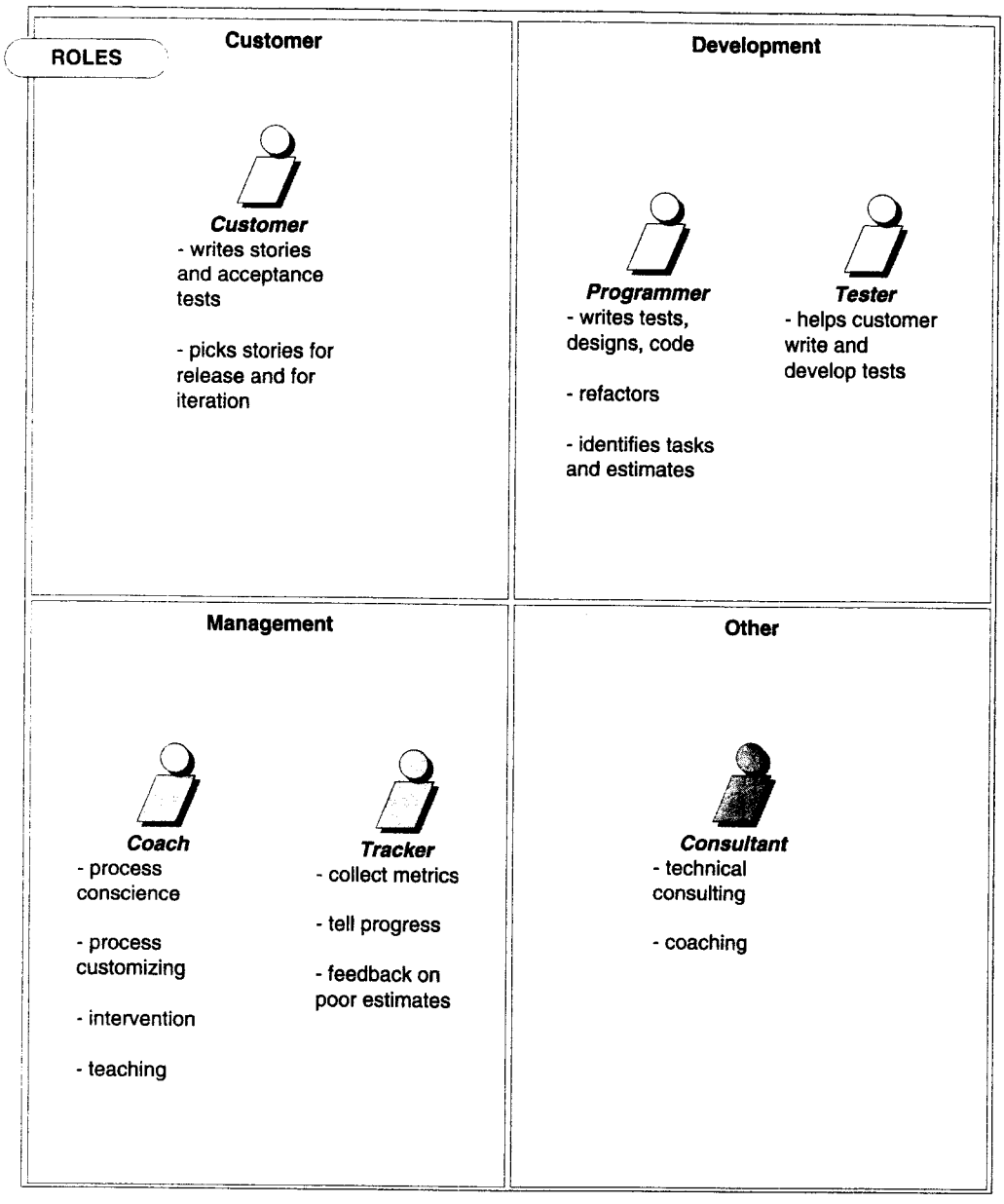
thus steer the project—based on current status, and their latest priorities for the release. Developers then break the stories into many short, estimated tasks. Finally, a review of the total estimated task-level effort may lead to readjustment of the chosen stories, as XP does not allow overworking the developers with more than they can do based on "family-friendly" work days, such as an eight-hour day. Overtime is seriously discouraged in XP; it is viewed as a sign of a dysfunctional project, increasingly unhappy people, and dropping productivity and quality.

4. Developers implement the stories within the agreed time-boxed period, continually collaborating with customers (in the common project room) on tests and requirement details.
5. If not finished for release, return to step 3 for the next iteration.

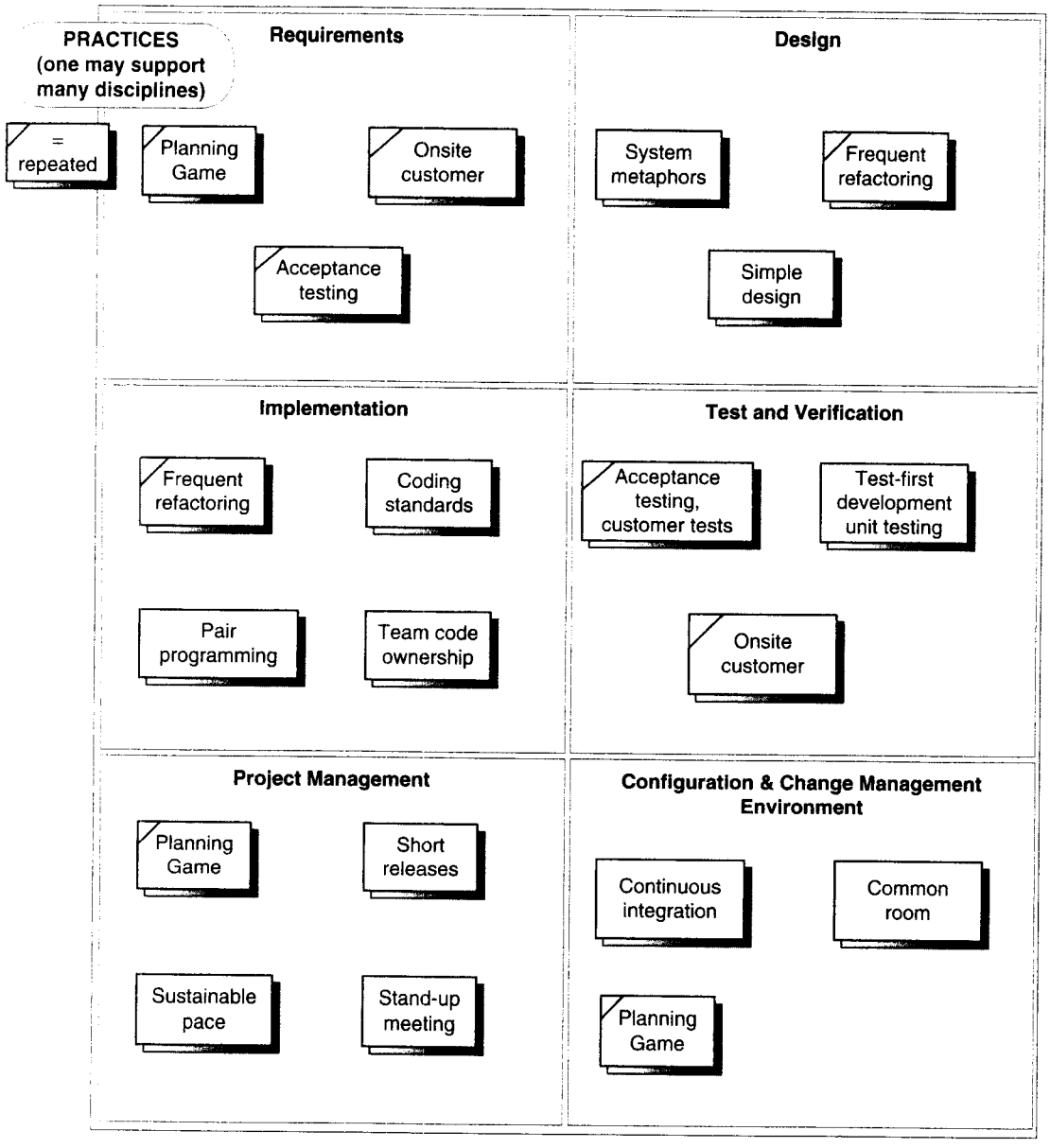
## WORKPRODUCTS, ROLES, AND PRACTICES



## Roles


ROLES
Customer
Customer
- writes stories and acceptance tests
- picks stories for release and for iteration
Development
Programmer
- writes tests, designs, code
- refactors
- identifies tasks and estimates
Tester
- helps customer write and develop tests
Management
Coach
- process conscience
- process customizing
- intervention
- teaching
Tracker
- collect metrics
- tell progress
- feedback on poor estimates
Other
Consultant
- technical consulting
- coaching

## Practices


PRACTICES
(one may support many disciplines)
= repeated
Requirements
Planning Game
Onsite customer
Acceptance testing
Design
System metaphors
Frequent refactoring
Simple design
Implementation
Frequent refactoring
Coding standards
Pair programming
Team code ownership
Test and Verification
Acceptance testing, customer tests
Test-first development unit testing
Onsite customer
Project Management
Planning Game
Short releases
Sustainable pace
Stand-up meeting
Configuration & Change Management Environment
Continuous integration
Common room
Planning Game

**Core Practices**

| Practice | Description |
|---|---|
| Whole team, or Onsite customers | The whole team—programmers and customers—work together in a common project room. One or more customers sit more-or-less full time with the team; they are expected to be subject matter experts and empowered to make decisions regarding requirements and their priority.<br><br>The customer contribution includes detailed explanation—to the programmers—of the features briefly summarized on story cards, Planning Game participation, clarification, and writing acceptance tests in collaboration with a programmer.<br><br>The purpose is in response to the consistent project failure research indicating that more customer involvement is paramount to successful projects.<br><br>The first release of XP spoke of only one onsite customer; this has been recently revised to emphasize a group of customers. |
| Small, frequent releases | Evolutionary delivery. Not applicable to all projects. Not to be confused with organizing one release cycle into many short iterations. |
| Testing: Acceptance testing & Customer tests | Testing practices in XP are very important. All features must have **automated acceptance (functional) tests.** All tests (acceptance and unit) must run with a binary pass/fail result, so that no human inspection of individual test results is required. The acceptance tests are written with collaboration of the customer—they define a testable statement of what acceptance means. This is called **Customer Tests** in XP. |

| Practice | Description |
|---|---|
| Testing: Test-driven development and unit testing | Unit tests are written for most code, and the practice of **test-driven development** (and **test-first development**) is followed. This includes the practice that the unit test is written by the programmer *before* the code to be tested. It is a cycle of test → code, rather than code → test. Usually, the open-source XUnit testing framework family (such as JUnit) is applied (see www.junit.org). All acceptance and unit tests are automatically run repeatedly in a 24/7 continuous integration build and test cycle.<br><br>See p. 292 for a detailed example. |
| Release planning game | The **Release Planning Game** goal is to define the scope of the next operational release, with maximum value (to the customer) software. Typically a half-day one-day session, customer(s) write **story cards** to describe features, and developers estimate them. There may also exist story cards from prior exploration phase work. The customer then chooses what's in the next release by either 1) setting a date and adding cards until the estimate total matches the time available, or 2) choosing the cards and calculating the release date based on their estimates. |
| Iteration planning game | The **Iteration Planning Game** goal is to choose the stories to implement, and plan and allocate tasks for the iteration. It happens shortly before each new iteration (1–3 weeks in length). Customer(s) choose the story cards for the iteration. For each, programmers create a **task list** (on cards or whiteboard) that fulfill the stories. This is followed by a volunteering step in which the programmers choose a set of tasks. They then estimate their task lengths. If tasks are not estimated in the half-day to two-day range, they are refactored. |

| Practice | Description |
|---|---|
| Simple design | Avoid speculative design for possible future changes. Avoid creating generalized components that are not immediately required. The design should avoid duplicate code, have a relatively minimal set of classes and methods, and be easily comprehensible. |
| Pair programming | All production code is created by two programmers at one computer; they rotate using the input devices periodically. Pairs may change frequently, for different tasks. The observer is doing a real-time code review, and perhaps thinking more broadly than the typist, considering tests and so forth.<br><br>Certainly, team productivity is not simply a function of the number of hands typing—it is more nuanced. The XP claim is that the combination of cross learning, the peer pressure of more disciplined practice observance and more hours actually programming than procrastinating, defect reduction due to real-time code review, and the stamina and insight to carry on when one programmer is stuck, all add up to an overall team improvement. |
| Frequent refactoring | Refactoring in the XP context is the continual effort to simplify the fine-grained code and larger design elements, while still ensuring all tests pass. That is, cleaning the code and design, without changing functionality. There is supposed to be "lots" of refactoring in XP. This practice is also known as **continuous design improvement**.<br><br>The goal is minimal, simple, comprehensible code. It is achieved by small change steps, verifying tests after each, and ideally the use of refactoring tools, now available in some IDEs. |

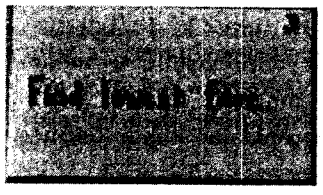
| Practice | Description |
|---|---|
| Team code ownership | Any pair of programmers can improve any code, and the XP value system is that the entire team is collectively responsible for all the code. The value of "it's her code, and her problem" is not endorsed; rather, if a problem or chance to improve is spotted, it's the spotter's responsibility. A related goal is faster development by removing the bottleneck associated with change requests in an individual code ownership model.<br><br>The obvious danger of modifying code one did not originally write is ameliorated in XP by some of the other practices: The guaranteed-present unit and acceptance tests running within an automated continuous integration build process inform you if you broke the code, your pairing partner brings another set of eyes to the change, and common adherence to coding standards ensures all the code looks the same. |
| Continuous integration | All checked-in code is continuously re-integrated and tested on a separate build machine, in an automated 24/7 process loop of compiling, running all unit tests and all or most acceptance tests. There are several open-source tools for this, built on the ubiquitous Ant technology, including CruiseControl and Anthill. |
| Sustainable pace | Frequent overtime is rightly considered a symptom of deeper problems, and doesn't lead to happy, creative developers, healthy families, or quality, maintainable code. XP doesn't support it—rather, it promotes "no overtime." |
| Coding standards | With collective code ownership, frequent refactoring, and regular swapping of pair programming partners, everyone needs to follow the same coding style. |

| Practice | Description |
|---|---|
| System metaphors | To aid design communication, capture the overall system or each subsystem with memorable metaphors to describe the key architectural themes. For example, the C3 payroll system was described in terms of an assembly line of checks with posting rule "machines" operating on them, extracting money from different "bins."<br><br>Many have reported this the least necessary practice. |

### Workproducts

**Story Cards**—Figure 8.3 shows a simple story card: A handwritten note on a paper index card. During the Planning Game, many of these are written. This spartan example was chosen to emphasize the minimalist approach to recorded requirements that XP encourages.[2]

Figure 8.3 sample story card



The story cards record **user stories**: features, fixes, or nonfunctional requirements that the user wants. There can even be a story card to create documentation. Stories are usually in the one-day to three-week range of estimated duration. Contrary to some misunderstanding, XP stories are *not* use cases or scenarios. They usually represent features. Note that XP prefers a feature-driven

2. In fact, when XP expert Ron Jeffries reviewed this chapter, he felt even the number "3" on this real-example card was excessive.

approach to describing requirements rather than the use-case-driven approach that UP promotes.

In XP, oral communication is preferred, and the story card purpose is *not* to detail the user story, but to jot a summary, make references to other documents, and in general, to view the card as "a promise to talk" (in Alistair Cockburn's words) with the customer who wrote it, by the developers implementing it. Since *whole team together* is an XP practice, the card donor should be readily available. XP coaches vary on their advice regarding granularity for estimation. Some say stories can be in the two-day to two-week range of effort, others recommend stories be estimated in units of one, two, or three weeks, but not in finer person-day units.

**Task List**—During an Iteration Planning Game, the team convenes around a whiteboard, and generates a list of tasks for all the stories chosen for the iteration. Another popular alternative is to generate individual task cards. Once a task is chosen by a volunteer, they enter an effort estimate (in *ideal engineering hours*)—tasks should be in the 1–2 day range.

**Visible Graphs**—The idea is to easily communicate—to the team—something they find useful to measure. Measure at least one thing. XP doesn't mandate what that should be, though known examples include acceptance tests defined and passing, story progress, and task progress.

### Other Practices and Values

- **Onsite customer proxies**—Many groups wishing to apply XP cannot find full-time "ultimate" customers to work in the project room. For example, consider a new internal system for (very busy) commodity traders. And this problem is common for commercial products for an external market. The common solution is to designate **customer proxies** that do join the team in the project room, have good (though not as

ideal as true customer) knowledge of the domain and requirements, and that represent the ultimate customers. If proxies are used, it is important that the true customers at least participate in end-of-iteration demos, and preferably, in the Planning Games.

- **Customer on call**—When the onsite customer is not present, arrange matters so that the customer representative is committed to fast access, such as via a mobile phone.

- **Embrace change**—The overarching attitude that XP promotes is to embrace rather than fight change, in the requirements, design, and code, and be able to move quickly in response to change.

- **Only by volunteering (accepted responsibility)**—Tasks are not assigned to people. Rather, during the Iteration Planning Game, people choose or volunteer for tasks. This leads to a higher degree of commitment and satisfaction in the self-accepted responsibility, as has been explored in [DL99].

- **Very light modeling**—XP encourages programming very early and does take to "extreme" the avoidance of up-front design work. Any more than 10 or 20 minutes of design thinking (e.g., at the whiteboard with sketches or notes) before programming is considered excessive. Contrast this with Scrum or the UP, for example, where a half-day of design thought near the start of an iteration is acceptable.

- **Minimal or "just enough" documentation**—With the goal of getting to code fast, XP discourages writing *unnecessary* requirements, design, or management documents. The use of small paper index cards is preferred for jotting brief descriptions, as are verbal communication and elaboration. Note that the practice of "avoiding documentation" is compensated by the presence of an onsite customer. XP is not anti-documentation, but notes it has a cost, perhaps better spent on programming.

- **Metrics**—XP recommends daily measurement of progress and quality. It doesn't mandate the exact metrics, but to "use

the simplest ones that could work." Examples include numbers of completed tasks and stories, and number and success rate of running tests.

- **Visible wall graphs**—The collected metrics are daily updated on wall graphs for all to easily see.
- **Tracking and Daily Tracker**—The regular collection of task and story progress metrics is the responsibility of a tracker. This is done with a walk-about to all the programmers, rather than email; commenting on this—and very telling of the XP attitude—Ron Jeffries (one of the XP founders) said, "XP is about people, not computers." Test metrics can be automatically collected by software.
- **Incremental infrastructure**—XP recommends (as do the other iterative processes) that the back-end infrastructure (for example, a persistence layer) not be the main focus of implementation in the early iterations, but rather, only enough is implemented to satisfy the user-functional requirements of each iteration.
- **Common project room**—XP projects are run in a common project room rather than separate offices. Pair programming tables are in the center of the room, and the walls are clear for whiteboard and poster work. Of course, people may have a private space for private time, but production software development is a team sport in XP.
- **Daily stand-up meeting**—As in Scrum, there is a daily short stand-up (to keep it short) meeting of status.
- **Ideal engineering hours (IEH)**—Task estimates—and possibly story estimates—are done in terms of IEH, or uninterrupted, dedicated, focused time to complete a task.
- **Story estimates**—To estimate larger stories, some XP practitioners recommend using only coarse values of one, two, or three week durations rather than IEH or day-level estimates.

## VALUES

Beck's XP description is noteworthy in the process world for being perhaps the first to explicitly state the values that underly the attitude and practices on a healthy XP project. To quote Beck on the relationship of values and practices:

> *The [practices] are what you do. The values are how you decide if you are doing it right.*

They are:

- **Communication**—XP accepts the widely appreciated observation that problems in communication underly most project difficulties. Communication between programmers is promoted through pair programming, the daily stand-up meeting, and the Planning Game. Communication is promoted through customer involvement in writing acceptance tests and the Planning Game.
- **Simplicity**—or, "Do the simplest thing that could possibly work." This applies not only to the design of software, but to other things such as requirements and project management "tools." For example, XP encourages the use of simple paper index cards to write a brief description of feature and task requests, if more formal artifacts can be avoided. In terms of software design, XP avoids speculative design for possible change ("future proofing") or the creation of more generalized components that aren't immediately justified by current requirements.
- **Feedback**—This value drives quality and adaptation. Feedback in the short term is driven by the XP practice of test-first development with unit tests. It also comes from the practice of continuous integration; a broken build tells the story. When a customer writes a story card (a feature description), programmers immediately estimate it, so the customers know the effort. The practice of daily tracker provides

feedback to the team and customer on progress for the iteration. On a longer scale, the customer written acceptance tests provide feedback. Short iterations give the customer the chance to see (and perhaps operate) an incrementally evolved partial system, and clarify or redirect the requirements. And the practice of frequent operational releases generates feedback from production users.

- **Courage**—The courage to develop fast, and make changes fast emerges from the support of the other values and practices, and modern technologies. For example, without a massive set of unit tests, acceptance tests, and continuous integration, making deep "architectural" changes in the code base is tricky business—difficult to tell what will break. But the presence of these, combined with a simple design, very clean code refined from frequent refactoring, and modern automated refactoring tools provided by many IDEs enables more rapid and radical change.

## COMMON MISTAKES AND MISUNDERSTANDINGS

### or, How to Fail with Extreme Programming

**Error: No onsite customer; rather, use specifications written for the next iteration**—It is normal and acceptable to create written specifications for the iteration (for example, two use cases) if the adopted method is Scrum, UP, Evo, etc. An iterative or agile project that takes this approach can work well, but is better characterized as being based on another method (e.g., the UP) that allows written evolutionary requirements. It is a cornerstone of XP to avoid detailed specifications, use oral communication of the requirements, and onsite customers.

**Error: Applying a subset of uncompensated practices; customizing before trying**—Many XP practices work as a synergistic whole, and it is a mistake to remove one that compensates or

supports another. For example, collective code ownership isn't feasible without testing, continuous integration, and coding standards. Minimal requirements documentation isn't possible without an onsite customer. Frequent refactoring doesn't work without the tests. That is why Beck, while not wishing to be rigid, in general recommends adoption of all or most of the core practices. To quote Beck, "Do all of XP before trying to customize it."

**Error: XP is just iterative development + minimal documentation + unit testing**—Although there is some flexibility in what practices must be present to define an XP project, it is more than this—common to several IID methods. One could run a Scrum, UP, DSDM, or other evolutionary methods primarily with just these practices. XP is characterized by a larger set of practices, including pair programming, onsite customers, customer-written acceptance tests, and more.

**Error: Not writing the unit tests first**—Test-first development has a more subtle dimension than first glance, and is an important XP practice. Writing the tests *first* influences how one conceives, clarifies, and simplifies the design. Test-first has an interesting psychological quality of satisfaction: I write the test, and then I make it succeed. There is a feeling of accomplishment that sustains the practice.

**Error: Customer doesn't decide**—XP is customer driven; they need to decide what the acceptance criteria are (via tests), and what stories go in a release and iteration.

**Error: No customer-owned tests**—Ron Jeffries has said, "The failure to have customer-owned acceptance tests [in each iteration] is one of the most common mistakes in XP."

**Error: Minimal refactoring**—The XP avoidance of design thought before programming is meant to be compensated by a relatively large refactoring effort, such as 25% of total effort applied

to refactoring. Beck's point is that one can't avoid both design thought before programming and refactoring; it's either/or.

**Error: Must have only one onsite customer**—The original XP books talked of *one* onsite customer in the project room. Beck and the XP leaders have since refined this to emphasize that the customer team needs to be considered as a whole, with requirements coming from perhaps many customers participating in the Planning Games. Thus, they have replaced the practice "onsite customer" with "whole team together."

**Error: Many fine-grained task cards**—Most task cards should be in the one or two-day range of effort. Most in the "few hours" range creates unnecessary information management.

**Error: Pairing with one partner too long**—XP pairing changes frequently, often in two days or less. Variation also helps spread the learning.

**Error: Customer or manager is tracker**—Programmers will feel awkward reporting slow progress.

**Error: Not integrating the QA team**—Many organizations have a separate Quality Assurance team, used to having a completed system "thrown over the wall" to them. One or more dedicated QA people need to be brought onto the project full-time—the whole team practice—usually to write the acceptance tests in collaboration with the customer.

**Error: Post-development design documentation is wrong**—XP isn't anti-documentation, but prefers programming if that's sufficient to succeed. XP can support the creation of design documentation to reflect the existing code base, once the program is complete. As always, the simplest approach that can work is the goal, such as video recording an explanation.

**Error: Diagramming is bad**—Although XP advice is minimal modeling or diagramming, such as 15 minutes before programming, a very little is acceptable.

**Error: Only young pair programmers**—Some XP projects have suffered in the pair programming practice when most of the developers were quite young, without the patience or maturity to handle working closely with others.

**Error: Pairing newbies**—One of the two partners should have pair programmed before.

**Error: One partner going too fast**—When pair programming, the quicker or more experienced partner needs to be sensitive to the speed or comprehension differential of their partner, and slow down their activities and explanations.

**Error: Observer can't easily see the monitor**—Self-explanatory, but a surprisingly frequent problem.

**Error: Not willing to learn; not willing to explain**—For successful pair programming, an attitude of openness to learning and of explaining yourself is required.

**Error: Full team (including customers) not briefed in XP and its motivations**—Self-explanatory.

**Error: Dissenter on team**—XP is about communication and collaboration, and a culture of development; lone programmers who don't wish to accept it can impair the team culture and project progress.

**Error: Stand-up meeting too long or unfocused**—Keep it below 20 minutes, and on status of tasks, not a discussion of design and requirements.

**Error: Lumping into one big "bug fix" story**—As defects accumulate, don't group them into one task or story card; keep them with their original related cards.

**Error: No dedicated acceptance tester**—One is needed to work with the customer on transforming their acceptance criteria into runnable tests. An exception is very small projects—the tester may fulfill that role part-time.

**Error: Onsite customer and Big Boss aren't aligned**—XP talks of the Big Boss, or the person ultimately owning or responsible for the project goals and milestones. These two stakeholders need to be in agreement.

**Error: Customer writing acceptance tests isn't the reviewer of their execution**—A classic problem of serving different masters.

**Error: Iterations too long**—XP iterations should be 1–3 weeks.

**Error: Iterations aren't timeboxed**—It is a misunderstanding to let the iteration length expand when it appears the goals can't be met within the original time frame. Rather, the preferred strategy is usually to remove or simplify goals for the iteration. And, analyze why the estimates were off.

**Error: Iteration doesn't end in an integrated and tested baseline**—An iteration doesn't just successfully end on the end date. The goal is that all the software has been integrated, tested, and baselined.

**Error: Each iteration ends in a production release**—The baselined software produced at the end of an iteration is an *internal* release rather than shippable code. It represents a subset of the final production release, which may only be ready after a dozen or more short iterations. It is true that in iterative develop-

ment one goal is that each iteration release is stable enough to potentiality release to production, if necessary. However, this is not the normal intent of an iteration release.

**Error: Predictive planning**—It is a misunderstanding to create, *at the start of the project*, a believable plan laying out exactly how many iterations there will be for a long project, their lengths, and what will occur in each. This is contrasted with the agile approach: adaptive planning. The XP team and customer plans the next iteration, and then planning adapts iteration by iteration, based on current feedback.

#### You Know You Didn't Understand XP When...

Some of the key misunderstandings expressed as a checklist:

- ❑ You think you should customize the choice of practices without having first applied them all.
- ❑ You think "doing XP" means to avoid the waterfall model and develop iteratively, or just to avoid documentation, or just to write some unit tests.
- ❑ Customers are not involved in the Planning Game, creating acceptance tests, or reviewing the iteration results.
- ❑ You create a plan laying out how many iterations there will be for the project, their lengths, and what will occur in each.

## SAMPLE PROJECTS

The following projects had significant XP influence:

**Large**—Atlas leasing system

- Three years, 60+ people, Java technologies, an E100 project, [Schuh01]

- *Fully adopted practices*: simple design, testing, frequent refactoring, collective code ownership, continuous integration
- Pair programming was attempted, but did not stick. There was no onsite customer.
- Prime developer: ThoughtWorks.

**Medium**—Orca security incident-response

- One year, 25 people, a D40 project, [Morales02]
- *Fully adopted practices*: most practices, with the exception of small, frequent releases as this was a commercial product.
- Prime developer: Symantec.

**Small**—C3 payroll

- One year, 10+ people, an E20 project [C3Team98]
- *Fully adopted practices*: This was the original project that defined XP, coached by Kent Beck and Ron Jeffries. All practices were adopted.
- Prime developer: Chrysler.

## Process Mixtures

### XP + Evo

*Evo p. 211*

*Evo specifications p. 231*

XP values and spirit regarding specifications is not compatible with Evo. XP's value of avoiding written or precise requirements, and preferring oral communication between developers and requirement donors is very different than Evo's emphasis that when a specification is required, it be done so with clarity and measurable qualities.

On the other hand, many XP development practices may be consistently applied with Evo, such as test-driven development, pair programming, and so forth.

XP's client-driven adaptive planning is also consistent with Evo. The XP stand-up meeting, common project room, and whole team together supports Evo's feedback goals.

*adaptive planning p. 253*

XP's 1–3 week iteration length is relatively consistent with Evo, which prefers 1–2 week iterations.

### XP + Scrum

Most Scrum practices are compatible with XP. The Scrum meeting is a refinement of the XP stand-up meeting (in fact, Beck got the idea from Scrum), using special questions. Both recommend a common project room. The Scrum practice of a demo to external stakeholders at the end of each iteration enhances XP's feedback and communication goals. The Scrum Backlog and progress tracking approaches are minor variations of XP practices.

*Scrum p. 109*
*Scrum meeting p. 120*

Scrum's 30-day iteration length is not consistent with XP—too long.

A Scrum practice is to have only one customer representative, the Product Owner, who is ultimately responsible for the requirements and priorities. But in recent updates to XP, there is an emphasis on collaborating with a *group* of customers—avoiding just a single person. It is nice to have a single customer voice, and it is useful to know and resolve multiple people's goals. XP and Scrum tackle this tension differently, shifting whether development or business is responsible for resolving the conflict.

Mike Beedle, one of the early Scrum practitioners, has explored combinations of XP and Scrum under the name "XBreed."

*see www.xbreed.net*

### XP + UP

*UP p. 173*

*UP practices p. 186*

Most XP practices are either equal to or specializations of UP practices, and many XP practices can be applied in the context of an overarching UP project. For example, test-first development is a specialization of the UP *continuously verify quality* best practice. The UP does not require or promote unnecessary document creation—all artifacts are optional—and so it is a misunderstanding to assume the methods are fundamentally incompatible. Although speaking of some XP within UP can have conceptual integrity, the opposite is not true, as there are some differences in style and emphasis.

One area of difference is in the accepted degree of up-front modeling (diagramming, etc.). For example, within a UP project and a two-week iteration, it is considered acceptable to spend a half-day near the start to consider design ideas "at the whiteboard" before programming. In XP, no more than 10 or 20 minutes before programming is considered suitable.

Another difference is in the goal of the early iterations. In the UP the goal is to identify and drive down the high risks: technical, political, satisfying the customer, and so forth. Although this may happen in the XP, it is not an explicit guiding principle.

A third difference is in requirements specifications. The UP allows and supports the creation of relatively detailed specifications (evolutionarily, over a series of iterations), assuming that an onsite customer is not going to be present. These will usually take the form of use cases and an associated nonfunctional specifications document, created in a series of timeboxed requirements workshops. The idea in the UP is, during the early programming iterations, to have a parallel track of requirements analysis where the majority of requirements are being written, while the development team is also programming something critical. The programming work is meant, in part, to help clarify the requirements work.

XP stories are normally features, rather than use cases. Thus, XP promotes a feature-driven approach to requirements. On the other hand, the UP promotes a use-case-driven approach, although the UP accepts and allows features rather than use cases.

## ADOPTION STRATEGIES

As always, coaching by an experienced method expert on the first project is recommended.

Similar to Scrum, but in contrast to the recommended gentle, pilot-project adoption strategy of UP (for example), XP recommends adoption like this:

1. Pick the worst project or problem.
2. Apply XP until solved.
3. Repeat.

If all the XP practices can't be swallowed at once, Beck recommends starting with:

- ❑ whole team together in a common project room
- ❑ test-first development
- ❑ acceptance tests written/owned by customers
- ❑ Planning Game
- ❑ pair programming

That said, there are dangers in only adopting a few of the practices, especially if one does not understand how they support each other. Avoidance of up-front design (even on a per-iteration basis) is compensated by frequent refactoring. Frequent refactoring is

supported by continuous integration, and test-first development. And so forth.

Introducing customers to this new engaged approach is a challenge. The key is to help them see the early, tangible business value that they want, and emphasize that they will be steering the team to meet their needs in short cycles. An XP goal is to so delight the customer with this new-found control and responsiveness, that after a few iterations they will love the process.

On the common problem of customers wanting more and more in an iteration, one technique is to exploit the physical nature of story cards: Once the cards have been estimated and chosen for an iteration—and therefore consuming all available development resources—they are laid out on a table. Clearly, with the "no overtime" rule of XP, adding a new card means an existing one must be physically removed. This visual and tangible impact teaches a clear lesson.

If you can't get a customer into the room, what to do? First, look for another related representative, such as a product manager. If no luck, establish the closest possible communications. Visit them frequently, use a mobile phone, spend time at their job, have them use an instant messenger service to simulate the feel of close communication, have them attend the Planning Games, and show lots of demos.

Programmers will adopt most of the practices without concern, except pair programming. XP recommends not inviting programmers opposed to the idea to an XP project. Do not put only young programmers together; the maturity and patience of some older developers is necessary to make pairing work. Ensure the pairs mix regularly, the typing developer rotates frequently, and that people are learning from each other; i.e., sharing with their partners what they know, and what they are thinking. If pairs aren't frequently talking together, something isn't working.

For XP, the physical environment must change: open common room with development stations near the center, and the walls exposed for visible graphs, sketching, and so forth. And, the stations need to support pair programming. Ensure the non-typing partner can easily see the monitor.

Since pair programming implies lots of talk, a culture of quiet talking needs to be encouraged.

## FACT VERSUS FANTASY

Reading the XP method practices can create the impression that they are a silver bullet, but of course, they are not. As always,

> *Process is only a second-order effect. The unique people, their feelings and qualities, are more influential.*

One fantasy regarding XP adoption is found in groups that believe by just adopting iterative and evolutionary development and avoiding up-front specifications, they are "doing XP." Likewise with unit testing, working in a common project room, and so forth. Although data is still sketchy, it seems that many of the projects claiming to be doing XP are simply applying some iterative and evolutionary practices common to many IID methods (such as short iterations), and the group mistakenly believes these are unique XP ideas.

Probably the most common XP fantasy is getting onsite customers. It seems to be rare as hen's teeth to achieve this. Also, there is no shortage of so-called "XP" projects one investigates that could not arrange pair programming, which Beck considers one of the basics of an XP project.

Resistance to pair programming is perhaps the most common issue among developers. Some just don't want to do it.

It is also rare to find a common project room, or enough whiteboards.

Test-first development, early acceptance tests defined with customers, constant refactoring, and continuous integration are all widely confirmed as sustainable, excellent practices.

## STRENGTHS VERSUS "OTHER"

### Strengths

- Practical, high-impact development techniques, many of which are easily and sustainably adopted by developers (e.g., continuous integration, test-driven development).
- Emphasizes customer participation and steering.
- Evolutionary and incremental requirements and development, and adaptive behavior.
- Programmers estimate the tasks they have chosen, and the schedule follows this, not vice versa (i.e., scheduling is rational).
- Emphasizes communication between all stakeholders.
- Emphasizes quality through many practices. Test-first development, continuous integration, and team code ownership are examples.
- Clarifies what is an acceptable system by requiring the customer to define the acceptance tests.
- Daily measurement, and developers are involved in measuring and defining what to measure.
- Every iteration, developers get practice (during the Planning Game) identifying tasks and estimating them, leading to improvement in these vital skills.

- Frequent, detailed reviews and inspections, as all significant work is done in pairs. Inspection is strongly correlated with reduced defect levels.

### Other[3]

- Requires the presence of onsite customers (or proxies). This is often not possible, and their absence makes difficult or impossible the practice of "oral requirements" and using short story cards. XP has no standard solution for written requirements. That takes us back to other methods, such as the UP, which have a mechanism for iteratively recording detailed requirements.

- Relies on oral history for knowing the design and requirements details. This has limitations related to quickly helping new members, or scaling to larger projects.

- The XP practices are interdependent and mutually supporting. It isn't really a pick-and-choose process; most need to be done. Yet, people avoid some in the urge to avoid "unnecessary" steps, and thus failure ensues. Then, XP is unfairly criticized.

- No standard way to describe or document the software design as a learning aid.

- Some developers do not want to do pair programming.

- Many projects will need a set of documents other than code. XP does not define what these may be, and thus each project may create ones with similar intent, but varying names and content. In other words, no common, standard workproducts that are shareable with common names across projects. This impedes reuse of workproducts and impedes a common workproduct vocabulary in larger organizations.

3. Could be viewed as a weakness, strength, or deliberate desirable exclusion, depending on point of view.

- Lack of architecture-oriented emphasis in the early iterations. Lack of architectural design methods. XP advocates claim simple design and refactoring lead to the same goal.

## HISTORY

In the mid-1980s, Kent Beck and Ward Cunningham worked together in a research group at Tektronix. They founded the idea of CRC cards and (the seminal contribution of) design patterns, while building many Smalltalk systems. The roots of XP come from this collaboration. Eventually, Beck branched into private consulting, slowly (re-)discovering the various XP practices, such as the value of working in a common room. Cunningham went on to create the popular and unique Web concept of Wiki Webs.

In the mid-1990s, Beck was retained by Chrysler to help with a new Smalltalk-based payroll system, the C3 project. One aim of the project was the education of the staff in object-technology skills; a successful payroll system was desirable, but not the only goal. Beck introduced the majority of practices that became XP, and brought in Ron Jeffries to daily lead and coach the team. Martin Fowler was also invited for some consulting. Primarily led by the vision of Beck, the XP practices coalesced on this project.

Beck says that at its heart, XP is expressing what he learned with and from Cunningham.

There has been some mis-information that the C3 project "failed." In fact, management felt that the team received good object-technology education, and the C3 payroll system did successfully go into production for several thousand employees, but was eventually phased out and the team reassigned, as management—with direction from the new Daimler owners—developed different ideas for handling payroll at the new-found DaimlerChrysler company.

## WHAT'S NEXT?

The next chapter presents the practices of the UP, an iterative method with a somewhat different emphasis than XP. Following that, Evo is introduced—one of the first evolutionary methods.

## RECOMMENDED READINGS

There are many XP books but only a few essentials. *Extreme Programming Explained* by Kent Beck is required reading. A good practical companion by three members of the original C3 team is *Extreme Programming Installed.*

Supporting or related texts that are recommended include:

- *Test-Driven Development: By Example*, by Kent Beck. Teaches the essentials of this key XP practice.
- *Refactoring: Improving the Design of Existing Code*, by Martin Fowler. The bible on refactoring skills.
- *Peopleware*, by Tom DeMarco and Tim Lister. Discusses some of the people-side issues that inspired Beck in XP.
- *The Deming Management Method*, by W. Edwards Deming and Mary Walton. Discusses the critical role of personal pride in workmanship. This also influenced Beck and XP.
- *Toyota Production System: Beyond Large-Scale Production*, by Taiichi Ohno. This book—by the creator of the Toyota method—on "lean manufacturing" is something of the physical-goods equivalent to XP for software. Although Beck did not read this work until after creating XP, he has since highly praised it for capturing many of his goals and values in XP.
- "Episodes: A Pattern Language for Competitive Development" in *Pattern Languages of Program Design 2.* Article by Ward Cunningham, edited by John Vlissides. Cunningham presents some of the key ideas that became XP.