Course "Softwareprozesse"

**Agile Technical Practices:**

eXtreme Programming (XP), ...

Lutz Prechelt

Freie Universität Berlin, Institut für Informatik

- Structure of agile methods
  - values, roles, technical practices, mgmt. practices
- eXtreme Programming (XP)
  - XP1 vs. XP2 vs. Jeffries
  - Values, roles
  - Management practices
  - Technical practices
- Criticism
- When (not) to use XP
- Empirical results: a survey
- XP and CMMI
- Other technical practices:
  - Continuous Deployment
  - Pragmatic programmer
Learning objectives

• Understand
  • the structure of methods,
  • the role of practices, and
  • the difference between management practices and technical practices

• Roughly understand the values and practices that make up XP and how they play together

• Roughly understand when to and when not to use XP
Agile "methods" vs. "practices"

"method": a systematic procedure for attaining something

• In contrast, so-called "agile methods" tend to be looser collections of procedure elements
  • Users need to find out how to arrange and use them

• They consist of
  • concepts (values, roles, artifact types, states, etc.),
  • principles, and
  • practices

• The latter can be divided into
  • (the discrimination is not sharp)
    • technical practices: concerned with code-centric work
    • management practices: concerned with everything else: requirements work, scheduling issues, delivery, staff issues, process improvement
How we will proceed

• Today:
  • Present **eXtreme Programming (XP)** as an example method
    • its values, roles, and practices
  • Focusing on **its technical practices**
    • even skipping some of the management practices completely
  • Present some additional technical practices

• Next week:
  • Look at **Pair Programming** in depth
    • a powerful and controversial **technical practice**
      (with management value, too)

• Week after next:
  • Look at **management practices** in more depth
    • by presenting "methods" that have no technical practices: **Scrum, Lean SD, Kanban**
Martin Fowler on XP

- "To make agile work, **you need solid technical practices**.
- A lot of agile education under-emphasizes these, but if you skimp on this you won't gain the productivity and responsiveness benefits that agile development can give you (stranding you at level 1 of the agile fluency model.)
- This is one of the reasons that I still think that **Extreme Programming is the most valuable of the named agile methods** as a core and starting point."

- [http://martinfowler.com/agile.html](http://martinfowler.com/agile.html)
XP sources

• The original, definitive source on XP is Kent Beck's book "Extreme Programming Explained: Embrace Change", Addison-Wesley, 1999 (or in short: Article)

• However, there is now a 2nd edition (2004)
  • Complete rewrite (with Cynthia Andres)
  • Fairly different set of practices:
    • Some removed (too difficult or too easy),
    • some made more precise (e.g. by quantification),
    • some added
  • Thus, the modified method is sometimes called XP2

• Ron Jeffries (xprogramming.com) uses a still different mix
  • Beck and Jeffries are the co-inventors of XP

• Many more books and articles have been written about XP
### Practices of XP, XP2, Jeffries' XP
(furthermore, XP2 has 11 "Corollary Practices")

<table>
<thead>
<tr>
<th>XP1 practices (&quot;traditional&quot;)</th>
<th>XP2 practices (&quot;evolutionary&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The Planning Game</td>
<td>1. Stories</td>
</tr>
<tr>
<td>2. Small Releases</td>
<td>2. Weekly Cycle</td>
</tr>
<tr>
<td>3. 40-Hour Week</td>
<td>3. Quarterly Cycle</td>
</tr>
<tr>
<td>4. On-Site Customer</td>
<td>4. Energized Work</td>
</tr>
<tr>
<td>5. Pair Programming</td>
<td>5. Slack</td>
</tr>
<tr>
<td>6. Collective Ownership</td>
<td>6. Whole Team</td>
</tr>
<tr>
<td>7. Metaphor</td>
<td>7. Sit Together</td>
</tr>
<tr>
<td>8. Simple Design</td>
<td>8. Informative Workspace</td>
</tr>
<tr>
<td>10. Testing</td>
<td>10. Incremental Design</td>
</tr>
<tr>
<td>11. Continuous Integration</td>
<td>11. Test-First Programming</td>
</tr>
<tr>
<td>12. Coding Standards</td>
<td>12. Continuous Integration</td>
</tr>
<tr>
<td></td>
<td>13. Ten-Minute Build</td>
</tr>
</tbody>
</table>

M: Mgmt, T: Technical

**J: Jeffries' additional practice:**
- Customer tests
XP practices: XP2, XP1, XP2 "corollary" (optional)

Note: Some connections are missing

Graphic: Stefan Roock
Ron Jeffries' view: core, infrastructure, customer interface

https://ronjeffries.com/xprog/what-is-extreme-programming/
XP roles

- Developer
  - the only role with always more than one representative
- Customer
  - usually (but not necessarily) a non-technical person
- Coach
  - responsible for process as a whole; guides the team to proper XP
- Tester
  - helps the customer write function tests (customer tests)
- Tracker
  - collects and feeds back estimates and plan tracking

- Customer, Tester, Tracker need not be full-time and thus may double as developer
  - but Coach should not.
  - Coach might double as Tracker and Tester
XP basic values

XP's set of practices is based on five fundamental ideas (called "values"):

- Communication
- Simplicity
- Feedback
- Courage ("Mut")
- Respect

The consequences of these crosscut technical and management practices.

see next slides
Basic values: Communication

• Very many problems in projects are related to communication that failed or simply did not happen
  • e.g. tacit assumptions about requirements
  • e.g. uncoordinated technical decisions
  • e.g. missing information about design ideas
  • e.g. missing notification about technical changes

• Therefore, XP uses practices that enforce early, frequent, successful communication
  • Practices that require communication:
    • continuous integration
    • effort estimation in the planning game
  • Practices that create communication:
    • pair programming
    • informative workspace
    • frequent releases
Basic values: Simplicity

• Simple solutions have many nice properties:
  • they are easy to design
  • they are easy to implement
  • they are easy to test and debug
  • they are easy to communicate and explain
  • they are easy to change

• This is true for both product and process

• Therefore, XP requires to always use the simplest solution that is sufficient for today's requirements
  • and not build something more complicated in the hope that it will be needed later.
  • Slogan: "You Ain't Gonna Need It!" (YAGNI)
Basic values: Feedback

• It is immensely helpful for a project if it always gets quick feedback about the consequences of actions or plans
  • How expensive would it be to realize this new requirement?
  • Is this new piece of code correct?
  • Does it fit with the rest of the system?
  • How useful is the system overall?

• Therefore, XP integrates concrete and immediate feedback into the process wherever possible
  • Immediate effort estimation for each storycard
  • Unit tests for each piece of code
  • Continuous integration
  • Short iterations and frequent releases
Basic values: Courage

- Many aspects of realizing the first three values require courage:
  - e.g. communicating that you will change an oft-used interface
  - e.g. building a simple solution only, although you firmly expect it to become insufficient later
  - e.g. facing negative feedback about incorrect code, incompatible interfaces, infeasible requirements, or impractical aspects of a delivered system

- Therefore, XP both uses a culture that encourages courage
  - e.g. with pair programming and the planning game
- and creates an infrastructure that allows to be courageous or even bold
  - in particular with automated testing and continuous integration
Basic values: Respect

- Respect
  - of one developer for another,
  - of developers for customer, and
  - of customer for developers

- is an important basis for continually realizing
  - communication,
  - feedback, and
  - courage

- Therefore, respect underlies all of XP as a kind of continuous admonition
  - it was not explicitly listed as a value in the XP1 book
Side note:
Management practices
Management practices: Iteration-related practices

• Agile development is always iterative.
• Managing this is a major aspect of any agile method

XP practices:
• XP1/J: Planning Game, XP2: Stories
  • (See last week)
  • Determine what goes into an iteration
  • A major topic of Scrum

• XP1/J: Small releases, XP2: Weekly cycle, Quarterly Cycle
  • keep iterations short!
  • Kanban talks a lot about this

• XP1: 40-hour week, J: Sustainable pace, XP2: Energized work, Slack
  • "No extended overtime"
  • and don't forget you need to learn new methods and technologies, too.

Lutz Prechelt, prechelt@inf.fu-berlin.de
XP2/J: Whole Team, XP1: On-site customer

- All qualifications and competences required should be represented in the team
  - this includes specialized technical knowledge
  - as well as business/requirements knowledge ("on-site customer")
  - as well as project-level responsibilities (coach, plan tracker)
- Thus, the team can always proceed without interruption

- Criticism:
  - It is often impossible to find a single person representing all requirements knowledge (or to bring several into the team)
  - XP requires all members to be full-time, but very specialized (and rare) technical knowledge may be needed in multiple projects
XP2: Sit Together

• The whole team should work as close together as possible, ideally in a single large office.
  • This greatly simplifies communication and makes it more likely to succeed
  • It greatly increases informal communication
    • by overhearing other pairs working

• Criticism:
  • 10 people in one room leads to high background noise and reduces concentration

Lutz Prechelt,

* http://stateofagile.versionone.com
XP1: Informative Workspace

- All important information about the project status should be available directly in the workspace, e.g.
  - currently open tasks
  - build and test status
  - architectural design sketch

- This can often be done by hanging note cards or flip chart sheets on the walls
Technical practices
XP1/2/J: Pair Programming (PP)
XP1/J: Collect. ownership, coding stds.

- All production code is written by two programmers working together at a single computer
  - Thus, a better design can be found,
  - many mistakes can be caught immediately,
  - the partners learn from each other
    - technology, operating style, design process, project details, etc.
  - at least two people are highly familiar with each piece of code.
  - Pairs switch frequently (e.g. twice daily)
  - Collective ownership and Coding standards make PP practical.

- Criticism:
  - (plenty)

- (Detailed discussion next week)
XP1/J: Simple Design, Metaphor  
XP2: Incremental Design

• The design is completed step-by-step, along with the code
  • It is not invented all at once beforehand
    • which would be known as "Big up-front design" (BUFD)
  • At any time, the design is oriented only towards the current requirements, not to those just expected to come later
    • XP1 (precariously): "Use the simplest design that can possibly work"
• Metaphor is a shared idea for the overall design approach that serves as core architecture
• When design changes are required, refactoring is used as the first step (in order to minimize risk)

• Criticism:
  • When used naively, this usually leads to very high amounts of rework, as "architecture breakers" then occur frequently
    • In particular, the XP 1 phrasing "Simple design" can mislead
What makes a design "simple"?

- To build "the simplest design that can possibly work" implies building the system with the smallest possible number of classes and methods in such a way that:
  - code and tests together clearly describe what we want to express and
  - there is no redundancy in the code

- Slogan: "Do everything once and only once" (OAOO)

- Eliminating redundancy automatically leads to a system that is clear, flexible, and that can easily be extended and adapted
  - Slogan: "Don't repeat yourself" (DRY)
  - However, recognizing and eliminating redundancy is difficult!
Simple design option cost example

Assume you build the simplest possible design $D$ today:

- Assume change $A$ becomes necessary 1 year later:
  - €1000 $D$ cost today
  - €1500 $A$ cost next year

- Assume incompatible change $B$ becomes necessary instead:
  - €1000 $D$ cost today
  - €1500 $B$ cost next year

Assume you build $D'$ anticipating a change $A$:

- Assume change $A$ becomes necessary 1 year later:
  - €1500 $D'$ cost today
  - €50 interest (10% of $D'$-$D$)
  - €500 $A$ cost next year

- Assume incompatible change $B$ does instead:
  - €1500 $D'$ cost today
  - €50 interest (10% of $D'$-$D$)
  - €500 $A$ rework cost next year
  - €1500 $B$ cost next year

If the uncertainty of $A$ vs. $B$ is high, $D'$ may be a bad idea!
XP1/J: Refactoring

- Refactoring means modifying the structure of a program without modifying its behavior
  - There are a number of well-defined elementary refactoring operations, e.g.:
    - Rename
    - Change Method Signature, Introduce Parameter
    - Convert Local Variable to Field, Encapsulate Field
    - Extract Class/Interf./Loc. Var./Method (opposite: inline)
    - Introduce Factory
    - Generalize Type, Pull Up, Push Down elements in class hierarchy
- XP allows courageous refactoring: the **automated tests** make it easy to verify whether a refactoring is correct
- Modern IDEs (such as Eclipse/IntelliJ for Java) support or even automate several such refactoring operations
XP1: Testing, J: Test-driven development
XP2: Test-First Programming

- **Before** some program element is written (e.g. a modest method), an automated test of this element is always written first
  - The test must fail as long as the element is still missing
  - It must succeed for the element to be considered finished

- **Advantages:**
  - Clarifies the requirements for the element before coding it
  - Defines the interface
  - Provides rapid and constant feedback
  - Thus allows courage during refactoring

- **Criticism:**
  - This amounts to a very high level of test automation which may be inefficient
J: Customer Tests

• Write automated tests at the story-level
  • testing relevant, user-visible, valuable functionality directly
  • ideally in a form the end user can read (for validation).
  • They then serve as always up-to-date documentation.
    • Very useful for user support.

• These add confidence beyond what unit tests and integration tests can provide
  • balance with the unit and integration tests, limit redundancy
  • write more of them if you often break stories

https://ronjeffries.com/xprog/xpmag/problems-with-acceptance-testing/
https://ronjeffries.com/xprog/blog/automating-story-tests/
• Developers check in their work into the common code base several times each day

• An automated process then (1) builds the system, (2) runs the automated tests, (3) logs results
  - XP2: Ten-minute build ensures results are frequent and understandable

• This build represents the project state
  - The build should be fully functional most of the time
  - A build that remains broken for some time is an indicator of bad project health

• Criticism:
  - It may be expensive to keep up a functional build during large refactorings
Practices support each other!

For instance:

- Simple Design simplifies PP and TDD
- Refactoring helps create Simple Design, perform PP, and perform TDD
- TDD makes Simple Design and Refactoring less frightening
- PP helps maintain discipline for Simple Design, Refactoring and TDD

...and so on with other practices
Criticism

- Gerold Keefer: "Extreme Programming Considered Harmful for Reliable Software Development 2.0", 2003
  (an earlier version appeared in the conference Conquest 2002 by isqi.org)
- Critically reviews the claims and reports about XP and argues that it is recommendable only in rare situations:
  - Requires staff competence far above average
  - Requires unusually high team stability (→ no documentation)
  - Cannot work if finding a suitable architecture is difficult
  - Is applicable only to projects of modest size
- Provides a good overview of the XP-related literature until 2002

- Many other criticisms of XP exist
  - Many of them unbalanced, half-ignorant, and highly polemic
  - Refer to Barry Boehm's balanced judgement as a primary source
When you should **not** use XP

From Kent Beck's book:

- **Too-big teams**
  - XP works for teams of 10, can work for teams of 20
  - For teams of 100, integration (that is, design coordination) will become a bottleneck

- **Unbelieving customers and organizations**
  - XP requires full concentration; it cannot work in a culture of continuous extensive overtime
  - Customers who insist on a thick specification document break the whole XP process

- **Change-hampering technology or constraints**
  - e.g. replacing a database that absolutely must be compatible with 164 different applications
  - e.g. working with technology that makes builds take 10 hours
  - e.g. working with insufficient opportunity for immediate communication
Introducing XP

• It is difficult to introduce all XP practices at once
  • Most need to be learned!

• They can be introduced one-by-one as follows:
  • Find the worst problem/weakness of the current process
  • Select the XP practice that can help most with this problem
  • Introduce it until the problem is much reduced
  • Find the now-worst problem and start over

• Good candidates for first practice to introduce:
  • Sit Together
  • Quarterly Cycles (Stories)
  • Continuous Build & Testing
A survey of XP use in embedded(!) systems projects

- Responses from 35 projects from 13 organizations from 8 countries
Results: Use of XP practices

- Simple Design
- 40-hour week
- Test-Driven Development
- Planning game
- Coding Standards
- Refactoring
- Continuous Integration
- Collective Code Ownership
- Open office space
- On-site customer
- Pair-Programming

Legend:
- □ systematically
- □ mostly
- □ sometimes
- □ rarely
- □ never
- □ N/A
- □ ?
Results: Experienced usefulness

Expectations of respondents *without* XP experience were 28% negative.

XP perceived as more useful than Scrum!
CMMI process areas in XP


- **Level 2: Managed**
  - + Requirements Mgmt
  - + Project Planning
  - + Project Monitoring & Control
  - - Supplier Agreement Mgmt
  - (Measurement and Analysis)
  - o (Process and) Product Quality Assurance
  - + Configuration Management

- **Level 3: Defined**
  - (Req's. Development)
  - + Technical Solution
  - (Product Integration)
  - + Verification
  - (Validation)
  - o Organizational Process Focus
  - o Organ'l Process Definition
  - o Organizational Training
  - + Integrated Project Mgmt.
  - (Risk Management)
  - (Decision Analysis and Resolution)

- **Level 4: Quantitatively Manag'd**
  - - Organizational Process Performance
  - - Quantitative Project Mgmt

- **Level 5: Optimizing**
  - - Organizational Performance Management
  - o Causal Analysis and Resolution

+ usually available
o avail. in reduced form
- usually mostly absent

(not a part of CMM, only CMMI)
CMMI versus XP

Paulk's summary:

- XP generally focuses on technical work
  - whereas the CMM generally focuses on management issues.
- Both methods are concerned with "culture"
- The CMM element most lacking in XP is "institutionalization"
  - Establishing a culture of "this is how we do things around here"
    - (lacking on organization level, but strong on team level)
  - XP largely ignores the infrastructure that the CMM identifies as key to institutionalizing good practices
- As systems grow, some XP practices become more difficult to implement
- Modern software projects should capture XP values
- CMM tells organizations what to do but does not say how
  - XP is a set of best practices with specific how-to information
An important non-XP technical practice: Continuous Deployment

- When a build is successful, it will automatically and immediately be deployed to the production system
  - So effectively iterations become *extremely* short
  - Only possible for web-based and similar systems
  - Precondition: Continuous Delivery (➡ Kanban)

- An ambitious goal!
  - e.g. Amazon, Facebook do this

https://www.agilealliance.org/glossary/continuous-deployment/
The Pragmatic Programmer

- Andrew Hunt, David Thomas: "The Pragmatic Programmer: From Journeyman to Master", Addison-Wesley 1999
- 70 "tips" of advice and practices for individuals, not teams
- Technical advice ranges from design (contracts, refactoring,...) over testing (using `assert`, ...) and debugging, to the use of command shells
Further technical practices

Lines represent practices from the various Agile "tribes" or areas of concern:

- Extreme Programming
- Scrum
- Design
- Teams
- Product management
- Testing
- Lean
- Devops
- Fundamentals

https://www.agilealliance.org/agile101/subway-map-to-agile-practices/
Further resources

• http://www.agilealliance.com
  • A community portal around the agile approach. Has lots of comments on XP.

• http://www.xprogramming.com
  • Ron Jeffries

• http://fairlygoodpractices.com
  • Some more practices that are helpful
    • including practices related to various toys

• http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap
  • A section of the original wiki.
    About many aspects of XP and its development.
Summary

- **XP** is a set of practices that mutually reinforce and support one another
  - in particular technical practices lacking in Scrum and Kanban

- It is based on the basic values of
  - intensive and direct communication,
  - simplicity in design and process,
  - early and constant feedback
  - courage in allowing things to change
  - mutual respect

- Successfully using XP requires
  - a highly competent and disciplined team and
  - the right environment: on-site customer, suitable project type
Thank you!
Old slides
Preamble: Why we look at XP

- In the early 2000s, XP was the most well-known agile method
  - most popular, most discussed

- Today, it is much less talked about, because many of its practices have become mainstream.
  - Many XP practices are used with most other agile methods
    - Sometimes explicitly, but often as a matter of course
    - So the relevance of knowing XP is as high as it was

- XP is still the most complete agile process model.
  - So the relevance of knowing XP is higher than it is for, say, Scrum or Kanban
  - XP focuses on technical work, less on management
  - Scrum and Kanban focus on mgmt., hardly on technical work

Lutz Prechelt, prechelt@inf.fu-berlin.de
History

- XP is based on ideas that have been around for a long time
- XP was developed into a method in the context of one single software project (using Smalltalk)
  - "C3": *Chrysler Comprehensive Compensation*, a project to develop a payroll system for the 87000 employees of Chrysler Corporation.
  - 1995-01: C3 starts
  - 1996-03: C3 has not delivered any working functionality. *Kent Beck* is hired as an advisor, brings in *Ron Jeffries*, reduces project staff, and starts putting C3 into XP mode
  - 1996 to 1998: A period of high productivity in the project
  - 1998-08: C3 system is piloted and payrolls 10 000 employees
  - 2000-02: C3 project is canceled after Chrysler/Daimler-Benz merger
XP practices

- XP as a method consists of a set of practices
  - Their manner of application can be adapted
  - but all of them are mandatory for a full XP process
    - although in practice very often not all are used.
    - Just picking your favorite five or so is not XP!
  - They mutually reinforce each other

```
XP1:
- On-site Customer
- Planning Game
- 40 hour week
- Acceptance Testing
- Pair Programming
- Simple Design
- Refactoring
- Collective Ownership
- Continuous Integration
- Short Releases
- Continuous Integration
- Testing
- Coding Standards

Graphic: K. Schneider
```
Practice: Energized Work

- All members of the team are motivated and work energetically at any time
  - In particular, there are no extended stretches of working overtime
    - This was formerly called "40 hour week" which was too inflexible in practice
  - Also, since Pair Programming (see below) is very intensive, a good routine of breaks and fun interludes is important

- Criticism:
  - Can you really call "working energetically" a practice that you consciously adopt?
Practice: Stories

- All requirements are stated in the form of stories
  - A short reminder is written on a card
  - Most of the information transfer is done verbally
  - The number of such cards must be modest
    - Mostly cards for the current iteration, never cards beyond the current release

- Criticism:
  - For some types of functionality, stories are just too imprecise
  - Non-functional requirements cannot be expressed by stories
    - but need to be considered early

www.jamesshore.com/Multimedia/Beyond-Story-Cards.html
Practice: Weekly Cycle

- The finest granularity of project-level planning is the so-called "iteration"
  - Each iteration implements one or more stories
  - An iteration should take about one week, maybe two
- The iteration is the elementary progress step visible for the customer
- During an iteration, requirements are fixed
  - Programmers can work without interruption
  - Programmers can estimate the effort well for work of this size
Practice: Quarterly Cycle

• The larger granularity of project planning is the release
  • There should be about four releases per year
  • A release is deployed into actual use by actual users (at least a pilot group) in actual business processes
• Frequent releases provide regular reality checks of the value generated by the project
  • and provide a rhythm for reflecting on the development process

• Criticism:
  • Rollout of a release is often very difficult and cannot be done frequently (e.g. because of required process changes)
Practice: Slack

• Developers have some freely available time (slack time) to be used for non-project work
  • e.g. learning about new technology.
• This time will also allow to eliminate delays from misestimation

• Criticism:
  • It is extremely difficult to keep up this practice in normal project reality for most organizations
Criticism:

• For many kinds of task (in particular simple ones), PP may be rather inefficient
  • See next week for a theory of when this is so
  • PP tends to help correctness and design quality
• Some programmers do not accept this style of working
• Pair partners may prefer different working styles
Practice: Ten-Minute Build

- Building the system and running system-level function tests must not take longer than 10 minutes
  - so that it is realistic that programmer-driven function testing occurs after each significant programming session

- Criticism:
  - This may be impossible for multi-platform products