Course "Softwareprozesse"

Agile Technical Practices: eXtreme Programming (XP), Part II

Lutz Prechelt
Freie Universität Berlin, Institut für Informatik

- Shared Code, Coding Standards
- Refactoring
- Simple Design, Incremental Design
- Practices support each other
- Values and practices

- Criticism of XP
- Usage survey
- When not to use XP
- Introducing XP
- Further technical practices
# 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>M ✓ 1. Stories</td>
</tr>
<tr>
<td>2. Small Releases</td>
<td>M ✓ 2. Weekly Cycle</td>
</tr>
<tr>
<td>3. 40-Hour Week</td>
<td>M ✓ 3. Quarterly Cycle</td>
</tr>
<tr>
<td>4. On-Site Customer</td>
<td>M ✓ 4. Energized Work</td>
</tr>
<tr>
<td>5. Pair Programming</td>
<td>T 5. Slack</td>
</tr>
<tr>
<td>6. Collective Ownership</td>
<td>T 6. Whole Team</td>
</tr>
<tr>
<td>7. Metaphor</td>
<td>T 7. Sit Together</td>
</tr>
<tr>
<td>8. Simple Design</td>
<td>T 8. Informative Workspace</td>
</tr>
<tr>
<td>10. Testing</td>
<td>T ✓ 10. Incremental Design</td>
</tr>
<tr>
<td>11. Continuous Integration</td>
<td>T ✓ 11. Test-First Programming</td>
</tr>
<tr>
<td>12. Coding Standards</td>
<td>T ✓ 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
Shared code, XP1/J: Coding standards

- Shared Code (corollary practice, XP1: Collective code ownership) means
  - "Anyone on the team can improve any part of the system at any time."
- Important for agility
  - especially Simple Design
- Requires a sense of responsibility
  - hence is corollary in XP2

- Coding standards (XP1) means
  - there are rules for code formatting
  - and for naming
- Important to make Shared Code and Pair Programming practical

- No longer in XP2
  - because it has become nearly self-understood
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 Function Declaration
    - Extract Class/Function/Superclass/Variable
      - opposite: Inline Class/Function/Superclass/Variable
    - Pull Up, Push Down elements in class hierarchy

- Modern IDEs (such as Eclipse/IntelliJ for Java) support or even automate several such refactoring operations
What is Refactoring?

- Refactoring is not just improving the design structure of a program
  - This is unavoidable in iterative development
- It is improving the design structure without changing the behavior
  - This can be a simplification if you have a good test suite
  - It is cumbersome otherwise
- XP allows courageous refactoring: the **automated tests** make it easy to verify whether a refactoring is correct
Fowler: Workflows of refactoring

- Martin Fowler: "Workflows of Refactoring", OOP 2014
  - Video
- TDD refactoring (2:28)
  - post-hoc design
- Yuck! refactoring (7:25)
  - clean up bad code
- I-don't-understand-this-refactoring (10:25)
  - Materialize freshly gained understanding
- Always: Find the right time (12:38)
  - Refactor only if tests are green!
- We-should-have-done-it-this-way refactoring (14:40)
  - prepare for future features
- Planned refactoring (17:30)
  - for all I have not yet learned how to do underway
- Long-term changes (19:14)
  - gradual contributions to large-scale design changes
- Always: Purpose is 'design stamina'
Research: Refactoring impact

  - based on 76 studies, many using multiple datasets

<table>
<thead>
<tr>
<th>Quality attribute</th>
<th>++</th>
<th>+</th>
<th>Insign. 0</th>
<th>0</th>
<th>-</th>
<th>--</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated External</td>
<td>0</td>
<td>39</td>
<td>0</td>
<td>13</td>
<td>41</td>
<td>0</td>
</tr>
<tr>
<td>Maintainability</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>13</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>Understandability</td>
<td>0</td>
<td>35</td>
<td>0</td>
<td>13</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>Reusability</td>
<td>0</td>
<td>25</td>
<td>0</td>
<td>13</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>Adaptability</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Flexibility</td>
<td>0</td>
<td>31</td>
<td>0</td>
<td>12</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>Testability</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Extensibility</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quality attribute</th>
<th>++</th>
<th>+</th>
<th>Insign. 0</th>
<th>0</th>
<th>-</th>
<th>--</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured External</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Reliability</td>
<td>6</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>Maintainability</td>
<td>0</td>
<td>30</td>
<td>3</td>
<td>100</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>Internal</td>
<td>3</td>
<td>108</td>
<td>18</td>
<td>129</td>
<td>146</td>
<td>21</td>
</tr>
<tr>
<td>Coupling</td>
<td>52</td>
<td>201</td>
<td>54</td>
<td>48</td>
<td>72</td>
<td>0</td>
</tr>
<tr>
<td>Cohesion</td>
<td>3</td>
<td>89</td>
<td>1</td>
<td>69</td>
<td>112</td>
<td>13</td>
</tr>
<tr>
<td>Complexity</td>
<td>1</td>
<td>42</td>
<td>7</td>
<td>17</td>
<td>42</td>
<td>4</td>
</tr>
<tr>
<td>Size</td>
<td>0</td>
<td>30</td>
<td>3</td>
<td>100</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>Inheritance</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Composition</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Data encapsulation</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Interpretation is difficult

Lutz Prechelt, prechelt@inf.fu-berlin.de
Research: Refactoring impact
Interpretation problems

- Most of the metrics applied are naive
  - e.g. coupling is a static measure: Each coupling counts the same
  - but in practice, some couplings hurt much more than others
- Tradeoffs occur:
  - To reduce coupling here, I sometimes increase coupling there
    - Perhaps avoidable, but if it costs more work and does not pay off...
- The competence of people and teams varies enormously
- (Probably several other problematic factors)
- What was even counted as a refactoring in those studies?
  - Refactorings are wildly mixed with other changes

Conclusion:
- The SLR does not tell us much about Refactoring.
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
- When new functions require design changes, Refactoring is used as the first step
  - in order to minimize risk

Criticism:

- When used naively, this may lead to very high amounts of rework, because "architecture breakers" may then occur frequently
  - In particular, the XP 1 phrasing "Simple design" can mislead
Simple Design:
Kent Beck's XP1 formulation

[Beck99]:

• "[Do] the simplest thing that could possibly work"

• Simple Design:
  1. "runs all the tests,
  2. communicates everything the programmers want to communicate,
  3. contains no duplicate code, and
  4. has the fewest possible classes and methods."
Incremental Design: Kent Beck's XP2 formulation

- "Invest in the design of the system every day.
  - Strive to make the design of the system an excellent fit for the needs of the system that day.
  - When your understanding of the best [...] design leaps forward, work gradually but persistently to bring the design back into alignment with your understanding."

- "Without daily attention to design, the cost of changes does skyrocket."

- "[Do not] minimize design investment over the short run, but [...] keep the design investment in proportion to the needs of the system so far.
  - The question is not whether or not to design,
    - the question is when to design.
    - Incremental design suggests that the most effective time to design is in the light of experience."

- "The simple heuristic [...] is to eliminate duplication."
What makes a design "simple"?
Low redundancy

- A low amount of duplication is not the only attribute of a high-quality design
  - but worth particular attention when a design is created incrementally

- Slogan: "Do everything once and only once" (OAOO)
- Slogan: "Don't repeat yourself" (DRY)

- Eliminating redundancy usually leads to a system that can easily be extended and adapted
  - However, recognizing and eliminating redundancy is difficult!
What makes a design "incremental"?
Avoid implementing ahead (YAGNI)

- Experience suggests that we are not good at predicting what changes will be needed in the future
  - Some we do not see coming at all
  - Others we see coming only vaguely
    - So our precautions against them may be the wrong ones
- Investing in flexibility mechanisms (to accommodate changes) is then risky

- Slogan: "You ain't gonna need it" (YAGNI)
  - Do not invest into flexibility mechanisms that are not yet needed.
  - Build flexible designs
    - if that flexibility is required now or
    - if implementing that flexibility does not cost anything
  - Think ahead, but do not implement ahead.

- Depressingly little research has been done on this idea
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!
Why is Incremental Design critical?

- Incremental Design is a lot of work
  - you often shift around lots of things with no immediate functional benefit
- Non-technical stakeholders get in the way:
  - I.D.'s benefit is even harder to see than that of tests
- ➔ It is easy for a team to neglect this practice
  - requires lots of discipline to keep it up
- ➔ XP is perhaps a better starting point than Scrum

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

XP1/2/J: Pair Programming (PP)

• 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:
  • How can this possibly pay off?

• (Detailed discussion next week)
Practices support each other!

For instance:

- Incremental Design simplifies PP and TDD
- Refactoring helps create Incremental Design, perform PP, and perform TDD
- TDD makes Incremental Design and Refactoring less frightening
- PP helps maintain discipline for Incremental Design, Refactoring and TDD
- …and so on with other practices

Jeffries' XP core
XP corollary practices

Graphic: Stefan Roock

Note: Some connections are missing
Chapter 9:

- Practices that are difficult or dangerous when the Primary practices are not yet fully in place.
  - "*Trust your nose about what you need to improve next.*"
  - *If one of the following practices seems appropriate, give it a try.*
  - *It might work or you might discover that you have more work to do before you can use it to improve your development process."

Interesting ones:

- Real Customer Involvement **M**
  - not only a proxy
- Team Continuity **M**
  - "*Keep effective teams together.*"
- Root-cause Analysis **M T**
  - Remove causes of defects
- Code and tests **T**
  - all else will be generated
- Single Code Base **T**
  - → toggles, not branches and others
XP values revisited

- Communication, Feedback, Courage, Respect sound like typical humanist agile bla bla.

- The values may not look technical but all of them are reflected in the technical practices

So let us look at that:
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
Values: Simplicity

- Simple solutions have many nice properties:
  - they are easy to design \( T \)
  - they are easy to implement \( T \)
  - they are easy to test and debug \( T \)
  - they are easy to communicate and explain \( T \)
  - they are easy to change \( T \)
- This is true for both product and process \( T \)

- 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)
YAGNI illustrated: "You Ain't Gonna Need It"

CAN YOU PASS THE SALT?

I SAID-

I KNOW! I'M DEVELOPING A SYSTEM TO PASS YOU ARBITRARY CONDIMENTS. IT'S BEEN 20 MINUTES! IT'LL SAVE TIME IN THE LONG RUN!

https://xkcd.com/974/
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? T
  • Does it fit with the rest of the system? T
  • How useful is the system overall?

• Therefore, XP integrates concrete and immediate feedback into the process wherever possible:
  • Immediate effort estimation for each storycard
  • Short iterations and frequent releases
  • Continuous integration, a rapid build T
  • Unit tests for each piece of code T
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 uses a culture practice that encourages courage
  - pair programming! 😊

- and creates an infrastructure that allows to be courageous or even bold
  - in particular with automated testing 😊 and continuous integration 😊
Values: Respect

- Respect
  - of one developer for another,
  - of developers for customer, and
  - of customer for developers (so we can keep up XP practices)

- is an important basis for continually realizing
  - communication,
  - feedback, and [e.g. respect asks to write all those tests]
  - courage
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)
  • Provides overview of XP-related literature until 2002
• Critically reviews the claims and reports about XP and argues that it is recommendable only in rare situations:
  • Requires staff competence far above average
    • XP reply: People can learn
  • No documentation: Requires unusually high team stability
    • XP reply: Not unusual for us!
  • Cannot work if finding a suitable architecture is difficult
    • XP reply: Often it is not. If it is, XP-style experiments can help.
  • Is applicable only to projects of modest size
    • XP reply: Large projects can use restricted XP at team level
• Who is right?
  • Depends! (Barry Boehm's balanced judgement is a better source)
A survey of XP use in embedded(!) systems projects

- Responses from 35 projects from 13 organizations from 8 countries

Lutz Prechelt, prechelt@inf.fu-berlin.de
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.
When you should **not** use XP

From the XP 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
    - "Change begins with awareness."
  - 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
  - Pair Programming
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/
Summary

- Agile development must work to keep design structure intact
  - Refactoring may be useful for doing this
  - Refactoring is difficult to research

- Incremental Design means avoiding to look ahead too much
  - and aim for a healthy SW structure despite the many changes

- XP practices support each other
  - and support the XP values

- XP should often be introduced practice-by-practice
  - many agile teams use technical practices too little
Thank you!

(extra slides follow)
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 focusses 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
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 responsibles (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
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
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, e.g.:
  - fix yet-unknown defects
  - improve yet-unknown gaps in existing design structure
    - ("repay technical debt")
    - (in a strong XP team, these two items will be small)
- Criticism:
  - It is extremely difficult to keep up this practice in normal project reality for most organizations