

Course "Softwareprozesse"

Agile Methods: eXtreme Programming (XP)

Lutz Prechelt

Freie Universität Berlin, Institut für Informatik

<http://www.inf.fu-berlin.de/inst/ag-se/>

- XP basic values
 - Communication, Simplicity, Feedback, Courage, Respect
 - XP practices
 - XP1 vs. XP2
 - Sit together, whole team, informative workspace, energized work, pair programming, stories,
- weekly cycle, quarterly cycle, slack, ten-minute build, continuous integration, test-first programming, incremental design
- Criticism
 - When (not) to use XP
 - Empirical results: a survey
 - XP and CMMI

Learning objectives

- Understand the basic idea of eXtreme Programming (XP) and where the name comes from
- Understand the values of XP
- Roughly understand the individual practices that make up XP
- Roughly understand when to and when not to use XP

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
- This is because many of its practices have become mainstream.
 - Many XP practices are used in 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 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 very 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

- The original, definitive source on XP is Kent Beck's book *"Extreme Programming Explained: Embrace Change"*, Addison-Wesley, 1999
- 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
 - Overview of the differences:
Stefan Roock: *"eXtreme Programming der zweiten Generation"*, <http://typo3.it-agile.com/fileadmin/docs/XP2.pdf>
- Ron Jeffries (xprogramming.com) uses a mix of both
- Many more books and articles have been written about XP



XP basic values

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

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

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 making the first three values a reality 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

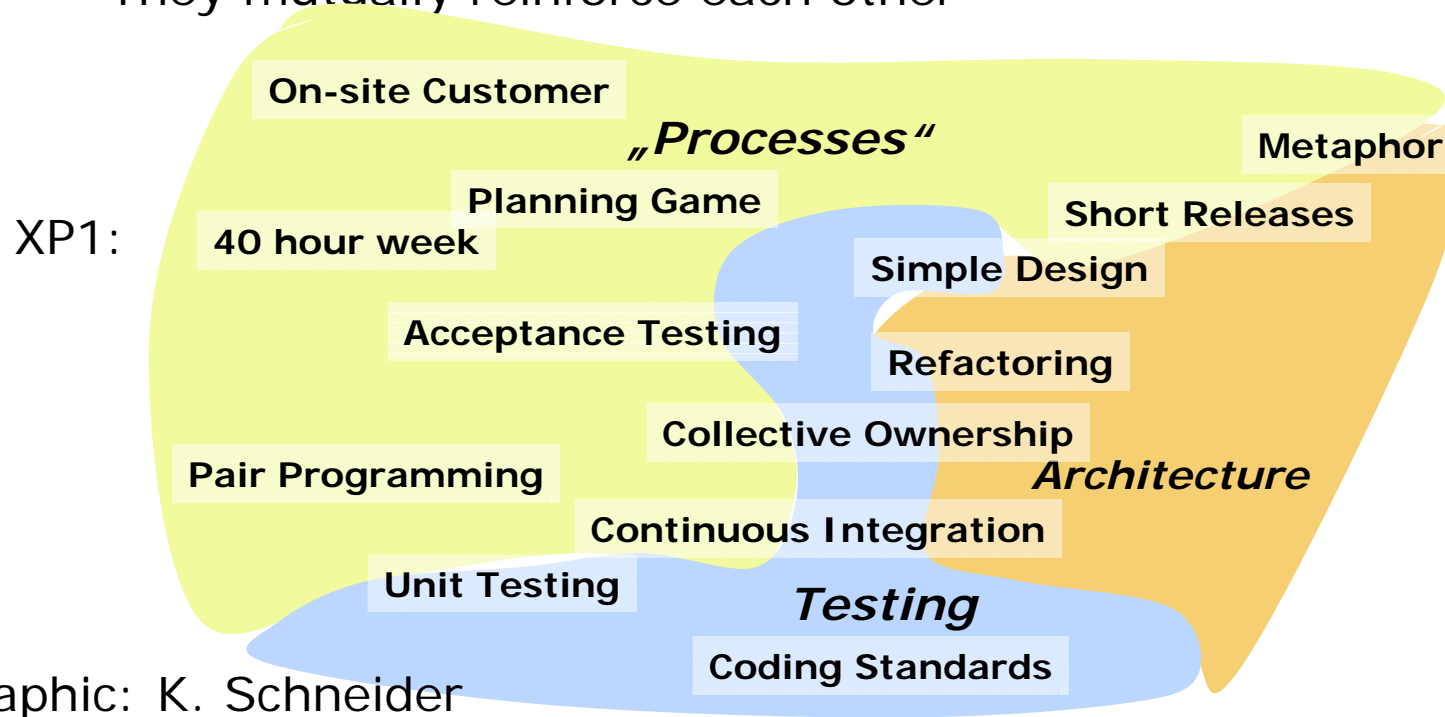


Basic values: Respect

- 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 1999 book

XP practices

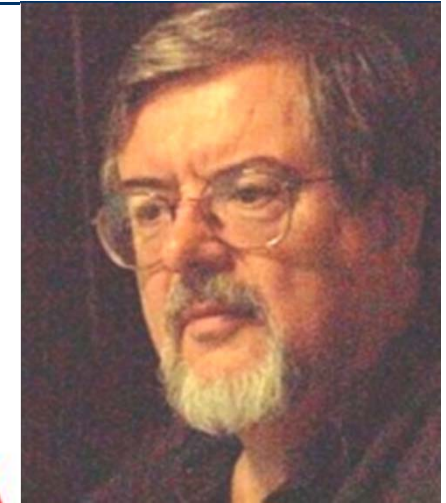
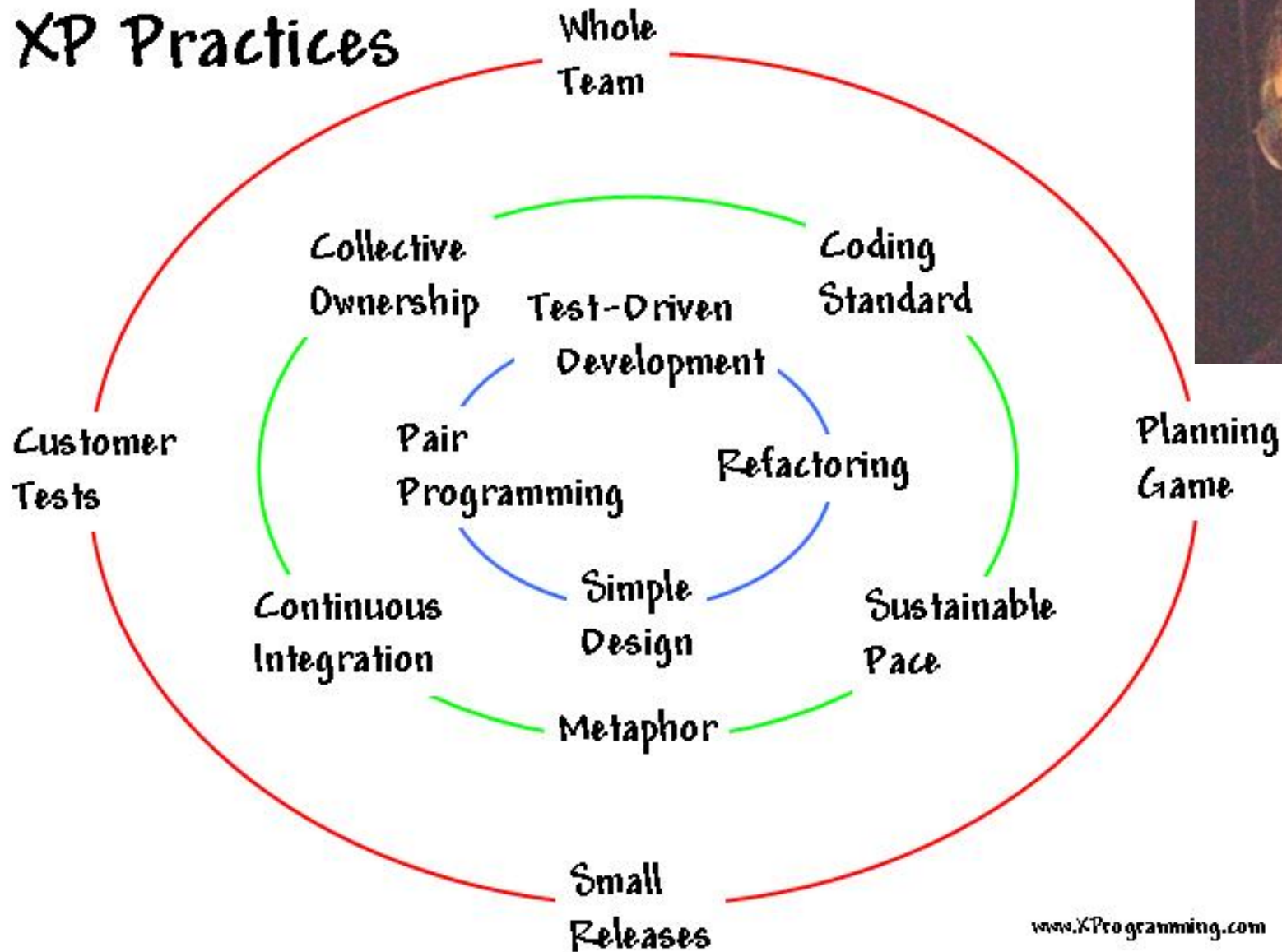
- XP as a method consists of a set of practices
 - Their manner of application can be adapted
 - but all of them are **mandatory** for a real XP process
 - although in practice very often not all are used.
 - Just picking your favorite five or so is not XP!
 - They mutually reinforce each other



Graphic: K. Schneider

Ron Jeffries' view (a mix of XP1 and XP2 practices)

XP Practices



Practices of XP and XP2

XP1 practices ("traditional"):

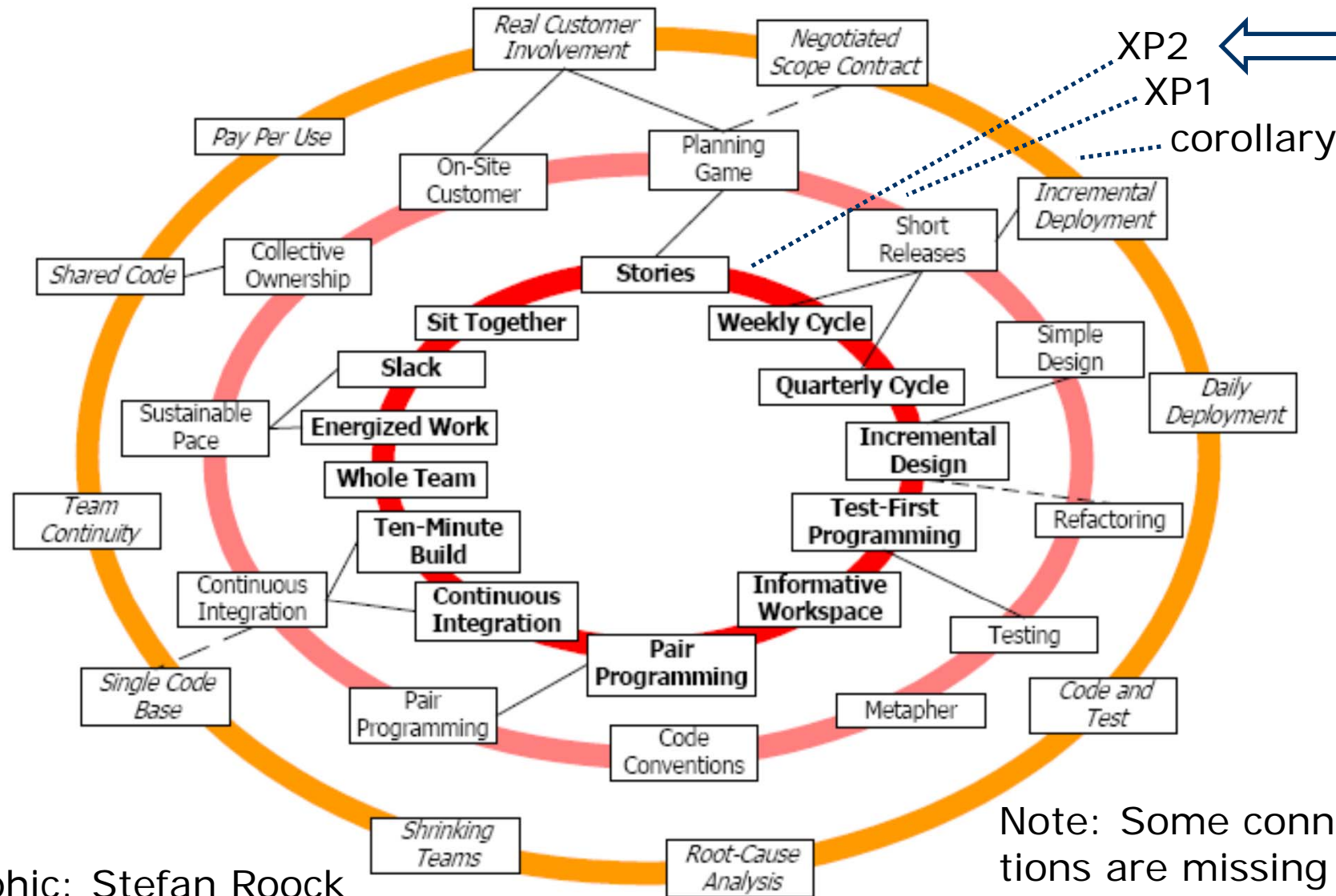
1. The Planning Game
2. Small Releases
3. Metaphor
4. Simple Design
5. Testing
6. Refactoring
7. Pair Programming
8. Collective Ownership
9. Continuous Integration
10. 40-Hour Week (sustain. pace)
11. On-Site Customer
12. Coding Standards

XP2 practices ("evolutionary"):

1. Sit Together
2. Whole Team
3. Informative Workspace
4. Energized Work
5. Pair Programming
6. Stories
7. Weekly Cycle
8. Quarterly Cycle
9. Slack
10. Ten-Minute Build
11. Continuous Integration
12. Test-First Programming
13. Incremental Design

Furthermore, XP2 has 11 "Corollary Practices"

The XP practices, old and new



Graphic: Stefan Roock

Practice: Sit Together

38% COMMON*

- 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
 - This also contributes to simplicity and courage
 - It greatly increases informal communication
 - by overhearing other pairs working
- Criticism:
 - 10 people in one room leads to high background noise and reduces concentration



COMMON?

Practice: Whole Team

- All qualifications and competences required should be represented in the team
 - this includes special technical knowledge
 - as well as business/requirements knowledge
 - (replaces and extends the former "on-site customer")
 - as well as project-level responsables (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



COMMON?

Practice: 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

COMMON?

- 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: Pair Programming (PP)

- All 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.
- One partner ("driver") uses keyboard and mouse
- Both "driver" and "observer" think about the design, any mistakes they've made, improvements etc.
- These roles may change frequently
 - e.g. every few minutes (but spontaneously)
- Pair composition should change frequently
 - e.g. twice a day



Practice: Pair Programming (2)

Criticism:

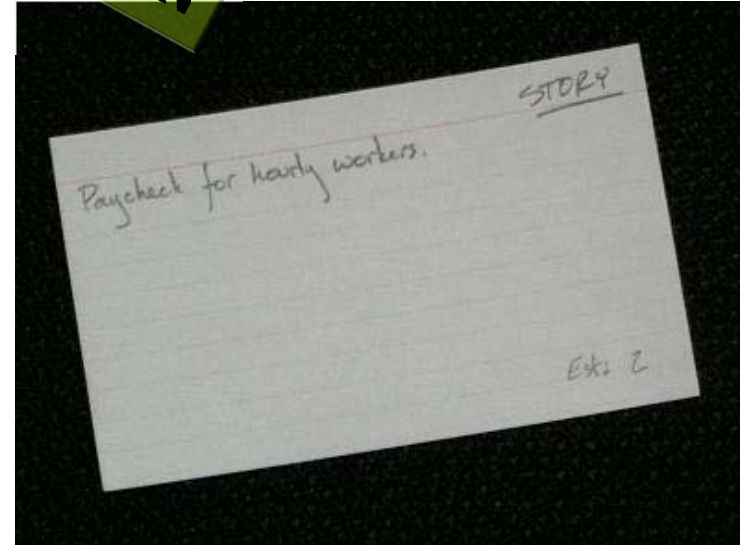
- For many kinds of task (in particular simple ones), PP may be rather inefficient
- There are a number of studies on this subject and the evidence is unclear:
 - Immediate productivity is lower than with two individual programmers, correctness and design quality are better,
 - but the secondary benefits are difficult to quantify
- Some programmers do not accept this style of working
- Pair partners may have incompatible working styles



Practice: Stories

74% COMMON

- 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

(COMMON)

- 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 new directions for the next requirements
- 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.
- Criticism:
 - It is extremely difficult to keep up this practice in normal project reality for most organizations

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



Practice: Continuous Integration

54% COMMON

- Developers check in their work into the common code base several times each day
- An automated process rebuilds the system after each such check-in and re-runs the system-level function tests
- 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 often an important alarm signal (indicator of bad project health)
- Criticism:
 - It is expensive or impossible to keep up a functional build during larger refactorings



38% COMMON??

Practice: Test-First Programming

- Before some program element is written (a class, a difficult 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 degree of test automation which is often inefficient

Practice: Incremental Design

COMMON

- 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 each time, the design is oriented more towards the current requirements, less to those just *expected* to come later
 - XP1 (misleadingly): "Use the simplest design that can possibly work"
 - 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 in the XP1 practice "Simple Design"



Note: Refactoring

48% COMMON

- 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
 - Martin Fowler: "*Refactoring: Improving the Design of Existing Code*", Addison-Wesley 1999
- XP allows courageous refactoring: the **automated tests** make it easy to verify whether a refactoring is correct
- Modern IDEs (such as Eclipse for Java) support or even automate several such refactoring operations



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"* (OA00)
- 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!

Note: Simplest design Option costs 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

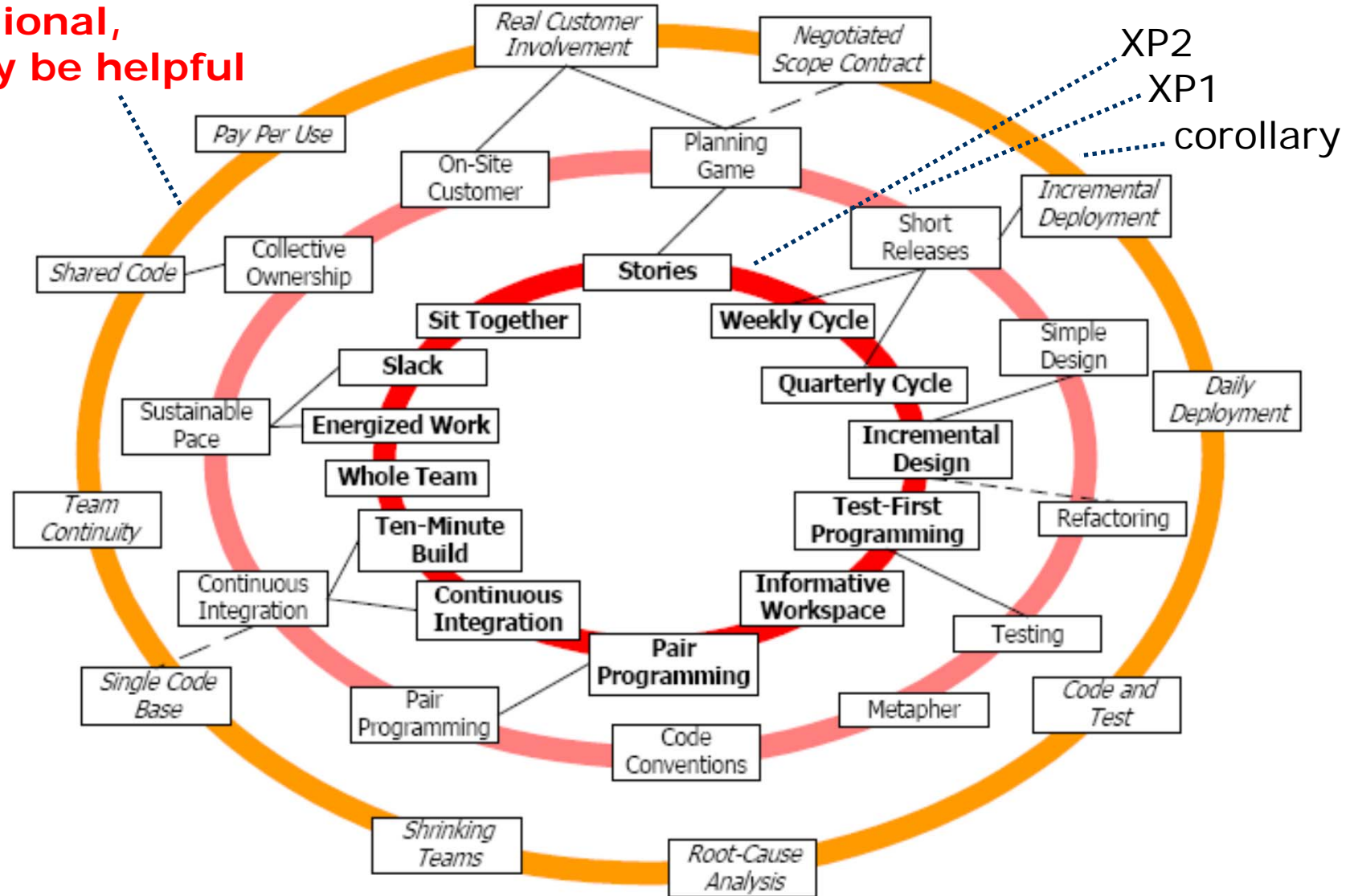
Assume you build **D'** anticipating a change A:

- 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!

The XP corollary practices

**Optional,
may be helpful**



- Gerold Keefer: "Extreme Programming Considered Harmful for Reliable Software Development 2.0",
<http://www.avoca-vsm.com/Dateien-Download/ExtremeProgramming.pdf>
(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

(These points are from Kent Beck's 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

- 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

XP roles

- Developer
 - the only role with always more than one representative
- Customer
 - usually (but not necessarily) a business-only person
- Tester
 - helps the customer write function tests
- Coach
 - responsible for process as a whole; guides the team to proper XP
- 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

A survey of XP projects

- Bernhard Rumpe, Astrid Schröder: *"Quantitative Untersuchung des Extreme Programming Prozesses"*, Technischer Bericht ViSEK/006/D, Dezember 2001

A survey of more-or-less-XP projects.

Characterization of respondents and projects:

- 47 respondents
 - reached via mailing lists, the XP 2001 conference, and direct contacts
- Location: 25% **US**, 20% **D**, 13% **CH**, 13% **UK**, 29% other
- Size (persons): 85% had **10 or fewer** (36% had 5 or fewer)
- Domain: 29% web, 16% financial, 16% tool, 38% other
- Language: 73% **Java**, 18% C++, 11% Smalltalk
- Customers: 56% had **more than one** group of customers

A survey of XP projects (2)

Main results:

- More than 90% of the projects were considered successful
 - although 51% used XP for the first time and although 51% had no external coach
 - but 69% had filled the coach role
 - although only 42% percent had teams of "all high" competence
 - although several were traditional projects in jeopardy that had been switched to XP
- 100% of XP users wanted to use it again
- Each practice was used by only some of the teams ("used" meaning **3...9** on a 0...9 usage intensity scale)
 - 98% Testing, 98% Refactoring,
 - 95% Simple Design, 91% Coding Standards
 - 89% Collective Ownership, 89% Pair Programming,
 - 85% Short Releases, 85% Continuous Integration,
 - 82% 40-hour week, 80% Planning Game,
 - **54%** Metaphor, **53%** On-site Customer

A survey of XP projects (3)

- Success factors and risks:
 - Testing was usually seen as a major success factor
 - as was Pair Programming.
 - 30% saw lack of on-site customer as a main project risk
- Perceived improvements due to XP:
 - 74% said their project was "much better" on schedule with XP than with previous methods
 - meaning answers +5 and +4 on a scale from -5 to +5
 - only 5% saw no improvement
 - 74% found work satisfaction "much better"
 - 74% found software quality "much better"
- Most difficulties were due to psychological barriers:
 - skeptical management,
 - refusal to send on-site customer,
 - developers not accepting Pair Programming

CMMI process areas in XP

M. Paulk: *"Extreme Programming from a CMM perspective"*, IEEE Software, Nov. 2001

- 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)

(not a part of CMM, only CMMI)

- + 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

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"*
 - (but other people would argue that XP is very strong on this)
 - 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

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.

- XP is a set of practices that mutually reinforce and support one another
- 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!