eXtreme Programming (XP)

L.EIC-ES-2021-22

Ademar Aguiar, J. Pascoal Faria

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Outline

- Introduction
- XP Values
- XP Practices
- Putting it all together
- Conclusions
- References and further reading

Introduction

Agile methods

- Dissatisfaction with the **overheads** involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
  - Focus on the code rather than the design
  - Are based on an iterative approach to software development
  - Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.
- The aim of agile methods is to **reduce overheads** in the software process (e.g. by limiting documentation) and to be able to **respond quickly to changing requirements** without excessive rework.

Agile manifesto (2001)

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

http://agilemanifesto.org/

eXtreme Programming (XP)

- One of the first agile methods
- Developed by Kent Beck in the late 90's:
  - XP is "a light-weight methodology for **small to medium-sized teams** developing software in the face of **vague or rapidly changing requirements.**"
  - Kent Beck came to XP 2018 in FEUP!

- Alternative to "heavy-weight" software development models (which deal poorly with changes)
  - "Extreme Programming turns the conventional software process sideways. Rather than planning, analyzing, and designing for the far-flung future, XP programmers do all of these activities a little at a time throughout development."
    -- *IEEE Computer , October 1999*

XP motto: embrace change

- In traditional software life cycle models, the cost of changing a program rises exponentially over time
  - Why would it cost more to make large changes during testing than during requirements specification?
- A key assumption of XP is that the cost of changing a program can be hold mostly constant over time
- Hence XP is a lightweight (agile) process:
  - Instead of lots of documentation nailing down what customer wants up front, XP emphasizes plenty of feedback
  - Embrace change: iterate often, design and redesign, code and test frequently, keep the customer involved
  - Deliver software to the customer in short iterations
  - Eliminate defects early, thus reducing costs

Early successes in industry

- Chrysler Comprehensive Compensation system
  - After finding significant, initial development problems, Beck and Jeffries restarted this development using XP principles
  - The payroll system pays some 10,000 monthly-paid employees and has 2,000 classes and 30,000 methods, went into production almost on schedule, and is still operational today (Anderson 1998)
- Ford Motor Company VCAPS system
  - Spent four unsuccessful years trying to build the Vehicle Cost and Profit System using traditional waterfall methodology
  - XP developers successfully implemented that system in less than a year using Extreme Programming (Beck 2000).

XP Values

Four Core Values of XP

- Communication
- Simplicity
- Feedback
- Courage

Communication

- What does lack of communication do to projects?
- XP emphasizes value of communication (namely face to face communication) in many of its practices:
  - On-site customer, user stories, pair programming, collective code ownership, daily standup meetings, etc.
- XP employs a coach whose job is noticing when people aren't communicating and reintroduce them

Simplicity

- "**Do the simplest thing that could possibly work**" (DTSTTCPW) principle
  - Elsewhere known as KISS (Keep It Simple, Stupid)
- A coach may say DTSTTCPW when he sees an XP developer doing something needlessly complicated
- "**You ain't gonna need it**" (**YAGNI**) principle
- Simplicity and communication support each other (how?)

Feedback

- Feedback at different time scales
- Unit tests tell programmers status of the system
- When customers write new *user stories,* programmers estimate time required to deliver changes
- Programmers produce new releases every 2-4 weeks for customers to review

Courage

- The courage to communicate and accept feedback
- The courage to throw code away (prototypes)
- The courage to refactor the architecture of a system

XP Practices

Twelve XP Practices

- Planning Game*
- Small Releases*
- System Metaphor
- Simple Design
- Test-driven Development
- Refactoring
- Pair Programming
- Collective Code Ownership
- Continuous Integration
- Sustainable Pace
- On-Site Customer
- Coding Standards

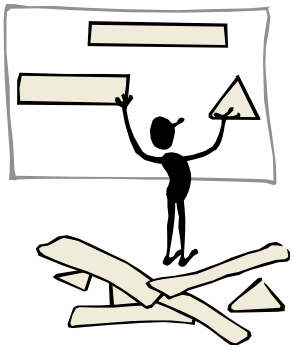
** First two practices are common to Scrum, but Scrum only covers project management and requirements management, whilst XP also covers development practices*

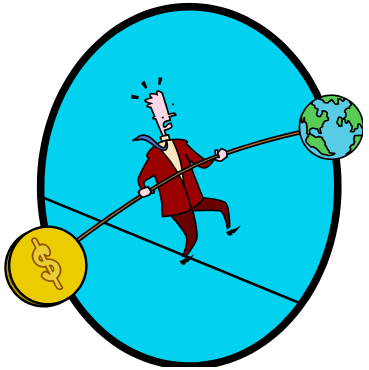
The Planning Game (1/2)

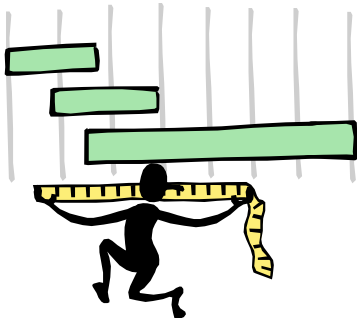
- Customer comes up with a list of desired features
- Each feature is written out as a **user story**
  - Describes in broad strokes what the feature requires
  - Typically written in 2-3 sentences on 4x6 story cards
- **Developers estimate** 'size' of each story (effort to implement)
- **Project velocity** = total size done per iteration
  - This is important to select the user stories for next iteration
- Given developer estimates and project velocity, the **customer prioritizes** which stories to implement
  - Why let the customer (rather than developer) set the priorities?

The Planning Game (2/3)

- Customers decide …
  - Scope (user stories)
  - Priorities
  - Release content
  - Delivery dates
- Developers decide …
  - (Effort) Estimates
  - Consequences
  - Process
  - Detailed (task) schedules







"The practices support each other.
The weakness of one is covered by the strengths of others."
– Kent, 1999

The Planning Game (3/3)

- Example of a user story
  - As a student,
  - I want to indicate preferences for colleagues to share the same scholar timetable,
  - So that I can be more productive in group works.

- Two planning processes:
  - Release planning – customers and developers decide about release content (user stories) and date for the next releases
  - Iteration planning – developers decide about development tasks for the next iteration (possible iterations per release)

Small Releases

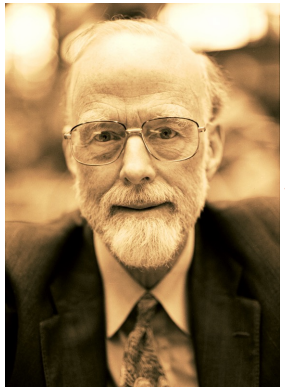
- Start with the smallest useful feature set
- Release early and often, adding a few features each time
- Releases can be date driven or user story driven

System Metaphor

- The system metaphor is a story that everyone - customers, programmers, and managers - can tell about **how the system works**.
- Role somewhat similar to architectural styles/patterns
- Example: Customer service application.
  - Customers call in to complain for problems. Representatives take information from customer, and ensure that technicians work on the problem.
- System Metaphor: **Assembly line**.
  - Think of problem reports and solutions as the Assembly, and the technicians as Workers at Stations (possibly several workers needed to solve a problem).

Simple Design

- Use the simplest possible design that gets the job done
- The requirements will change tomorrow, so only do what's needed to meet today's requirements (remember YAGNI)
- Avoid big up-front design



Tony Hoare, Turing Award Lecture, 1980

I conclude there are two ways of constructing a software design:

one way is to make it so simple that there are obviously no deficiencies,

and the other way is to make it so complicated that there are no obvious deficiencies.

Test-driven Development

- **Test first:** before adding a feature, write a test for it!
  - If code has no **automated** tests, it's assumed it doesn't work
  - Also create test for bugs discovered, before fixing them
- **Unit Tests** (or **developer tests**): testing of small pieces of functionality as developers write them
  - Usually for testing single methods, classes or scenarios
  - Usually automated with a unit testing framework, like JUnit
  - Experiments show that TDD reduces debugging time
- **Acceptance Tests** (or **customer tests**): specified by the customer to check overall system functioning
  - A user story is complete when all its acceptance tests pass
  - Usually specified as scripts of UI actions & expected results
  - Ideally automated with a UT or AT framework, like FIT

Refactoring

- Recently updated to “Design Improvement”
- Refactoring = improve the structure of the code without changing externally visible behavior
- E.g., refactor out any duplicate code generated in a coding session
- Refactoring and automated tests go hand-in-hand (why?)
- Simple design and continuous design improvement (with refactoring) go hand in hand (why?)

Pair Programming

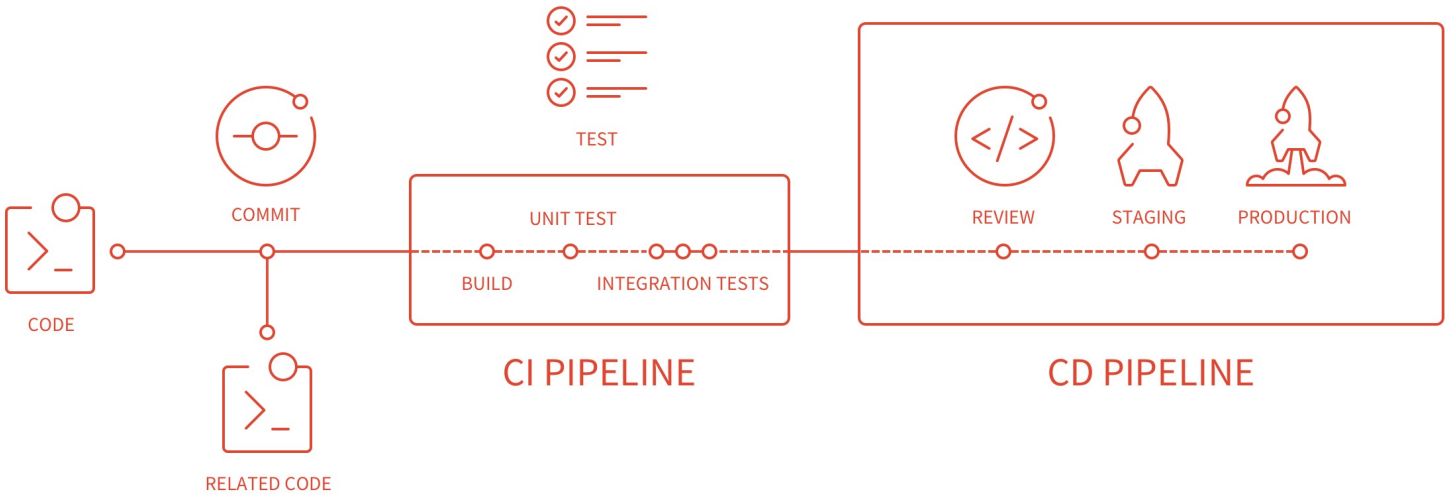
- How it works:
  - Two programmers work together at one machine
  - Driver enters code, while navigator critiques it
  - Periodically switch roles and pairs
  - Requires proximity in lab or work environment
- Advantages:
  - Serves as an informal review process
  - Helps developing collective ownership and spread knowledge
  - Improves quality (less defects, better design), whilst maintaining (or improving) productivity
- Note: If you don't do XP/PP, at least do peer reviews

Collective Code Ownership

- What it means:
  - No single person "owns" a module
  - Any developer can work on any part of the code base at any time
- Advantages:
  - No islands of expertise develop
  - All the developers take responsibility for all of the code
  - Pressure to create better quality code
  - Change of team members is less of a problem

Continuous Integration

- All changes are integrated into the code base at least daily
  - As opposed to “big-bang integration”
- Tests have to run 100% before & after integration
- Enables frequent releases



GitLab continuous integration & continuous delivery pipeline

Sustainable Pace

- Aka “40 hours work a week”
- Programmers go home on time
- “Fresh and eager every morning, and tired and satisfied every night”
- In crunch mode, up to one week of overtime is allowed
- More than that and there’s something wrong with the process

On-Site Customer

- Recently updated to “Whole team”
- Development team has continuous access to a real live **customer**, that is, someone who will actually be using the system, or a **proxy**
  (in Scrum: product owner)

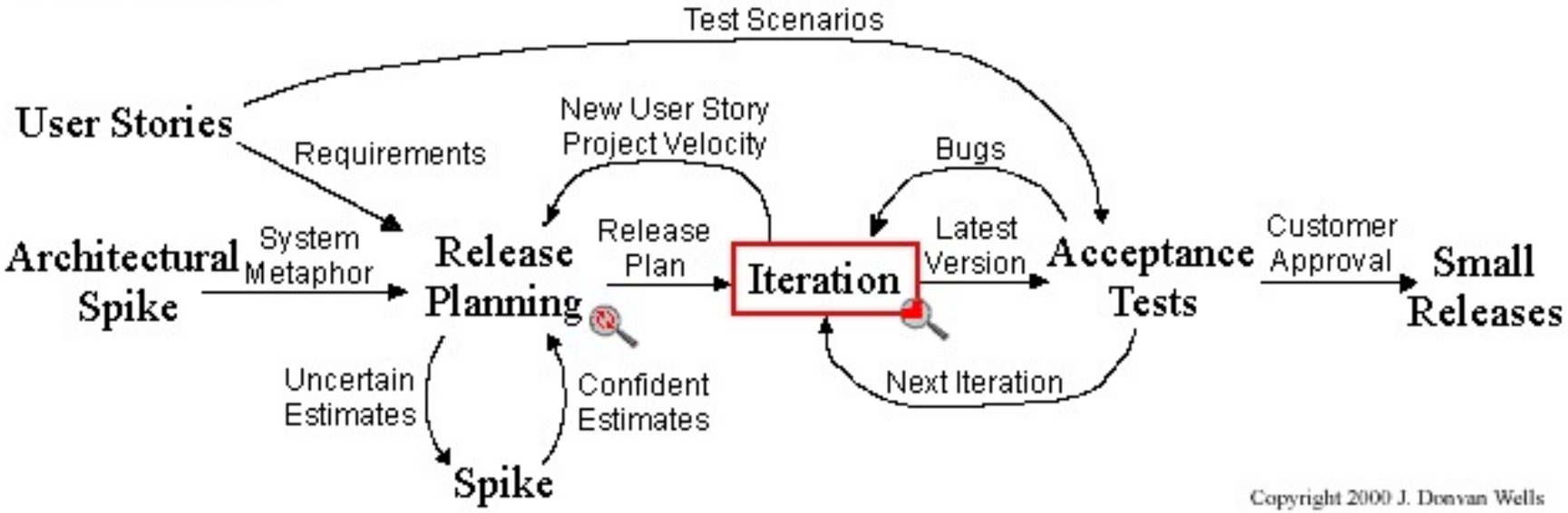
- Why is this important?
- Why is this difficult?

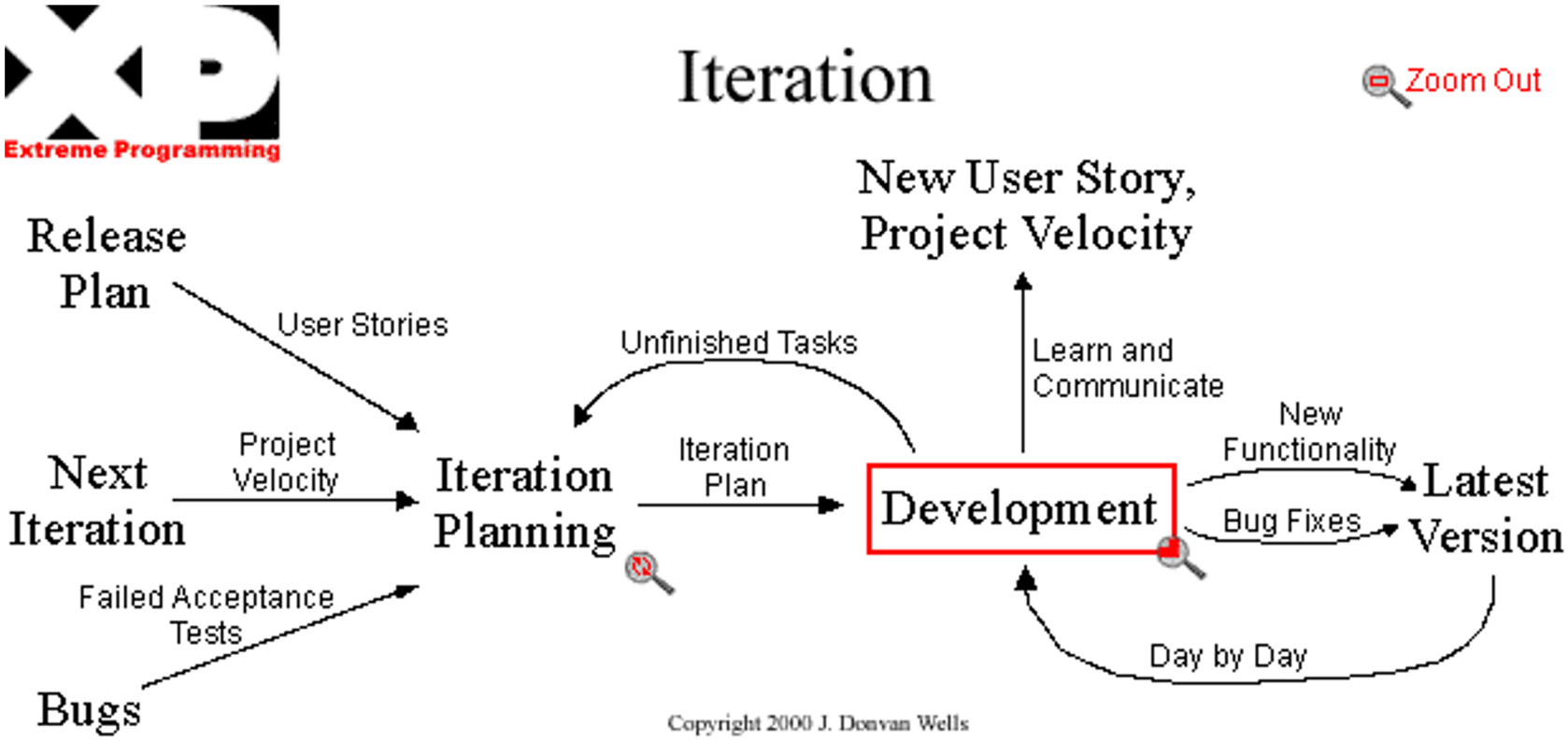
Coding Standards

- Everyone codes to the same standards
- Coding standards typically cover:
  - naming conventions, file organization, indentation, comments, declarations, statements, white space, ...
- Ideally, you shouldn't be able to tell by looking at it who on the team has touched a specific piece of code
- Coding standards are important because:
  - 80% of the lifetime of a piece of software goes to maintenance
  - Hardly any software is maintained for its whole life by the original author
  - Coding standards improve software readability, allowing engineers to understand new code more quickly & thoroughly

Putting it all together

Extreme Programming Project





Release planning: Relase content & date

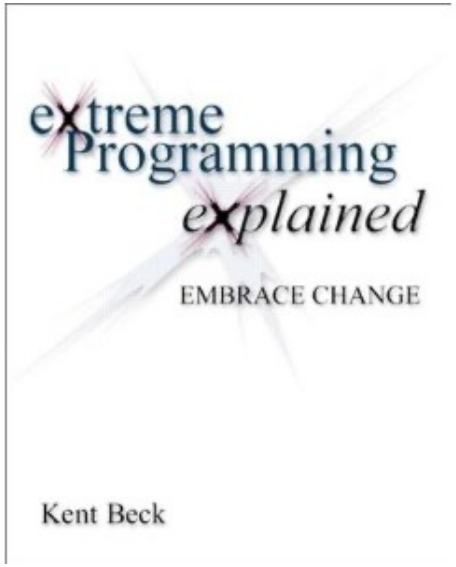
Conclusions

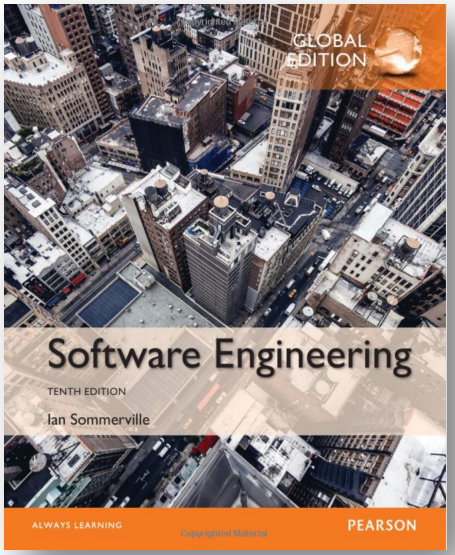
- Extreme programming is a well-known agile method that integrates a range of good engineering and management practices such as frequent releases of the software, continuous design improvement and customer participation in the development team.

- A particular strength of extreme programming is the development of automated tests before a program feature is created. All tests must successfully execute when an increment is integrated into a system.

References and further reading

- Kent Beck, Extreme Programming Explained (Addison Wesley 2000).
- Ian Sommerville, Software Engineering, 10th edition (chapter 3)
- http://www.extremeprogramming.org
- http://wiki.c2.com/?ExtremeProgramming





aaguiar@fe.up.pt, jpf@fe.up.pt