Information Systems Concepts

eXtreme Programming

Roman Kontchakov

Birkbeck, University of London

Based on Chapter 21 of Bennett, McRobb and Farmer:

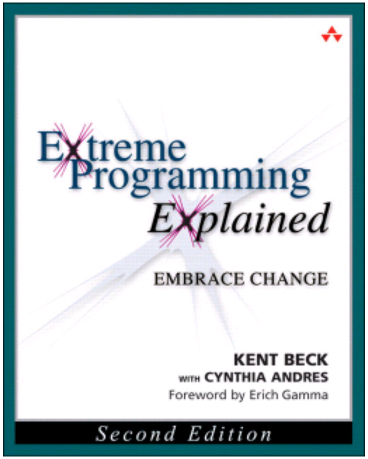
Object Oriented Systems Analysis and Design Using UML, (4th Edition), McGraw Hill, 2010

Outline

- XP
  - Section 21.6 (pp. 625–627)

What is XP?

eXtreme Programming (XP) is probably the most prominent **agile** development methodology (first publicized by Kent Beck)



eXtreme

Young Programmer: Wow! Cool! Let´s try it!

Senior Manager: Isn´t it silly, tricky and dangerous?

What is XP?

XP is a novel combination of
best practices in software development

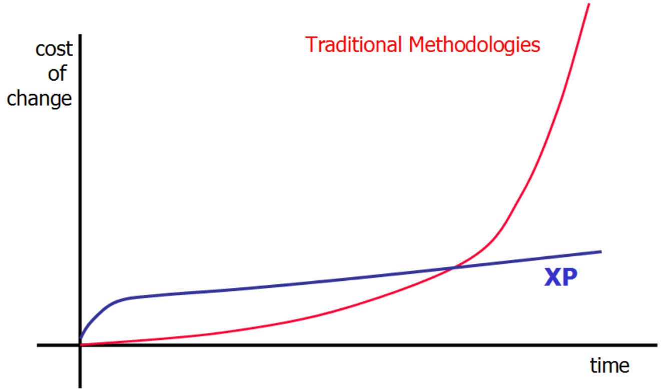
each knob is a ‘best practice’
known to work well
in software development

turn all knobs up to 10 (the extreme)
and leave out everything else

Embracing Change

XP emphasises that embracing **change** is important and key to systems development

- everything in development changes: business, technology, team, requirements, etc.



- the problem is not change, because change is going to happen anyway
- the problem is our inability to cope with change

Underlying Principles: 1. Communication

Poor communication is a significant factor in failing projects

XP highlights the importance of **good communication**

- among developers
- between developers and users

Developers work in open workspaces and rely on oral communication

Underlying Principles: 2. Simplicity

Developers are sometimes tempted to use technology
for **technology's sake**
rather than seeking the simplest effective solution
and justify complex solutions as
a way of meeting possible future requirements

XP focuses on the **simplest solution**
for the immediate known requirements

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."

Bill Gates

Underlying Principles: 3. Feedback

Unjustified optimism is common in systems development

- Developers tend to underestimate the time required to complete any particular programming task
- This results in poor estimates of project completion, constant chasing of unrealistic deadlines, stressed developers and poor productivity

XP is geared to giving the developers **frequent** and **timely feedback** from users and from test results

Work estimates are based on the work actually completed in the previous iteration

Underlying Principles: 4. Courage

The exhortation to be courageous urges the developer to **throw away** code that is **not quite correct** and start again, rather than trying to fix the unfixable

Essentially the developer has to abandon unproductive lines of code, despite the personal emotional investment in work done

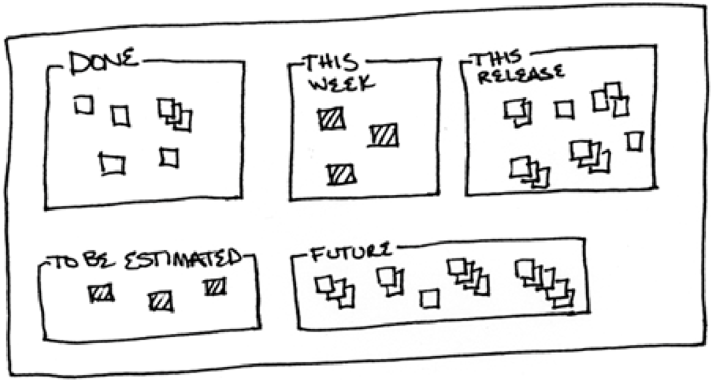
12 Best Practices

- Planning Game
- Small Releases
- Metaphor
- Simple Design
- Testing
- Refactoring
- Pair Programming
- Collective Ownership
- Continuous Integration
- 40-Hour Week
- On-Site Customer
- Coding Standards

Best Practices:
1. Planning Game

The planning game involves quickly defining the scope of the next release from user priorities and technical estimates

The plan is updated regularly as the iteration progresses



User Stories

Requirements capture in XP is based on **user stories** that describe the requirements

- written by the user
- form the basis of project planning and the development of tests

A user story is very similar to **use cases**, though there are key differences in granularity:

- A typical user story is about three sentences long and does not include any detail of technology
- When the developers are ready to start work, they get detailed descriptions of the requirements by sitting face-to-face with their customer

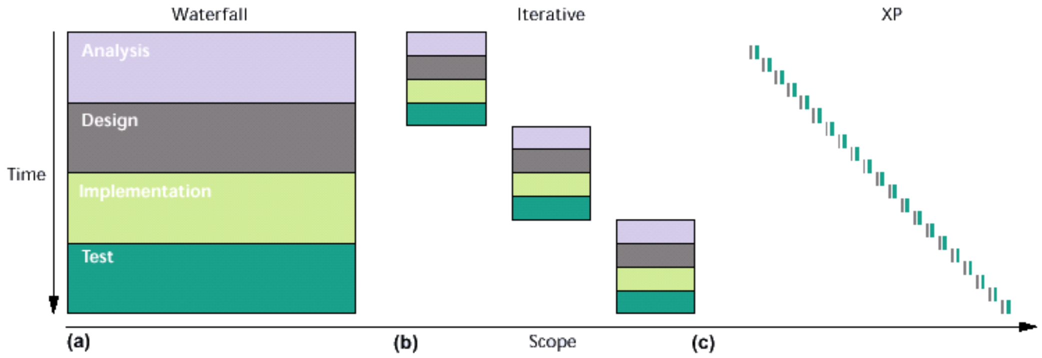
User Story: Examples

- As a user, I want to search for my customers by their first and last names.
- As a non-administrative user, I want to modify my own schedules but not the schedules of other users.
- Starting Application: The application begins by bringing up the last document the user was working with.
- As a user closing the application, I want to be prompted to save if I have made any change in my data since the last save.

Best Practices:
2. Small Releases

The information system should be delivered in small releases that incrementally build functionality through rapid iteration

- Each release is as small as possible, but still delivering business value
- Get customer feedback early and often



Best Practices: 3. Metaphor

A unifying metaphor or **high-level shared story** focuses the development

- provides the view from 10k feet above
- helps to guide the team, e.g., when naming objects
- visualizes the information system in terms of something both simple and concrete

For example, in the C3 payroll project:
the paycheque goes down the assembly line and pieces of information are added to it

Best Practices: 4. Simple Design

The system should be based on a **simple design**

No Big Design Up Front (BDUF)

Do the simplest thing that could possibly work

You Ain't Gonna Need It (YAGNI)

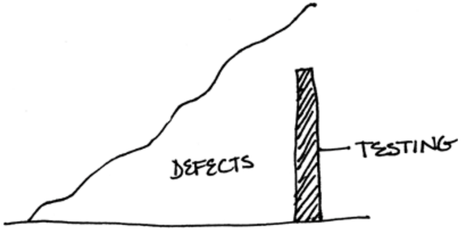
Best Practices: 5. Testing

Developers prepare **unit tests** in advance of software construction

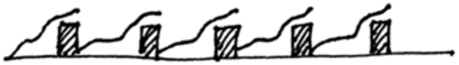
- Test-Driven Development (TDD)
- tests are automated, often using the xUnit framework
- must run at 100% before proceeding

Customers define **acceptance tests**

- written together with the customer
- acts as 'contracts'
- a measure of progress



late, expensive testing leaves many defects



frequent testing reduces costs and defects

Best Practices: 6. Refactoring

Restructuring the existing program code
(without changing its functionality) to

- **remove duplication**
- **simplify** the code
- improve **flexibility**

Refactor Mercilessly:

XP developers can rely on testing to ensure nothing breaks in the process of refactoring

Best Practices: 7. Pair Programming

- Two developers write code together using one workstation
- One developer, the **driver**, has control of the keyboard and mouse and creates the implementation
- The other developer, the **passenger**, watches the driver's implementation to identify **defects** and participates in on-demand brainstorming
- The roles of driver and passenger are periodically rotated between the two developers

Best Practices: 7. Pair Programming

- An empirical study by Laurie Williams (NCSU)
  - Pairs produced higher quality code
    15% less defects
  - Pairs completed their tasks in about half the time
    58% of elapsed time
  - Most programmers reluctantly embark on pair programming
    pairs enjoy their work more (92%)
    pairs feel more confident in their work products (96%)
- India Technology Company
  - 24% increase in productivity (kloc/person-month)
  - 10-fold reduction in defects

Best Practices: 8. Collective Ownership

The code is owned collectively and

anyone can change any code

- Cleaner code

  developers are not required to work around deficiencies in objects they do not own

- Faster progress

  no need to wait for someone else to fix something

Best Practices: 9. Continuous Integration

The system is integrated and **built frequently** each day
This gives the opportunity for regular testing and feedback

- Do it often

  integrate and test every few hours, at least once per day
- All tests must pass

  easy to tell who broke the code
- Only one pair integrates code into the base at a time

  simplifies issues arising from parallel integration
- Eliminates need for an integration team

Best Practices:
10. 40-hour Week

Normally staff should work no more than **40 hours a week**

Kent Beck: "...fresh and eager every morning,
and tired and satisfied every night"

Burning the midnight oil kills performance

Tired developers make more mistakes,
which slows you down more in the long run

If you mess with people's personal lives (by taking it over),
in the long run the project will pay the consequences

Best Practices: 11. On-Site Customer

A user should be a **full-time member** of the team

The customer available on site clarifies user stories and makes critical business decisions

- developers don't make assumptions
- developers don't have to wait for decisions

Face to face communication minimizes the chances of misunderstanding

Best Practices: 12. Code Standards

All developers should write code according to **agreed standards** that emphasize good communication through the code

- **Consistency** saves time and money
  - makes it easier to understand other people's code
  - avoids code changes because of syntactic preferences
- The code should be **intention-revealing**
  - if you can't explain your code with a comment, rewrite it
  - if your code needs a comment to explain it, rewrite it

Using XP

- The effectiveness of XP comes from using the 12 best practices **together**

  You can use a practice outside of the XP context, but you will not receive maximum benefits in productivity or quality

- XP is not sympathetic to using UML for system analysis and design

  The code itself is its own design documentation

- XP relies on clear **communicative** code and rapid **feedback**

  If these are not possible then XP would be problematic

- XP is best suited to projects with a relatively **small** projects (e.g., no more than 10 programmers)

Take Home Messages

- XP
  - 4 Underlying Principles
  - 12 Best Practices