Introduction to eXtreme Programming
September 5, 2001

Developing software with
eXtreme Programming (XP): what, why, how?

Björn Eiderbäck
NADA, KTH
Email: bjorne@nada.kth.se

Contents
Traditional methods
Problem
XP

Today we especially look at
XP overview: slides 19-24, 26
Testing and JUnit: slides 48-49
We’ll come back to rest later on in the course

What, why, how?
We take a brief look at some central programming routines
Refactoring
Testing
Continuous integration

Traditionally

• A brief look at the “traditional” way of developing software
  – Methodology
  – History
  – Problems
  – ...

© Björn Eiderbäck 2001
Construction of Software

- How could we develop the system to make it:
  - Useful
    - So it does what the customer wants
  - Reliable
    - It should not fail
  - Flexible
    - It should be adaptable and changeable
  - Not to expensive
  - Available
    - Easy to use
  - Maintainable
    - It should be easy to implement new or changed requirements

How should we develop the system?

- Problems
  - Hard to identify what the client wants
  - Hard to describe large systems
  - The human’s cognitive ability is limited

- Solution
  - Find means to communicate with customers
    - Scenarios
    - Prototyping
  - Analysis Phase
    - Find out the real requirements and develop a structure (of the whole system)
Development and methodology

- The type of the project could influence the development process
  - Programming in the small
    - The software is written by one or few developers. A single person could be know the whole project.
    - Main problem (software): design and develop algorithms
  - Programming in the large
    - The software is written by a big team. Some persons could make the analysis, other people design, the integration is perhaps made by a third group, (all) the testing is made by yet another group, etc. No single person could know the whole project.

Focus on the interface first

- Scenarios
  - To describe required functionality
- Prototypes
  - To try out ideas and possibilities
- Use cases
  - To explore and formalize scenarios to be able to among other things relate functionality to components in the software
- The Software
  - It has been proven many times that it is best to develop a system in accordance to the components’ APIs rather than their behavior
    - Then it is easier to change or replace a module or component
Abstract

- Remove details and focus on the things of most importance
- While you are abstracting some details are lost
  - But this is the purpose…
- We abstract to understand, divide the problem (into manageable chunks), and to get a conceptual model of the system
- The degree of the abstraction could vary in different parts of the system
  - The analysis phase is more abstract than the design phase which is more abstract than the construction phase or the code

History Methodologies ...

- Problems
  - Hard to develop software
  - 80% maintenance
  - We want formalism, but simple and useful
  - Communication
  - The cost of change
- The 1960’s
  - Programmers developed software any way they could
    - Sometimes the intention was to make the programs as hard to understand as possible
    - Dijkstra reacted to this 1968, *GOTO Statement Considered Harmful*
- The 1970’s
  - Flow diagrams
  - Structured programming
  - Top-down, bottom-up or middle-out
- Many object-oriented methodologies become popular during the 80’s
  - OMT, ObjectOry, Booch, Shlaer-Mellor, Coad-Yourdon
  - …
…UML…

- Unified Modeling Language UML
  - 1990’s
  - Unification of three dominating methodologies/notations
    - OMT, ObjectOry OSE, Booch
  - ”Standard”, OMG
  - More a notation than a methodology (so far..)
  - Not (to) stringent = useful

The Process

- Requirement analysis – describe and validate

- The Analysis phase – find out and develop the structure of the system

- The Design phase – describe more concretely how the system should be realized

- Implementation – implement the system and perform unit tests

- Testing – verify the system
  - Acceptance tests, functional tests, …
A lot of things to do, but what? Not well-defined how!

- The procedure
  - Find potential actors
    - Give the respective actor a name and a short description
    - Limit the system
  - Make a dictionary (to formulate and describe a vocabulary)
  - For every actor: Find use-cases
    - Give each use-case a name and a short description
  - Evaluate the actors and use-cases and iterate
    - Are there some missing actors or use-cases? Duplicates?
  - Identify commonalities, structure the model, iterate
  - Describe each use-case
  - Examine the descriptions and iterate
    - Are there any missed defective functionality?
  - Examine, validate and approve the model

Traditionally

Collect background information and data
- Requirement specifications, wishes, description of the current business or the existing system, interviews. Define the domain of the problem.

Define use-cases
- I.e describe how the system will be used

Search for object-candidates
- For instance by means of CRC-cards or some other brainstorming technique [see slides 17-19]

Classify objects
- Class-names, responsibilities, and maybe important attributes and methods

Relations among objects
- By means of class and object diagrams

Evaluation
- Desk top test where uses-case, relations, and so forth are manually tested. Are the chosen names okay?

Document the analysis phase
- By means of diagrams and test protocols
We also have to work with different perspectives

- **Conceptual**
  - In this perspective we draw diagrams describing concepts in the domain. These concepts are often realized by classes, but quite often not.
  - The hardware and the software that should be used to implement the system have none or minor impact on the construction of the conceptual model.

- **Specification**
  - In this perspective we focus on the interface (i.e. APIs) of the components, rather than how they are implemented (in detail). We are looking at types instead of classes.

- **Implementation**
  - In this perspective we develop the classes and the implementation becomes obvious.

The Waterfall model

Traditional idealised model of the development process

- Analyze
- Design
- Implement
- Testing
- Maintenance
By Cunningham and Beck, the mid 80’s.
- Developed as a means to teach object-oriented programming
- Gave components physical appearances

- Very good for brainstorming

- Process:
  - Write *class-names on cards*, but don’t make any selections among the classes now (all classes are good classes!)
  - After a while, when there are plenty of classes, *choose* ”the best ones”
  - Identify *responsibilities and behavior* for each class
  - Identify *collaborations between classes*
  - Try to organize classes *hierarchically* and identify *abstract classes*
• An empty CRC-card

![CRC-card diagram]

| Class-name | Responsibilities | "Collaborators"

### Registry

- Handle a register of Persons
- Persons can be added or removed
- The register could be sorted by name, birthday, or address

<table>
<thead>
<tr>
<th>Person</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>superclass: Object</td>
<td>subclass: Student</td>
</tr>
<tr>
<td>Handles info about a person: name, address, telephone, birthday</td>
<td></td>
</tr>
</tbody>
</table>

| Student | |
|---------| |
| superclass: Person | subclasses: |
| Handles email and status of courses | |
**Problems**

- Traditional methods doesn’t solve all problems (and perhaps even add new ones)

- Not even the developers of the methodologies (always!?) use them themselves

- Therefore the interest for more light-weight methodologies has increased significantly

**The Basic Problem: Risk**

<table>
<thead>
<tr>
<th>Risk Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule slips</td>
<td>The software is not ready at the expected date</td>
</tr>
<tr>
<td>Project canceled</td>
<td>After numerous slips the project is canceled</td>
</tr>
<tr>
<td>System goes sour</td>
<td>After a couple of years the cost of making changes or the defect rate rises so much that the system must be replaced</td>
</tr>
<tr>
<td>Defect rate</td>
<td>The production system is not used since the defect rate is too high</td>
</tr>
</tbody>
</table>

...which XP tries to solve
...the problems... (are more!)

- Business misunderstood
  - The software in production solves the wrong problem

- Business changes
  - The business problem the software is designed to solve is replaced

- False feature rich
  - The software includes a lot of features which were fun to program, but none of which makes the customer much money

- Staff turnover
  - The programmers start to hate the program and leave

...other problems

- Traditionally there are several teams working on the system
  - Some analyze, others design, and yet others construct and code

  - The ones that only analyze easily lose their connection to the reality

  - Usually could only parts of the analyze be used (a lot of problems remains)
### XP: an overview

**eXtreme Programming: an overview**

- **eXtreme Programming (XP),** by Kent Beck
- **Methodology (the 12 practices)**
  - The Planning Game
    - Quickly determine the scope of the next release. Priorities
  - Small releases
    - Release new versions in very short cycles
  - Metaphor
    - Find a simple metaphor describing how the system works
  - Simple design
    - Make the design as simple as possible
  - Testing
    - Test the code continuously. Write the tests before the production code.
  - Refactoring
    - Restructure the code to remove duplications, improve communication, simplify, or add flexibility
  - Pair programming
    - Two programmers at one machine
  - Collective ownership
    - Everyone owns and can change any code anywhere in the system
  - Continuous integration
    - Integrate and build the system many times a day
  - 40-hour week
    - Don’t work more than 40 hours a week
  - On-site customer
    - Include a real user in the team
  - Coding standards
    - Use a coding standard to improve communication

### XP: the process, an overview

1. **The Planning Game**
   - The Customer writes stories
   - The Developers analyze, break apart, etc
   - How long will this story take?
   - Divide a story into (programming-) tasks
   - The iteration is planned with the Customer
   - What is most important?

2. **Programmers sign up for tasks** for one iteration

3. **Pair-programming**
   - Work in pairs, two programmers at one computer while writing production code.
   - Write tests first.
   - Do the simplest thing that works, that is only implement enough to pass the tests
   - Continuous integration, that is you should at all times have a working platform with all the pairs’ code (integrate several times a day…)

4. **Refactoring** to make the code do what it should and become as beautiful as possible. You strive for code that “speaks” itself!
See the following link for an illustration of the whole process:

http://www.extremeprogramming.org/map/project.html

**Workflow in the XP process; some basic principles**

- Program in pairs
  - Both contribute. One partner is steering some time while the other partner is co-driver. The “roles” are switched when suitable.

- The development is driven by tests
  - Write tests first. Production code is written afterwards.
  - Any feature without an automated test simply doesn’t exist.
  - Customers write functional tests.
  - All unit tests must work before you proceed.

- Pairs develops single tasks but also the whole system
  - Integration, tests, refactoring

- The Integration always follow the implementation of a certain task
  - All unit- and integration-tests must pass before the integration is ready
**XP’s solution to the problem: Risk**

- **Schedule slips**
  - XP uses short cycles and releases often

- **Project canceled**
  - XP asks the Customer to choose the smallest release that gives maximal value. So less could go wrong and the value of the software is greatest.

- **System goes sour**
  - XP creates and maintains a comprehensive suite of tests which are run and re-run after every change. Therefore the system is always kept in prime condition.

- **Defect rate**
  - XP tests from perspective of both programmers and customer.

---

**...XP’s solution to the problem**

- **Business misunderstood**
  - XP calls for the customer to be an integral part of the team.
  - The specifications are continuously refined during development.

- **Business changes**
  - XP shorten the release cycles, so there is less change of development during a single release.
  - During a release the customer could substitute to provide new functionality.

- **False features**
  - XP insists that only the highest priority tasks are addressed.

- **Staff turnover**
  - XP ask programmers to accept responsibility for estimating and completing their own work.
  - XP encourages human contacts among the team.
  - In short XP treats team members as intelligent species.
**We try to control four variables**

- **Cost**
  - More money solve some problems but soon creates new problems

- **Time**
  - More time to deliver can improve quality and increase scope
  - However too much time could hurt the project
  - Too little time and quality suffers, with scope not far behind

- **Quality**
  - By sacrificing quality make short time gains, but in the long run the cost is enormous

- **Scope**
  - Less scope makes it possible to deliver better quality
  - It also lets you deliver sooner or cheaper

---

**The cost of change**

- There is an old “truth” saying that the cost for changing increases exponentially in relation to time (see slide 9)

- In XP the cost of change seems not to rise dramatically over time, rather it flattens out
  - The reasons are
    - Continuous (automatic) testing
    - Do the simplest thing that could possible work
    - Only once
    - Continuous integration
    - Refactoring
  - Additionally, these habits make us used to regard change and modification of the design and system
**Four values**

- Communication
  - Developers are encouraged to communicate with each other
  - A lot of the practices encourage communication
    - Pair programming, testing, planning game, continuous integration,…

- Simplicity
  - What is the simplest thing that could possibly work?

- Feedback
  - Tests, stories, the system is well structured, simple, and self-explanatory

- Courage
  - Correct errors or shortcomings with great courage, don’t hesitate
    ("the rules” in concert give us this result!)

---

**Basic principles**

- Rapid feedback
  - It usually is good to have a short time between action and feedback

- Assume simplicity
  - Threat every problem as if it can be solved with ridiculous simplicity

- Incremental change
  - Solve problems as a series of the smallest changes that make a difference.
    Even big problems should be broken down an treated in this way

- Embracing change
  - The best strategy is the one that preserves the most options while actually
    solving your most pressing problem

- Quality work
  - Everybody likes doing a good job
**Basic elements**

- **Coding**
  - At the end it is only the code that counts, independently of how many diagrams that are produced

- **Testing**
  - We don’t know if anything works unless we test it
  - The tests make us think on a certain problem a little bit differently

- **Listening**
  - Listen to customers, domain experts, and so forth

- **Designing**
  - To get a well structured and working system we must, of course, design in XP as well

---

**Why does XP work?**

- Some motivations to each rule (mainly after Beck)

- Kent Beck tell us that in principle all the rules are needed to be applied together to get everything working
  - However, “rules” as write tests first, simple design, refactoring, and continuous integration are valuable even if not XP, or the rest of the rules, are applied
  - And the ”rule” pair programming have been proven beneficial even if used in isolation
The Planning Game

- You couldn’t possibly start development with only a rough plan? You couldn’t constantly update the plan?

- Unless
  - The customer did the updating of the plan themselves
    - Based on estimates provided by the programmers
  - You had enough of a plan at the beginning to give the customer an idea of what was possible over the next couple of years
  - You made short releases so any mistake in the plan would have very short impact
  - Your customer was sitting with the team, so they could spot potential changes and opportunities for improvements quickly

Short Releases

- Presumptions:
  - You couldn’t possible go into production after a few months.
  - You certainly couldn’t make new releases of the system on cycles ranging from daily to every couple of months

- Unless
  - The Planning Game helped you to focus on the most essential
  - You integrate continuously, so the cost of packaging release was minimal
  - Your testing reduced the defect rate enough so you can avoid lengthy test sequences
  - You could make a simple design, sufficient for this release, not for all time
Metaphor

- You couldn’t start development with just a metaphor?

- There isn’t enough detail there, and besides, what if you’re wrong?

- Unless
  - You quickly have concrete feedback from real code and tests.
  - Your customer is comfortable talking about the system in the terms of the metaphor
  - You refactor to continually refine your understanding of what the metaphor means in practice

Simple Design

- You couldn’t possible have just enough design for today’s code? You would design yourself into a corner and you’d be stuck, unable to continue evolving the system

- Unless
  - You were used to refactoring
    - So making changes was not a worry
  - You had a clear metaphor so future changes would follow a clear idea
  - You were programming with a partner, so you were confident you were making a simple and good design
**Testing**

- You couldn’t possibly write all those tests? It would take too much time. "Programmers won’t write tests!"

- Unless
  - The design is as simple as it can be
  - You are programming with a partner, so if you couldn’t think of another test your partner can
  - You feel good when you see the tests all running
  - Your customer feels good about the system when they see all of their tests running

**Refactoring**

- You couldn’t possibly refactor the design of the system all the time? It would take too long, it would be too hard to control, and it would most likely break the system?

- Unless
  - You are used to collective ownership of the code
    - So you don’t mind making changes wherever they are needed
  - You have coding standards
  - You program in pairs
    - More courage to tackle a though refactoring
  - You have simple design
  - You have the tests
  - You have continuous integration
    - So it is a very short time till you know if your refactoring conflicts with someone else’s work
  - You are rested
Pair programming (see also http://www.pairprogramming.com)

- You couldn’t possibly write all production code in pairs? It will be to slow? What if two people don’t get along?

- Unless
  - Coding standards are used
  - Everyone is fresh and rested
  - The pairs write the tests together, giving them a chance to align their understanding before tackling the meat of the implementation
  - The pairs have a metaphor
  - The pairs are working with a simple design, so they can both understand what is going on

- In addition
  - Sometimes you switch partner and pairs
    - Gives mentor effect
    - More developers learn more parts of the system
    - Coding standards and understanding is upheld

Collective Ownership

- You couldn’t possibly have everybody changing potentially anywhere? Folk would be breaking stuff left and right, and the cost of integration would go up dramatically

- Unless
  - You integrate often, so the chances of conflicts go down
  - You write and run tests, so the chance of breaking things accidentally goes down
  - You pair program, so you are less likely to break code, and programmers learn faster what they can profitably change
  - You adhere to coding standards
**Continuous Integration**

- You couldn’t possibly integrate after only a few hours of work? Doesn’t integration take too long time?

- Unless
  - You can run the tests quickly
    - So you know you haven’t broken anything
  - You program in pairs
    - Half as much to integrate…
  - You refactor, so there are more smaller pieces
    - Reduces the chance of conflicts

**40-Hour Week**

- You couldn’t possibly create enough business value in a 40 hour week?

- Unless
  - The Planning Game is feeding you more valuable work to do
  - The combination of the Planning Game and Testing reduces the frequency of nasty surprises
  - The practices as whole help you program at top speed
On-Site Customer

- You couldn’t possibly have a real customer on the team, sitting there full-time?

- Unless
  - They can produce value for the project by writing functional tests.
  - They can produce value for the project by making small-scale priority and scope decisions for the programmers.

Coding Standards

- You couldn’t possibly ask the team to code to a common standard? Programmers are deeply individualistic, and would quit rather than put their curly braces somewhere else.

- Unless
  - The whole XP helps them to be members of a winning team.
Some programming techniques used in XP

- We look at some of the central programming techniques in XP which fruitfully also could be used in almost all program construction.

- We briefly look at
  - "Test-infected" program construction
  - Refactoring
  - Continuous integration

Testing

- How is testing fulfilled?

- When do we write the test and when are they executed?
  - Unit tests (are in a sense driving the whole of the XP-process)
  - Integration tests
  - Functional tests, acceptance tests and tests of performance

- Are there any software at our aid?

- What is tested?

- References We look for them (in-) directly on the net
Examples

- We look at some papers on the net. But this time only briefly and we will come back and look at them in more detail later on in the course.
  - Beck’s original paper
    - [http://www.xprogramming.com/testfram.htm](http://www.xprogramming.com/testfram.htm)
  - Java
    - [http://www.junit.org/](http://www.junit.org/)
  - Other programming languages
    - [http://www.xprogramming.com/software.htm](http://www.xprogramming.com/software.htm)

What is refactoring?

- Definition 1
  - Refactoring (noun): a change made to the internal structure to make it easier to understand and cheaper to modify without changing its observable behavior

- Definition 2
  - Refactoring (verb): to restructure software by applying a series of refactorings without changing its observable behavior
A catalogue of refactoring “rules”

- We look at some of Martin Fowler’s short tips for refactoring at the following web-page:

- As you see they are based on design patterns...

The two hats

- When you code it is useful to use two hats!
  - One for adding functionality
  - And one while you improve, tidy up, and restructure the code, while you refactor

- But perhaps there are at least three hats!
  - Since when you add functionality you use
    - The testing hat
    - The coding hat

    - Where the adding hat comprises both testing and parts of the coding and the refactoring hat at least is used while you are coding
**Why should you refactor?**

- Refactoring improves the design of the software
  - Otherwise the code easily "rotten" after a while
    - One makes (rapid) short-term changes to the code without fully thinking on the design
  - With continuous improvement of the code usually a lesser amount of code could do the same as if one had used "uglier" code

- Refactoring makes it easier to understand the software
  - One reason to refactor is to get nicer code, and that is…

... 

- Refactoring helps you to find bugs
  - While refactoring you must understand the code better in order re-design it. This process often leads to that you find bugs

- Refactoring helps you to program faster
  - Perhaps a little bit contradictory to our intuition since you must go through more activities. But in the long run you get an improved productivity, since you have to refactor all the time
When should you refactor?

- The rule of Three
  - Not the first time
  - Not the second time either
  - But the third time

- When you add function

- When you need to fix a bug

- Refactor as you do a code review

Why refactoring works?

- Programs that are hard to read are hard to modify

- Programs that have duplicated logic are hard to modify

- Programs that require additional behavior that requires you to change running code are hard to modify

- Programs with complex conditional logic are hard to modify
Refactoring: some references

- Martin Fowler (also includes general XP-material)

- Wikipages

Continuous integration

- What is continuous integration?
  - Continuous integration means that you continuously integrate all the pairs’ stories into one common platform
    - Ideally several times a day

- Is it hard and expensive to integrate?
  - What is the cost of continuous integration?
    - A lot of developers have experienced how costly integration could be
    - Is it not as costly to continuously integrate as if you do it more seldom?

- References, see for instance
  - [http://www.xprogramming.com/Practices/PracContIntFreqTest.html](http://www.xprogramming.com/Practices/PracContIntFreqTest.html)
  - [http://www.martinfowler.com/articles/continuousIntegration.html](http://www.martinfowler.com/articles/continuousIntegration.html)
**Conclusions**

- XP is a light-weight methodology which not require that a lot of documentation is produced
  - But traditional documentation is not forbidden (for instance the customer could write a story that requires documentation...)

- However there are a lot of rules to follow
  - But you are allowed to be pragmatic and think by yourself! That is, among other things, use your common sense
  - All the rules together lead to better projects, better code, more rapid development, more satisfied developers and customers

- XP has developers, customers and the code in focus!

**References**

- There are a lot of information on XP on the web
  - XP’s “homepages”
    - [http://www.extremeprogramming.org/](http://www.extremeprogramming.org/)

  - Wiki

  - Conferences

- There are some references directly on the slides as well