Extreme Programming

Origin

Kent Beck  - Martin Fowler

DeMarco comments

Process versus flexibility – armor versus mobility
Most important new development in software engineering?

Evolutionary development

Ready fire aim

Coarse aim – redesign with feedback

Driving car analogy

Use feedback to make small corrections

Develop ability to make small changes corrections

Must be able to change code

Redesign – refactor

Must be able to make small corrections in direction of project

Variables

Cost
Time
Quality
Scope –

Focus on scope

Stories – use cases

Lightweight in order to make easy corrections

Simple design
Simple tools

Small team 10- 20

Practices

Planning game

Simple methods for planning – interactive customer involved

Small releases

Metaphor

Simple design

Testing

Programmers continually write unit tests
Customers write/plan test demonstrating that features are finished

Refactoring
Programmers restructure the system without changing its behavior to remove duplication improve communication, simplify, or add flexibility.

**Pair programming**
- All production code is written by two programmers at one machine.

**Collective ownership**

**Continuous integration**

**40-hour week**

**On-site customer**

**Coding standards**

**Facilities – central area**

**Scoping a project**

**Does the project make sense**

**Big stories**

**Rough estimates of the time to implement each**
- days or weeks – stories are too small
- experience – fake it?
- write a little prototype
- ask someone

**Budget**

**Constraints**

**Planning Game**

**Players**

**Development**
- Collectively all people responsible for implementing system

**Business**

**Who**
- Real on site customer rep
- Marketing department

**Makes decisions on releases**
- **Scope**
- **Priority**
- **Content**

**Pieces – story cards - stories are:**
- Understandable to customers and developers
- **Sentence or two – index card**
  - Example: The system should check the spelling of all words entered in the comments field
- **Agreement to talk about feature**
- **Testable**
Valuable to the customer
Small enough so that programmers can build a few in each iteration (each iteration 2-3 weeks)

Phases – moves
Exploration phase
  Write a story
    Business writes a story describing something the system needs to do. On index cards, name and short paragraph describing the purpose of the story
  Estimate a story
    Development estimates how long the story will take to implement. May ask Business to clarify or split. Estimates based on past stories – use ideal time (no interruptions)
  Split a story
    If needed
Commitment Phase
  Sort by value
    Business sorts stories into 3 piles
      Must have to function
      Significant business value
      Nice
  Sort by risk
    Development sorts stories into 3 piles by risk
      Estimate precisely
      Estimate reasonably well
      Cannot estimate
  Set velocity
    Development tells Business how fast in ideal time per calendar month
  Choose scope
    Business chooses set of cards in the release
      Choose date figure cards
      Choose cards figure date
      A few months worth of stories typically focusing on a public release

Steering Phase
  Iteration
    Every 1 – 3 weeks business picks one iteration worth of the most valuable stories.
  Recovery
    Development can ask for fewer stories – business decides which
  New story
    Business can write a new story
    Development estimates
Business removes stories with the equivalent estimate

Iteration planning game
Players - programmers
Pieces are task cards
Task - something that can be done in a few days

Moves
Exploration Phase
Write a task
take stories for the iteration and turn them into tasks
Split/combine tasks
Commitment Phase
Accept a task
Programmers sign up
Estimate a task
the responsible programmer estimates in ideal engineering days
Set load factors
A programmer can only sign up for the same number of day’s worth of tasks as he completed last iteration (velocity)

Steering Phase
Implement a task
Take a task card
Find a partner
Write test cases
Make them work
Integrate
Release when universal test suite runs

Record progress
A team member checks on time

Recovery (overcommitted programmers)
Reduce scope of some tasks
Ask customer to reduce scope of stories
Shedding nonessential tasks
Getting better help
Asking customer to defer stories to a later iteration

Verify story (when tasks are complete)
Run functional tests to verify that the story works

Development Strategy
Continuous Integration
After a few hours (no more than a day) integrate
Must have fast build time
Reasonably complete test suite that runs in a few minutes
Collisions
Low chance two pairs of programmers change same class or method at same time
Reconcile easy – represents few hours of work

**Collective ownership**
Anyone can change any piece of code in the system at any time
Tests save you
Others simplify your complex code
Feeling of personal power
Not stuck with someone else’s stupidity
Spreads knowledge of system

**Pair programming**
Dialog not just one person sitting and watching
Switch pairs – my task your task
Information spreads through group
Code quality higher
Maintain standards testing etc.
You learn to communicate about code

**Design strategy**
We will continually refine the design of the system, starting from a very simple beginning. We will remove any flexibility that doesn’t prove useful.

**Simplest thing that could possibly work**
Simplest design easiest to communicate
Feedback – know when design right and wrong quickly
Need simple strategy

**Problem with design up front**
Cost of up front design
May not need features for tomorrow
May learn better how to design between now and then

**THE STRATEGY**
Start with a test. Forces some design just to write test.
What are the objects and their visible methods?
Design and implement just enough to get that test running. You will have to design enough of the implementation to get this test and all previous tests running.
Repeat.
If you ever see the chance to make the design simpler, do it.

**Attributes**
Ridiculous implementation just for 1st test
Second test – refactor then add capability then test
Etc.
If big refactoring needed
   A little at a time
   Team meeting CRC cards

Testing Strategy
Tests automatic
Tests isolated
Test things that might break
   If code is so simple that it can’t possibly break and
doesn’t in practice, need not test
   Testing is a bet that it might break

Programmer tests
   Method by method
   When
     If the interface of a method is unclear, write a test
     before you write the method
     If interface clear, but you imagine that the
     implementation is the least bit complicated write a
     test before you write the method
     Unusual circumstance in which the code should
     work - write a test
     If problem later, write a test that isolates the
     problem
     When about to refactor and aren’t sure how its’
supposed to behave, and not already a test write it.

Customers write tests story-by story
   What would have to be checked before I would be
   confident that this story was done
   Have help to write tests – sometimes one person
dedicated to this
   Functional tests not 100%? Near release decide

Other tests
Can we adapt XP?