Introduction to XP

“When the tests all run, you’re done”
Options

✔ XP is designed around the concept of options
  – Option to abandon
  – Option to switch
  – Option to defer
  – Option to grow and learn
The Four Variables

✔ Management or the Customer chooses 3 of the four variables, the development team defines the fourth.

✔ Cost
  – Cost is the amount of capital available, which defines resources. More resources don’t necessarily mean better quality or shorter time (remember Brooks?)

✔ Time
  – The amount of time available for the project through delivery

✔ Quality
  – Quality is the degree to which and aplomb with which functionality meets requirements

✔ Scope
  – Scope is the *amount* of work to be done, the totality of the set of requirements. As requirements come and go, scope vacillates.
The Paradigm Shift

✓ XP is based on the rejection of a fundamental and long-standing principle, that it costs less to make changes earlier in the development cycle rather than later. That the graph of cost to change is exponential across time. This fundamental principle has led to several strategies:
  – Better safe than sorry
  – Functional extravagance
  – Design extravagance
  – Proliferation of activities that may never provide a return on the investment

✓ The fundamental technical premise of XP is that the graph of cost to change is not exponential but digressive, and as time goes by, the cost to change is asymptotic. “You make the big decisions as late in the process as possible.” This has several strategies:
  – You implement only what you have to, and add functionality later only if necessary
  – Design is parsimonious
  – Thoreau’s principle: Simplify, Simplify, Simplify.
  – Automated tests
  – Refactoring
  – Learning to drive analogy
  – informality
The Four Values

✔ Communication
  – Communication is bipartite. Developers need to communicate with customers as well as between themselves

✔ Simplicity
  – “What’s the simplest thing that could possibly work?” Let’s do that.

✔ Feedback
  – Continuous and instant feedback to all artifacts
  – Continuous and instant feedback to the project progression
  – Continuous and instant feedback to code

✔ Courage
  – The courage to change (alter design, throw away code)
  – The courage to decide
  – The courage to do
  – The courage to be
The Basic Principles of XP

- Rapid feedback
  - instant evaluation of all work and deliverables
- Assume simplicity
  - 98% of problems can be solved with “ridiculous simplicity”
  - What happened to complexity?
    - Complexity != complex solutions
- Incremental change
  - Avoid big changes, make smaller changes more often (driving analogy)
- Embracing change
  - Might as well. Heraclitus was right, Parmenides was wrong. You simply will not be stepping into the same river twice.
- Quality work
  - Work ethic
  - Is Beck a little too hopeful on the human condition?
Subordinate Principles

✔ Teach learning
✔ Small initial investment
✔ Play to win
✔ Concrete experiments
✔ Open, honest communication
✔ Work with people’s instincts, not against them
✔ Accepted not foisted responsibility
✔ Local adaptation (of process)
✔ Travel light (the *nomadic* team)
✔ Honest measurement (no lying)
The Four Basic Activities

✓ Coding
✓ Testing
✓ Listening
✓ Designing
Dominance of Coding and Testing

- Code is unambiguous and constant. It offers no opinions.
- Code is another language for communication (as in pair programming)
- Tests allow for a secondary view into the code, from another angle
- Tests verify that “what was meant” was actually implemented
- Tests can validate performance as well as functionality
- You are responsible for writing multiple unit tests, you write a simple test for every possible way to “break” your code.
- Automated tests can prolong the longevity of the code, and provide continuous validation.
- A testing mentality promotes more self-assured programming style, as successful tests yield confidence in the code.
The Practices

✓ Planning – quickly determine the scope of the next iteration. Customers do the planning based on feedback from the developers.
   – “Software development is always an evolving dialog between the possible and the desirable.”
✓ Small Releases – take baby steps in each iteration. Rank iterations according to those which deliver the most valuable business requirements.
✓ Metaphor – define a simple story of how the system will work. It should be enlightening.
✓ Simple design – few classes and methods, no duplicated logic
✓ Testing – Developers write unit tests, Customers write functional tests
✓ Refactoring – revisiting code with rules that simplify the code. “When the system requires that you duplicate code, it’s asking for refactoring.”
✓ Pair Programming
✓ Collective Ownership – anyone can change any code at any time.
The Practices, cont.

- ✔ Continuous Integration – code is integrated every half or full day at most. Integration is putting new code with the current system.
- ✔ Sane work week
- ✔ On-site customer – customer needs to be around
- ✔ Coding standards that all coders follow
Pair Programming

✓ One programmer writes the code, at the low level. He/she “has the ball”, or at least the keyboard.
✓ The other programmer looks at the code being written from a higher strategic level:
  – What additional tests could break this?
  – Can this be done more simply? (designing)
  – Have I seen this before? (Refactoring)
  – Did the guy with the ball just introduce a bug?
  – Is this the best approach to this problem?
  – Did the guy with the ball forget something?
  – Does a question need to be answered by the Customer?
✓ Coding standards help reduce the need for reformatting code and bickering about style.
✓ Pairs write tests together too, following the same principles.
“Problems” With Pair Programming

✔ What happens on a geographically distributed development team?
✔ Management will object to “waste”, you only get half as much done, or we’ll need twice as many programmers.
✔ Pairs will naturally “self-select” in a Darwinian sense, militating against teaching learning.
Project Planning

✔ Three phases:
  – Exploration
  – Commitment
  – Steering
Exploration Phase

✔ Write a story (think “simplified” Use Case)
✔ Estimate a story: how long will it take to code this?
✔ Split a story: if a part of a story is more important than another, split it into two stories
Commitment Phase

✔ Business chooses the scope and delivery date of the next iteration

✔ Four movements:
  - Sort by value (must have, should have, nice to have)
  - Sort by (estimation) risk
  - Set velocity – how quickly do we expect to move on this?
  - Choose Scope – Ok, given the above, what are we to deliver and when is it due?
Steering Phase

✔ Four movements:
  
  - Iteration
    - Iterations run 1 to 3 weeks generally.
    - Each iteration selects one or more stories to implement. Each iteration must yield a system that runs end-to-end, however embryonically.
  
  - Recovery: if development has overstated velocity, re-evaluate the set of stories (deliverables)
  
  - New story: If business realizes it’s got a new story, the new story is estimated, ranked, and added.
  
  - Reestimate: If development feels the plan is inadequate, it can reestimate the remaining stories and reset the estimated velocity.
Iteration Planning

✔ Task planning
✔ Three Phases:
  – Exploration Phase
    • Write a task by breaking down the stories into tasks
    • Split a task if necessary
  – Commitment Phase
    • Accept a task
    • Estimate a task
  – Steering Phase
    • Implement a task
    • Record Progress
    • Recovery – what to do if overworked: manage scope
    • Verify story with functional tests
What about Design Strategy?

✓ Start with a test. A simple test.
✓ Design and implement just enough to get that test running, and make sure you don’t break another test.
✓ Add functionality and repeat
✓ Refactor.
✓ “The definition of the best design is the simplest design that runs all the test cases.”