Agile Processes: Extreme Programming
Old School: The Waterfall Model

Gather Requirements

Specification

Design

Implementation

Integration

Product
Old School: Waterfall drawbacks

- The major risks are:
  - Relies heavily on being able to accurately assess requirements at the start
  - Little feedback from users until very late
    - Unless they understand specification and design documents
  - Problems in the specification may be found very late
    - Coding or integration
  - Whole process can take a long time before the first working version is seen
Extreme Programming

• Waterfall model inspired by civil engineering
• Civil engineering metaphor is not perfect
  - Software is more organic than concrete
  - You “grow the software” to meet changing requirements

• Extreme Programming (XP) addresses this
  - An iterative model
Software Process Control Variables

- Control variables in a software project
  - Time
  - Quality
  - Scope (features)

- If you try to fix all three then the hardest to measure (i.e., quality) will suffer
- Management fixes two and the development team controls the third variable

Scope is the control variable for XP
Extreme Programming (XP)

- XP: like iterative but taken to the extreme
XP Development Cycle

Short cycle (2-4 weeks):

1. Meet with client to elicit requirements
   - User stories + acceptance tests

2. Planning game
   - Break stories into tasks, estimate cost
   - Best if client prioritizes stories first

3. Implementation
   - Write unit tests first
   - Simplest possible design to pass the tests
   - Pair programming
   - Occasionally refactor the code

4. Evaluate progress and reiterate from step 1
XP Customer

• Expert customer is part of the team
  - Preferably on site, available constantly
  - XP principles: communication and feedback, often
  - Make sure we build what the client wants

• Customer involved actively in all stages:
  - Clarifies the requirements
  - Writes and runs acceptance tests
  - Negotiates with the team what to do next
  - Constantly evaluates intermediate versions
The Planning Game: User Stories

• Write on index cards
  - meaningful title
  - short (customer-centered) description

• Focus on “what” not the “why” or “how”

• Uses client language
  - Client must be able to test if a story is completed

• No need to have all stories in first iteration
Accounting Software

• I need an accounting software to create named accounts, list accounts, query the account balance, and delete accounts.

• Analyze the customer’s statement and create some user stories
Title: Create Account
Description: I can create a named account

Title: List Accounts
Description: I can get a list of all accounts.

Title: Query Account Balance
Description: I can query account balance.

Title: Delete Account
Description: I can delete a named account
### User Stories

<table>
<thead>
<tr>
<th>Title: Create Account</th>
<th>Title: List Accounts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> I can create a named account</td>
<td><strong>Description:</strong> I can get a list of all accounts.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title: Query Account Balance</th>
<th>Title: Delete Account</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> I can query account balance.</td>
<td><strong>Description:</strong> I can delete a named account</td>
</tr>
<tr>
<td>Title: Create Account</td>
<td>Title: List Accounts</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td><strong>Description:</strong> I can create a named account</td>
<td><strong>Description:</strong> I can get a list of all accounts, sorted alphabetically.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title: Query Account Balance</th>
<th>Title: Delete Account</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> I can query account balance.</td>
<td><strong>Description:</strong> I can delete a named account</td>
</tr>
</tbody>
</table>
## User Stories

<table>
<thead>
<tr>
<th>Title: Create Account</th>
<th>Title: List Accounts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> I can create a named account</td>
<td><strong>Description:</strong> I can get a list of all accounts. I can get an alphabetical list of</td>
</tr>
</tbody>
</table>

- Can I delete if a balance is not zero?

<table>
<thead>
<tr>
<th>Title: Query Account Balance</th>
<th>Title: Delete Account</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> I can query account balance.</td>
<td><strong>Description:</strong> I can delete a named account</td>
</tr>
</tbody>
</table>
## User Stories

<table>
<thead>
<tr>
<th>Title: Create Account</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> I can create a named account</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title: List Accounts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> I can get a list of all accounts. I can get an alphabetical list of</td>
</tr>
</tbody>
</table>

**Can I delete if a balance is not zero?**

<table>
<thead>
<tr>
<th>Title: Query Account Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> I can query account balance.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title: Delete Account</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> I can delete a named account if the balance is zero.</td>
</tr>
</tbody>
</table>
User Story?

Title: Use AJAX for UI
Description: The user interface will use AJAX technologies to provide a cool and slick online experience.

Not a user story (an implementation detail)
Customer Acceptance Tests

• Client must describe how the user stories will be tested
  - With concrete data examples,
  - Associated with (one or more) user stories

• Concrete expressions of user stories
Example: Accounting Customer Tests

Tests are associated with (one or more) stories

1. If I create an account “savings”, then another called “checking”, and I ask for the list of accounts I must obtain: “checking”, “savings”.
2. If I now try to create “checking” again, I get an error.
3. If now I query the balance of “checking”, I get 0.
4. If I try to delete “stocks”, I get an error.
5. If I delete “checking”, it should not appear in the new listing of accounts.
6. …
Automate Acceptance Tests

• Customer can write and later (re)run tests
  - E.g., customer writes an XML table with data examples, developers write tool to interpret table

• Tests should be automated
  - To ensure they are run after each release
  - We’ll talk more about test automation later
XP Development Cycle

Short cycle (2-4 weeks):
1. Meet with client to elicit requirements
   • User stories + acceptance tests
2. Planning game
   • Break stories into tasks, estimate cost
   • Client prioritizes stories to do first
3. Implementation
   • Write unit tests first
   • Simplest possible design to pass the tests
   • Pair programming
   • Occasionally refactor the code
4. Evaluate progress and reiterate from step 1
Tasks

• Each story is broken into tasks
  - To split the work and to improve cost estimates
• Story: customer-centered description
• Task: developer-centered description

• Example:
  - Story:  “I can create named accounts”
  - Tasks:  “ask the user the name of the account”
            “check to see if the account already exists”
            “create an empty account”

• Break down only as much as needed to estimate cost
Tasks

• Validate the breakdown of stories into tasks with the customer
• If a story has too many tasks: break it down
• Team assigns cost to tasks
  - We care about relative cost of task/stories
  - Use abstract “units” (as opposed to hours, days)
  - Decide what is the smallest task, and assign it 1 unit
  - Experience will tell us how much a unit is
  - Developers can assign/estimate units by bidding: “I can do this task in 2 units”
Play the Planning Game

- Read Story Cards
- Write Task Cards
- Unclaimed Tasks
- Select and Estimate Tasks
- Accepted Tasks: Programmer 1, Programmer 2, Programmer 3, Programmer 4

Copyright 2000 by William C. Wake
Planning Game

• Customer chooses the important stories for the next release

• Development team bids on tasks
  - After first iteration, we know the speed (units/week) for each subteam

• Pick tasks $\Rightarrow$ find completion date
  - Pick completion date, pick stories until you fill the budget

• Customer might have to re-prioritize stories
XP Planning Game

EXTREME PROGRAMMING
I CAN'T GIVE YOU ALL OF THESE FEATURES IN THE FIRST VERSION.

AND EACH FEATURE NEEDS TO HAVE WHAT WE CALL A "USER STORY."

OKAY, HERE'S A STORY: YOU GIVE ME ALL OF MY FEATURES OR I'LL RUIN YOUR LIFE.

Copyright © 2003 United Feature Syndicate, Inc.
XP Development Cycle

Short cycle (2-4 weeks):

1. Meet with client to elicit requirements
   - User stories + acceptance tests
2. Planning game
   - Break stories into tasks, estimate cost
   - Client prioritizes stories to do first
3. Implementation
   - Write unit tests first
   - Simplest possible design to pass the tests
   - Pair programming
   - Occasionally refactor the code
4. Evaluate progress and reiterate from step 1
Test-driven development

• Write unit tests before implementing tasks
• Unit test: concentrate on one module
  - Start by breaking acceptance tests into units
• Example of a test
  addAccount("checking");
  if (balance("checking") != 0) throw ...;
  try {
    addAccount("checking");
    throw new TestFailed();
  } catch (DuplicateAccount e) {
  }

  Think about names and calling conventions
  Test both good and bad behavior
Why Write Tests First?

- Testing-first clarifies the task at hand
  - Forces you to think in concrete terms
  - Helps identify and focus on corner cases

- Testing forces simplicity
  - Your only goal (now) is to pass the test
  - Avoid premature optimization

- Tests act as useful documentation
  - Exposes (completely) the programmer’s intent
Why Write Tests First?

• Ensures that you think about testability early
  - How will you know when you are done?
  - In what order do you need to test components?
  - What infrastructure you need to test?
Test-Driven Development & Bug Fixes

- Fail a unit test?
  - Fix the code (or test) to pass the test
- Fail an acceptance test (user story)?
  - Add a functional test, then fix the code
- Fail on beta-testing?
  - Add user stories, acceptance tests
  - Make one or more tests from failing scenario

- Always write code to fix tests
  - Ensures that you will have a solid test suite
XP Development Cycle

Short cycle (2-4 weeks):
1. Meet with client to elicit requirements
   • User stories + acceptance tests
2. Planning game
   • Break stories into tasks, estimate cost
   • Client prioritizes stories to do first
3. Implementation
   • Write unit tests first
   • Simplest possible design to pass the tests
   • Occasionally refactor the code
   • Pair programming
4. Evaluate progress and reiterate from step 1
Simplicity

• Just-in-time design
  - Design and implement what you know right now; don’t worry too much about future design decisions

• No premature optimization
  - It is too easy to miss the performance bottleneck

• In every big system there is a simple one waiting to get out
Refactoring: Improving the Design of Code

• Make the code easier to read/use/extend
  - Change “how” code does something
  - Tests should work as before

• Why? Incremental feature extension often outgrows the initial design
  - Expected because of lack of extensive early design

• But needed even for waterfall-model code
  - Plan for it and expect it
  - No point in trying to avoid it
Refactoring: Remove Duplicated Code

• Why? Easier to change, understand
• Inside a single method: move code outside conditionals
  if(...) { c1; c2 } else { c1; c3}
  c1; if(...) { c2 } else { c3 }
• In several methods: create new methods
• Almost duplicate code
  - ... balance + 5 ... and ... balance - x ... 
  - int incrBalance(int what) { return balance + what; }
  ... incrBalance(5) ... and ... incrBalance(-x) ...
Refactoring: Change Names

• Why? A name should suggest what the method does and how it should be used

• Examples:
  - moveRightIfCan, moveRight, canMoveRight

• Meth1: rename the method, then fix compiler errors
  - Drawback: many edits until you can re-run tests

• Meth2: copy method with new name, make old one call the new one, slowly change references
  - Advantage: can run tests continuously
Refactoring and Regression Testing

• Comprehensive suite **needed** for fearless refactoring

• Only refactor working code
  - Do not refactor in the middle of implementing a feature

• Plan your refactoring to allow frequent regression tests

• Modern tools provide help with refactoring

• Recommended book: Martin Fowler’s “Refactoring”
XP Development Cycle

Short cycle (2-4 weeks):

1. Meet with client to elicit requirements
   - User stories + acceptance tests

2. Planning game
   - Break stories into tasks, estimate cost
   - Client prioritizes stories to do first

3. Implementation
   - Write unit tests first
   - Simplest possible design to pass the tests
   - Occasionally refactor the code
   - Pair programming

4. Evaluate progress and reiterate from step 1
XP: Pair programming

- Pilot and copilot metaphor
  - Or driver and navigator

- Pilot types; copilot monitors high-level issues
  - Simplicity, integration with other components, assumptions being made implicitly

- Disagreements point early to design problems

- Pairs are shuffled periodically
Benefits of Pair programming

WE'RE GOING TO TRY SOMETHING CALLED EXTREME PROGRAMMING.

FIRST, PICK A PARTNER. THE TWO OF YOU WILL WORK AT ONE COMPUTER FOR FORTY HOURS A WEEK.

THE NEW SYSTEM IS A MINUTE OLD AND I ALREADY HATE EVERYONE.

Copyright © 2003 United Feature Syndicate, Inc.
Benefits of Pair Programming

• Results in better code
  - Instant and complete, and pleasant, code review
  - Copilot can think about big-picture

• Knowledge and skill migration
  - Good habits spread

• Reduces risk
  - Results in collective understanding of design/code
Why Do Some Programmers Resist Pairing?

• “Will slow me down”
  - But train somebody else for future maintenance
• It is stressful to relate to people all the time
• Need time alone to figure things out
• Afraid to show you are not a genius
  - Neither is your partner
  - The more genius you are, the more you try to do, the more you can use another pair of eyes
Why Do Some Managers Resist Pairing?

• Myth: inefficient use of personnel
  - That would be true if the most time consuming part of programming was typing!
  - Assume 15% higher dev. cost, and 15% fewer bugs
    • 2 individuals separately: 50 loc/h each, 1 bug/33 loc
    • 1 team: 80 loc/h, 1 bug/40 loc
    • 1 bug fix costs 10 hours
    • 50kloc program 2 individuals: 1000 devel + 15,000 bug fix
    • 50kloc program 1 team: 1250 devel + 12,500 bug fix

• Resistance from developers
  - Ask them to experiment for a short time
  - Find people who want to pair
Evaluation and Planning

- Run acceptance tests
- Assess what was completed
  - How many stories?
- Discuss problems that came up
  - Both technical and team issues
- Compute the speed of the team
- Re-estimate remaining user stories
- Plan with the client next iteration
Putting It All Together

Extreme Programming Project

User Stories → Requirements → Release Planning → Iteration → Acceptance Tests → Small Releases

Architectural Spike → System Metaphor → Uncertain Estimates

Release Plan → Confident Estimates → Next Iteration

Test Scenarios → New User Story Project Velocity → Bugs → Latest Version

Copyright 2000 J. Donvan Wells
What’s Different About XP

• No specialized analysts, architects, programmers, testers, and integrators
  - Every XP programmer participates in all of these critical activities every day

• No complete up-front analysis and design
  - Start with a quick analysis of the system
  - Team continues to make analysis and design decisions throughout development
What’s Different About XP

• Develop infrastructure and frameworks as you develop your application
  - Not up-front
  - Quickly delivering business value is the driver of XP projects
When to (Not) Use XP

• **Use for:**
  - Dynamic project done in small teams (2-10 people)
  - Projects with requirements prone to change
  - Projects where the customer is available

• **Do not use when:**
  - Cost of late changes is very high
  - Your customer is not available (e.g., space probe)
  (In these cases, hopefully requirements are truly known and fixed)
Conclusion

• Extreme Programming is an incremental software process designed to cope with change

• Core principles: customer-on-team, planning game, test-first, rapid cycle, simplicity
  - Additional principles: team programming, sustainable rhythm

• With XP you never miss a deadline; you just deliver less content