Extreme Programming

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
  - A version of the iterative model discussed before

Goals

- Minimize unnecessary work
- Maximize communication and feedback
- Make sure that developers do most important work
- Make system flexible, ready to meet any change in requirements

History

- Kent Beck
  - Influential book "Extreme Programming Explained" (1999)
- Speed to market, rapidly changing requirements
- Some ideas go back much further
  - "Test first development" used in NASA in the 60s
  - Is this surprising?

XP Practices

- On-site customer
- The Planning Game
- Small releases
- Testing
- Simple design
- Refactoring
- Metaphor
- Pair programming
- Collective ownership
- Continuous integration
- 40-hour week
- Coding standards

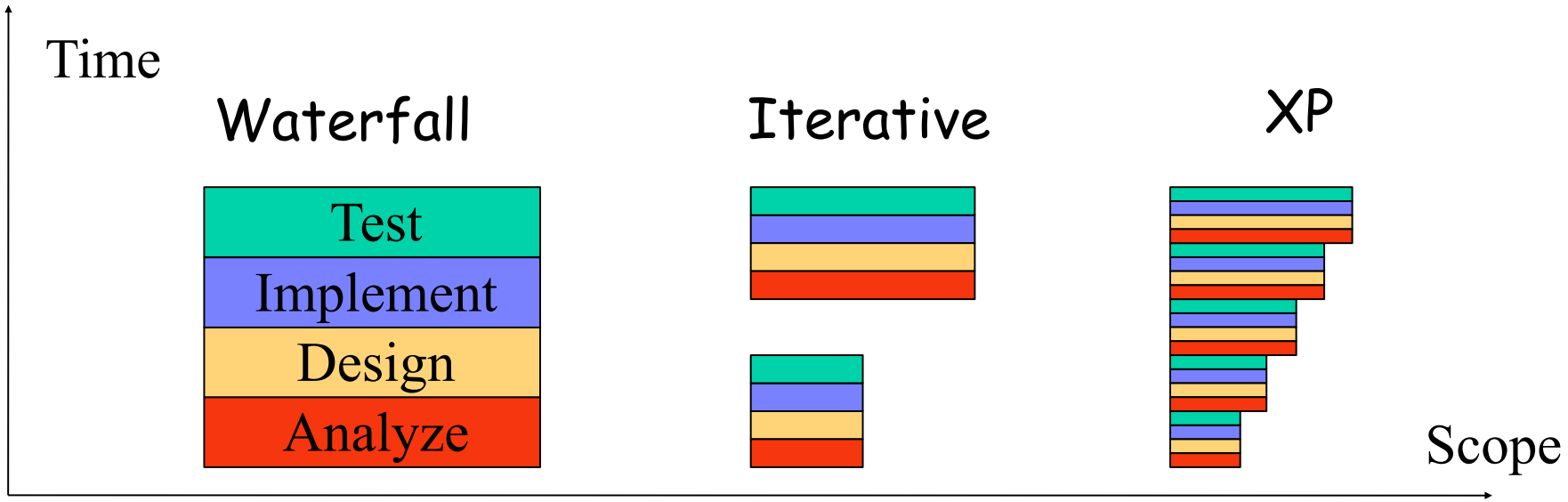
XP Process

Multiple short cycles (2 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 programmer tests first
   - Simplest possible design to pass the tests
   - Code in pairs
   - Occasionally refactor the code
4. Evaluate progress and reiterate from step 1

Extreme Programming (XP)

- XP: like iterative but taken to the extreme



XP Customer

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

- Customer involved actively in all stages:
  - Clarifies the requirements
  - Negotiates with the team what to do next
  - Writes and runs acceptance tests
  - Constantly evaluates intermediate versions
  - Question: How often is this feasible?

The Planning Game: User Stories

- Write on index cards (or on a wiki)
  - 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

Accounting Software

- I need an accounting software that let's me
  - create a named account,
  - list accounts,
  - query the balance of an account,
  - delete an account.

- Analyze the CEO's statement and create some user stories

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 the account balance.

Title: Delete Account
Description: I can delete a named account

User Stories

How is the list ordered?

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

How is the list ordered?

Title: Create Account
Description: I can create a named account

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

Title: Create Account

Description: I can create a named account

Title: List Accounts

Description: I can get a list of all accounts. I can

Possible if balance is not zero?

Title: Query Account Balance

Description: I can query account balance.

Title: Delete Account

Description: I can delete a named account

User Stories

Title: Create Account

Description: I can create a named account

Title: List Accounts

Description: I can get a list of all accounts. I can

Possible if balance is not zero?

Title: Query Account Balance

Description: I can query account balance.

Title: Delete Account

Description: I can delete a named account if the balance is zero.

User Story?

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

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

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

User Stories

Title: Create Account
Description: I can create a named account

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

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

Title: Delete Account
Description: I can delete a named account if the balance is zero.

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 must 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

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

Tasks

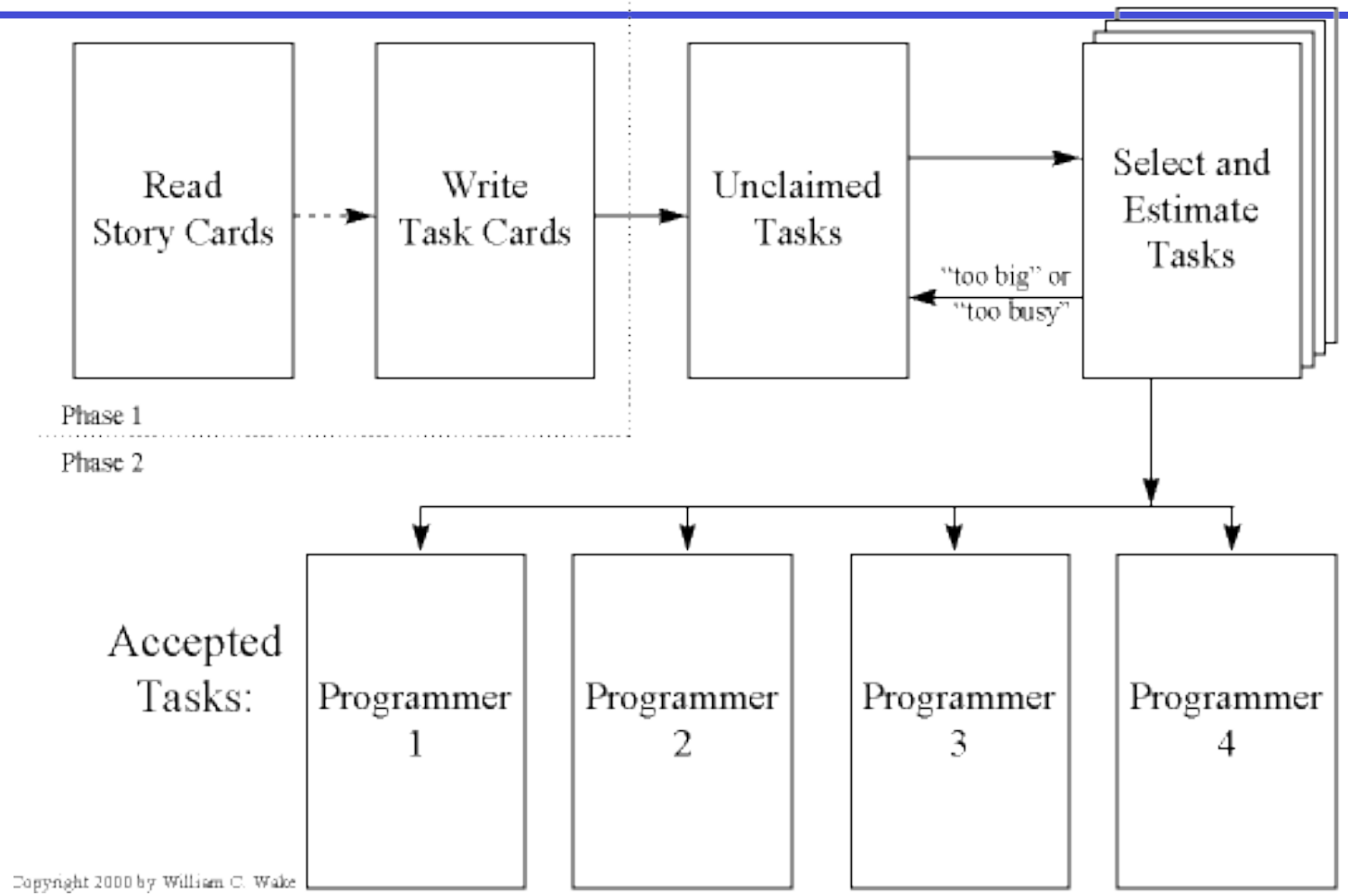
- Each story is broken into tasks
  - To split the work and to improve cost estimates
- Story: customer-centered description
- Task: developer-centered description
- Example:
  - Story: "I can create named accounts"
  - Tasks: "ask the user the name of the account"
    "check to see if the account already exists"
    "create an empty account"
- Break down only as much as needed to estimate cost
- Validate the breakdown of stories into tasks with the customer

Tasks

- 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



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 sub-team
- Pick tasks => find completion date
- Pick completion date, pick stories until you fill the budget
- Customer might have to re-prioritize stories

XP Planning Game

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 …;
} 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
  - Fight premature optimization
- Tests act as useful documentation
  - Exposes (completely) the programmer's intent
- Testing increases confidence in the code
  - Courage to refactor code

Test-Driven Development. Bug Fixes

- Fail a unit test
  - Fix the code to pass the test
- Fail an acceptance test (user story)
  - Means that there aren't enough user tests
  - Add a user test, then fix the code to pass the test
- Fail on beta-testing
  - Make one or more unit tests from failing scenario
- Always write code to fix tests

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
  - You are not going to need it (YAGNI)
- 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/modify
  - Change "how" code does something
- Why?
  - Incremental feature extension might outgrow the initial design
  - Expected because of lack of extensive early design

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; }

Refactoring: Change Names

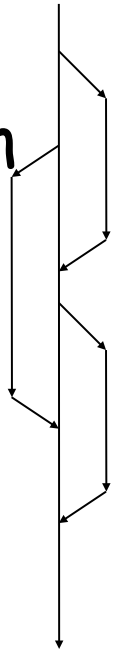
- Why?
  - A name should suggest what the method does and how it should be used
- Examples:
  - moveRightIfCan, moveRight, canMoveRight
- Meth1: rename the method, then fix compiler errors
  - Drawback: many edits until you can re-run tests
- Meth2: copy method with new name, make old one call the new 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"

Continuous Integration

- Integrate your work after each task.
  - Start with official "release"
  - Once task is completed, integrate changes with current official release.
- All unit tests must run after integration
- Good tool support:
  - Hudson, CruiseControl



Hudson

What is Hudson?

Hudson monitors executions of repeated jobs, such as building a software project or jobs run by cron. Among those things, current Hudson focuses on the following two jobs:

1. **Building/testing software projects continuously**, just like CruiseControl or DamageControl. In a nutshell, Hudson provides an easy-to-use so-called continuous integration system, making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build. The automated, continuous build increases the productivity.
2. **Monitoring executions of externally-run jobs**, such as cron jobs and procmail jobs, even those that are run on a remote machine. For example, with cron, all you receive is regular e-mails that capture the output, and it is up to you to look at them diligently and notice when it broke. Hudson keeps those outputs and makes it easy for you to notice when something is wrong.

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

Pair programming

Benefits of Pair Programming

- Results in better code
  - instant and complete and pleasant code review
  - copilot can think about big-picture
- Reduces risk
  - collective understanding of design/code
- Improves focus and productivity
  - instant source of advice
- Knowledge and skill migration
  - good habits spread

Why Some Programmers Resist Pairing?

- "Will slow me down"
  - Even the best hacker can learn something from even the lowliest programmer
- Afraid to show you are not a genius
  - Neither is your partner
  - Best way to learn

Why Some Managers Resist Pairing?

- Myth: Inefficient use of personnel
  - That would be true if the most time consuming part of programming was typing !
  - 15% increase in dev. cost, and same decrease in bugs
    - 2 individuals: 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 + 15000 bug fix
    - 50kloc program 1 team: 1250 devel + 12500 bug fix
- Resistance from developers

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

XP Practices

- On-site customer
- The Planning Game
- Small releases
- Testing
- Simple design
- Refactoring
- Metaphor
- Pair programming
- Collective ownership
- Continuous integration
- 40-hour week
- Coding standards

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:
  - A dynamic project done in small teams (2-10 people)
  - Projects with requirements prone to change
  - Have a customer available
- Do not use when:
  - Requirements are truly known and fixed
  - Cost of late changes is very high
  - Your customer is not available (e.g., space probe)

What can go wrong?

- Requirements defined incrementally
  - Can lead to rework or scope creep
- Design is on the fly
  - Can lead to significant redesign
- Customer representative
  - Single point of failure
  - Frequent meetings can be costly

Recommended Approach in This Class

- "Agile + Classical"
- Classical:
  - Staged waterfall development
  - Generation of project documentation as you go
- Agile
  - XP planning game to move from customer requirements (user stories) to design specification
  - Test-driven development
  - Refactoring
  - Continuous system integration
  - Pair-programming (encouraged)

Conclusion: XP

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

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

Agile Software Development

- "Agile Manifesto" 2001

"Scrum" project management

+ Extreme programming engineering practice

Build software incrementally, using short 1-4 week iterations

Keep development aligned with changing needs

Structure of Agile Team

- Cross functional team
  - Developers, testers, product owner, scrum master
- Product Owner: Drive product from business perspective
  - Define and prioritize requirements
  - Determine release date and content
  - Lead iteration and release planning meetings
  - Accept/reject work of each iteration

Structure of Agile Team

- Cross functional team
  - Developers, testers, product owner, scrum master
- Scrum Master:Team leader who ensures team is fully productive
  - Enable close cooperation across roles
  - Remove blocks
  - Work with management to track progress
  - Lead the "inspect and adapt" processes

Iterations

- Team works in iterations to deliver user stories

- Set of unfinished user stories kept in "backlog"
- Iteration time fixed (say 2 weeks)
  - Stories planned into iterations based on priority/ size/team capacity
  - Each user story is given a rough size estimate using a relative scale

Stories implemented by Tasks

- Story = Collection of tasks
- Wait to break stories into task until story is planned for current iteration
- Tasks estimated in hours
- Stories validated by acceptance tests

When is a Story done?

- "done" means:
  - All tasks completed (dev, test, doc, …)
  - All acceptance tests running
  - Zero open defects
  - Accepted by product owner

SCRUM

- "Process skeleton" which contains a set of practices and predefined roles
  - ScrumMaster (maintains processes)
  - Product Owner (represents the business)
  - Team (Designers/developers/testers)

- At each point:
  - User requirements go into prioritized backlog
  - Implementation done in iterations or sprints

Sprint Planning

- Decide which user stories from the backlog go into the sprint (usually Product Owner)
- Team determines how much of this they can commit to complete
- During a sprint, the sprint backlog is frozen

Meetings: Daily Scrum

- Daily Scrum: Each day during the sprint, a project status meeting occurs

- Specific guidelines:
  - Start meeting on time
  - All are welcome, only committed members speak
  - Meeting lasts 15 min
- Questions:
  - What have you done since yesterday?
  - What are you planning to do today?
  - Do you have any problems preventing you from finishing your goals?

Scrum of Scrums

- Normally after the scrum
- Meet with clusters of teams to discuss work, overlap and integration
- Designated person from each team attends

- 4 additional questions:
  - What has the team done since last meeting?
  - What will the team do before we meet again?
  - Is anything slowing your team down?
  - Are you about to put something in another team's way?

Sprint-related Meetings

- Sprint Planning
- Sprint Review
- Sprint Retrospective

Conclusion: Process

- NO SILVER BULLET!
  - Need to adapt according to specific goals
  - No single process uniformly good or bad

- Necessary (See ESR email to Linus Torvalds)

Acknowledgements

- Many slides courtesy of Rupak Majumdar