CSED332: Software Design Methods

Lecture 3: Agile

Kyungmin Bae

Department of Computer Science and Engineering
POSTECH

Last Lecture: Testing

- Software teting
  - test cases, test suites, test oracles, ...
  - JUnit and test automation
- Test coverage
  - equivalence partitioning, boundary analysis, combinatorial testing
  - structural coverage: method, statement, branch, condition

This Lecture: eXtreme Programming

- What is eXtreme Programming (XP)?
- How extremely does XP differ?
- When (not) to use XP?

Many Way to Develop Software

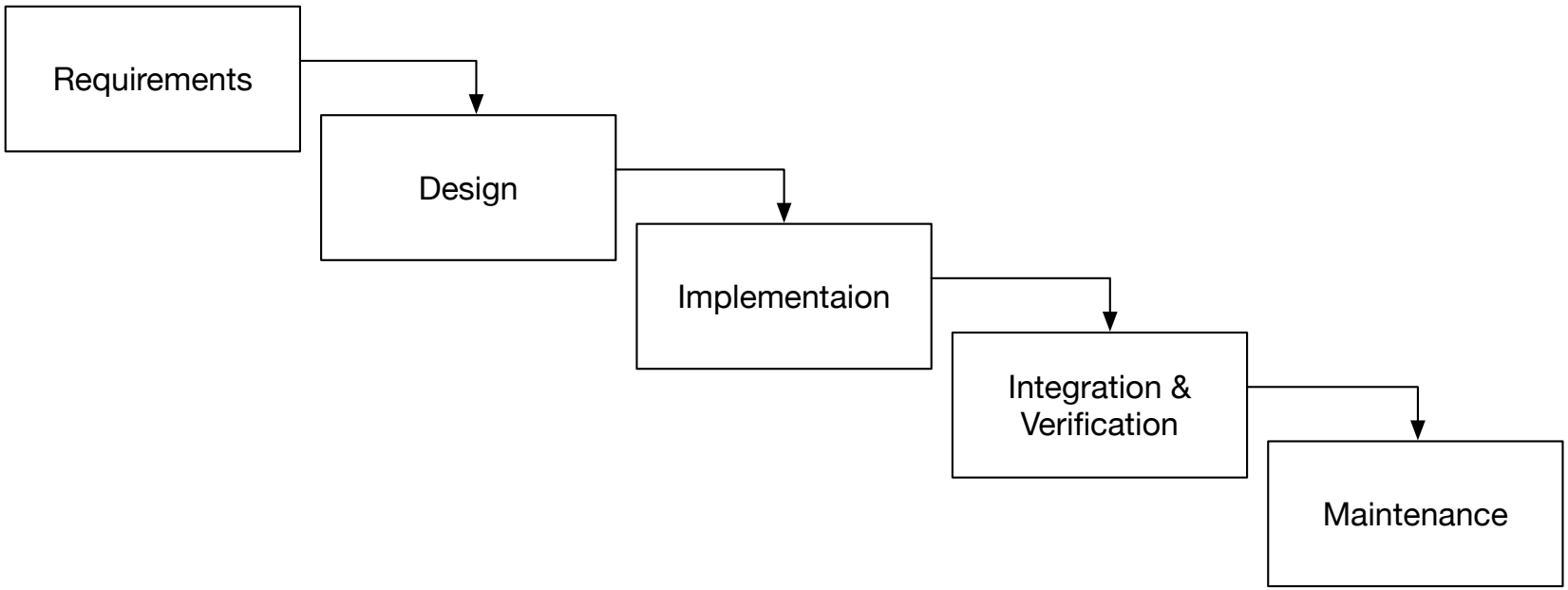
- Plan-driven / agile
- Centralized / distributed
- High math / low math
- Close / little interaction with customers
- Much testing / little testing
- Organize by architecture / features
- ...

Plan-Driven Process: Waterfall Process Activities

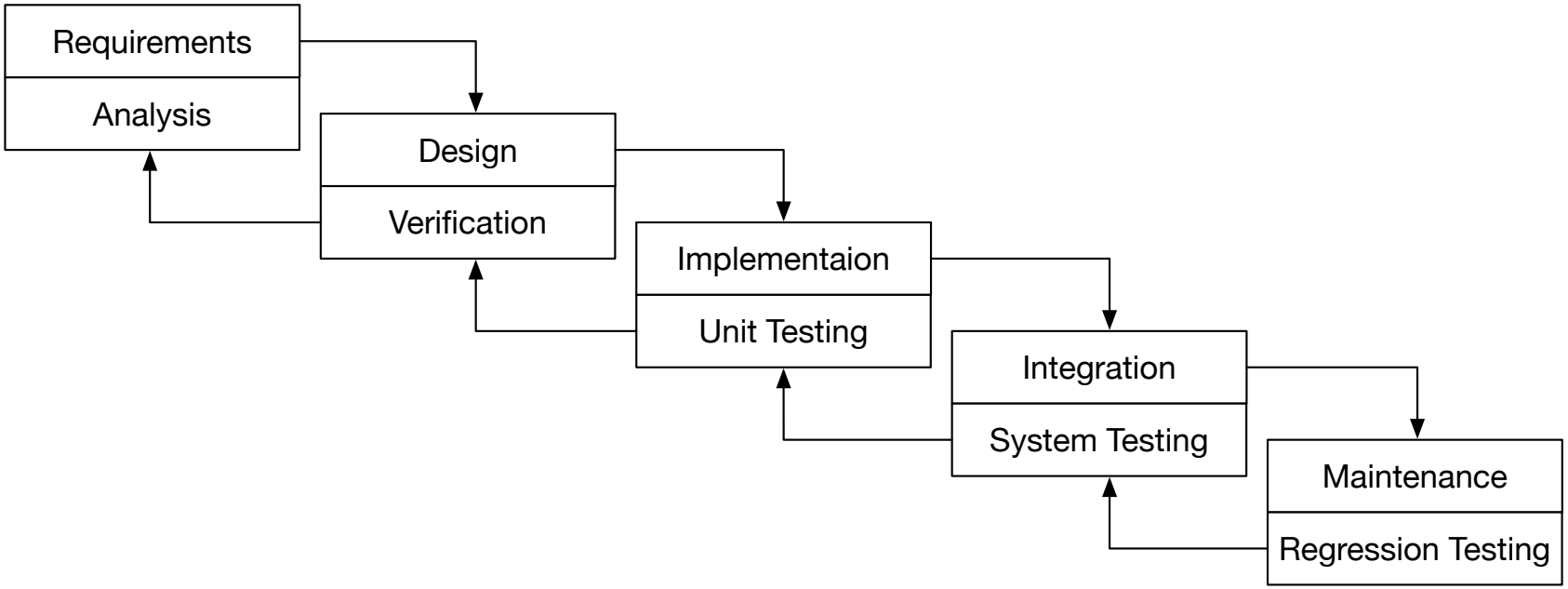
- Requirements: what software should do
- Design: structure code into modules
- Implementation: write code
- Integration: put modules together
- Testing: check if code works
- Maintenance: keep making changes

Often merged together as Verification

Theoretical Waterfall Model



Real-Life Waterfall Model



Agile Process: eXtreme Programming (XP)

- Different from the rigid waterfall process
  - replace it with a collaborative and iterative design process
  - respond quickly to changing requirements
- Main figure: Kent Beck

Main Ideas of XP

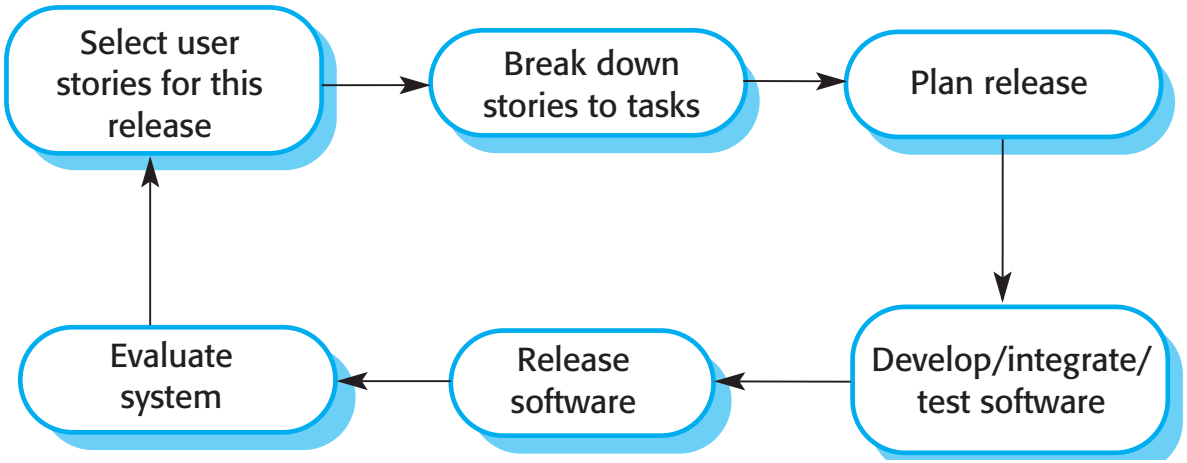
- Do not write excessive documentation
  - working code and tests are the main written product
- Implement features one by one
- Release code frequently
- Work closely with the customer
- Communicate a lot with team members

XP: Some Key Practices

- Planning game for requirements and user stories
- Test-driven development for design and testing
- Refactoring for design and implementation
- Pair programming for development
- Continuous integration for integration

XP is an Iterative Process

- Iteration = two week cycle (1-3 weeks)



1. Plan each iteration in iteration meeting at the start of the iteration
2. Iteration is going to implement set of user stories
3. Divide work into tasks small enough to finish in a day
4. Each day, programmers work in pairs to finish tasks

Pair Programming

https://collaboration.csc.ncsu.edu/laurie/pair.html

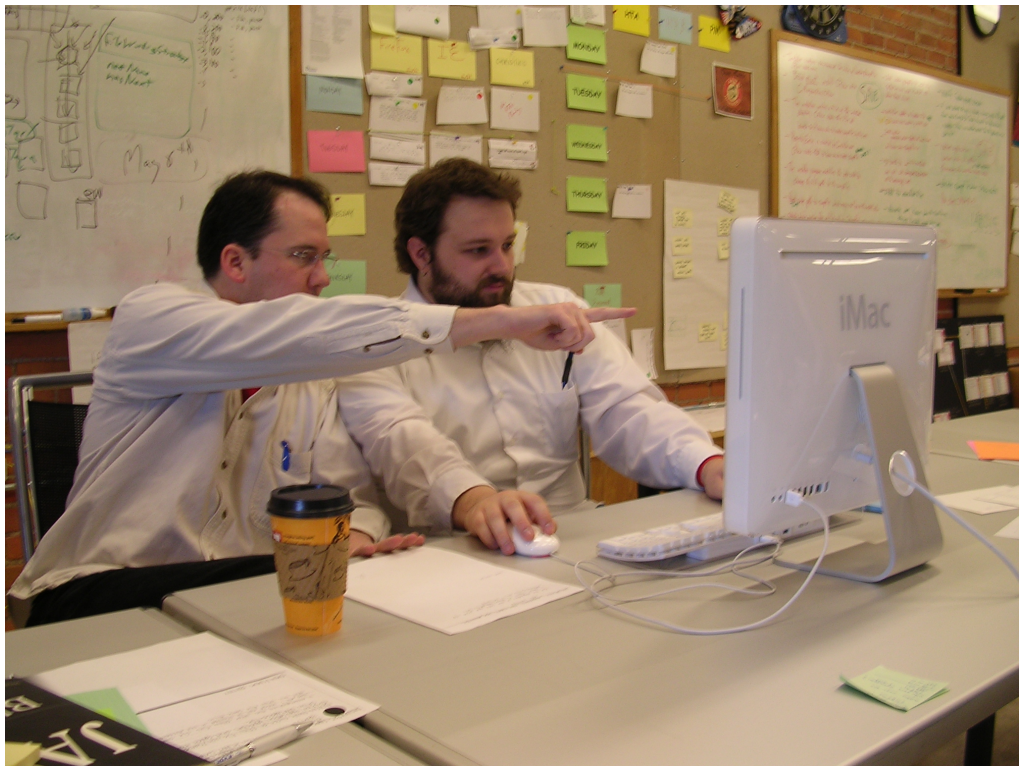
What Is Pair Programming?

> *Pair programming is a simple, straightforward concept. Two programmers work side-by-side at one computer, continuously collaborating on the same design, algorithm, code, and test. It allows two people to produce a higher quality of code than that produced by the summation of their solitary efforts.*
>
> — Laurie Williams

- Driver: types or writes
  - coding, design, debugging, testing, etc.
- Navigator: observes
  - looking for tactical and strategic defects
- Periodically switch roles of driver and navigator
  - possibly every 30 minutes or less

Example: Pair Programming

- Pair programming



- This is NOT Pair Programming
  - `https://www.youtube.com/watch?v=kE0O-Q5D1fw`

Research Findings to Date

- Strong anecdotal evidence from industry

  We can produce near defect-free code in less than half the time.

- Empirical Study
  - Pairs produced higher quality code (15% fewer defects)
  - Pairs completed their tasks in less time (58% of elapsed time)
  - Pairs enjoy their work more (92%)
  - Pairs feel more confident in their work products (96%)
- Measurements suggest
  - productivity of pair is similar to that of two working independently

Benefits of Pair Programming

- Higher product quality
- Improved cycle time
- Increased programmer satisfaction
- Enhanced learning
- Pair rotation
  - ease staff training and transition
  - sharing of knowledge (reduced risk when team members leave)
  - enhanced team building

Issue: Partner Work

Expert paired with an Expert

Expert paired with a Novice

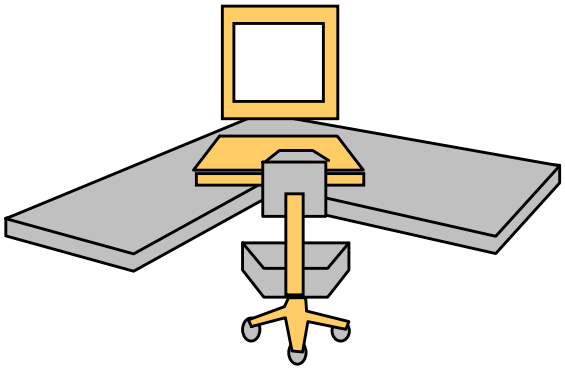
Novices paired together

Professional Driver Problem

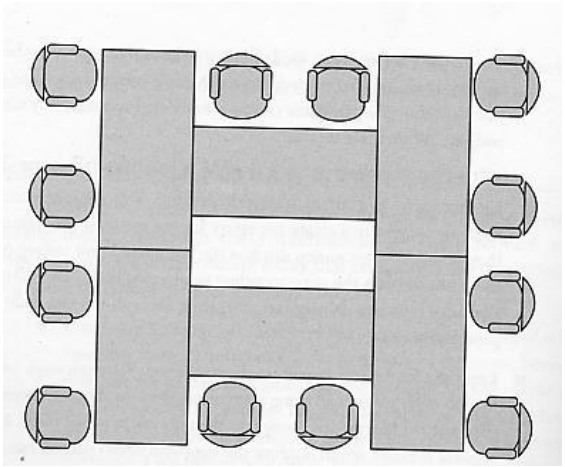
Culture

Issue: Pair Should Rotate

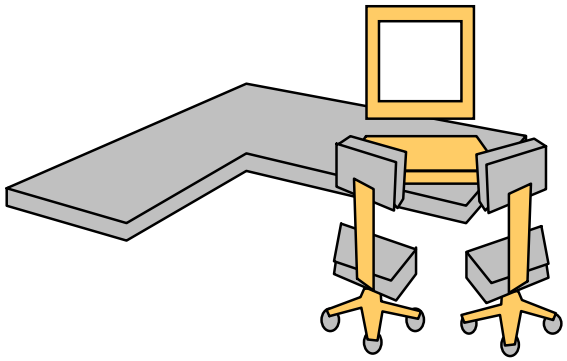
Issue: Workplace Layout



Bad



Best



Better

Issue: Process

- Pair programming is an old concept
  - since the beginning of programming (1950s)
- Used extensively in eXtreme Programming
  - collective code ownership: everyone is responsible for all the code
- But also in other development processes
  - pair programming can be added to any process

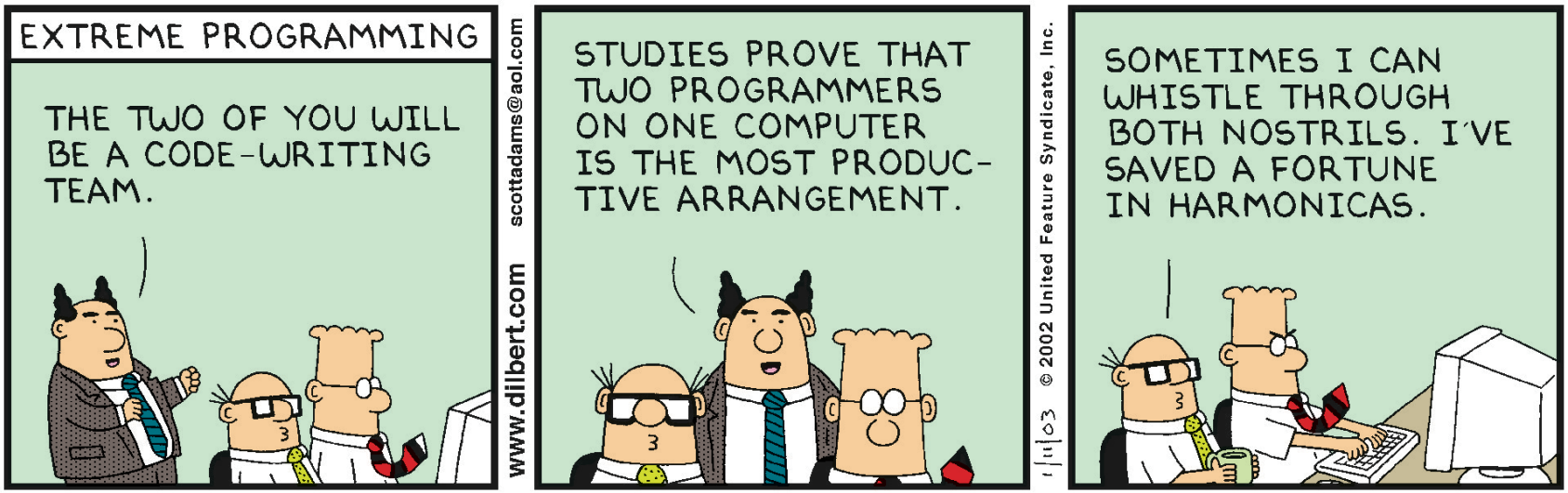
How Does This Work? (1)

- Pair Pressure
  - keep each other on task and focused
  - don't want to let partner down
  - "embarrassed" to not follow the prescribed process
- Pair Negotiation
  - "searching through larger spaces of alternatives"
  - bring different prior experiences, while having shared goals
  - must negotiate how to approach the problem jointly
- Pair Courage
  - if it looks right to me and it looks right to you, ...
  - it's probably right!

How Does This Work? (2)

- Pair Reviews
  - "four eyeballs are better than two"
  - continuous design and code reviews
  - removes programmers' distaste for reviews
- Pair Debugging
  - explaining the problems to another person
  - "never mind; I see what's wrong. sorry to bother you."
- Pair Learning
  - learn from partners techniques, knowledge of language, domain, ...
  - take turns being the teacher and the student minute by minute

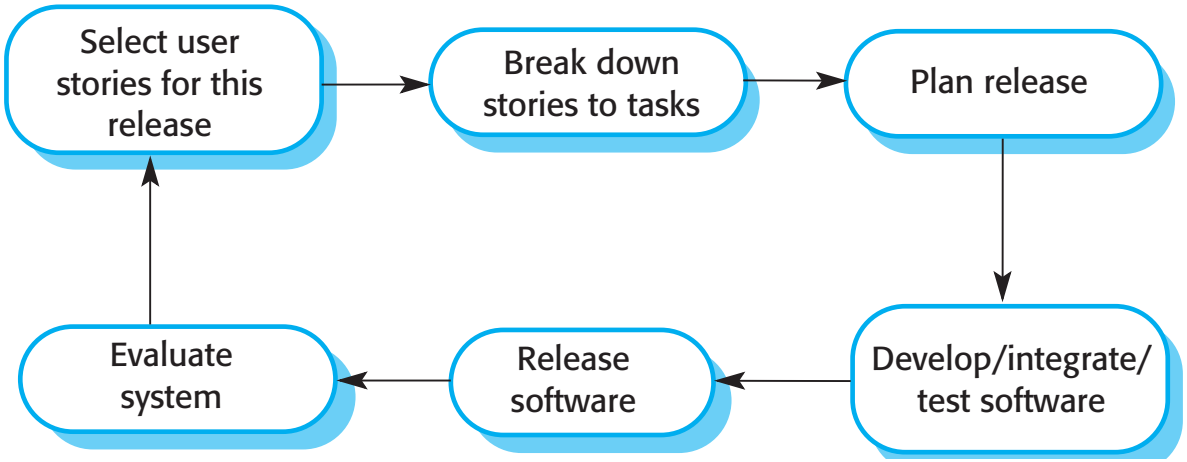
Pairs Means Working Together



User Stories

XP is an Iterative Process

- Iteration = two week cycle (1-3 weeks)



1. Plan each iteration in iteration meeting at the start of the iteration
2. Iteration is going to implement set of user stories
3. Divide work into tasks small enough to finish in a day
4. Each day, programmers work in pairs to finish tasks

What are User Stories?

A user story is the smallest amount of information (a step) necessary to allow the customer to define (and steer) a path through the system.

— David Astels, et al.

- A user story represents :
  - a feature customers want in the software
- Written by our customers (communication with developers)
  - and not by developers (?)
- Typically written on index cards

Writing User Stories

- Materials
  - a stack of blank index cards, pens or pencils, rubber bands
- Start with a goal of the system
  - e.g., applicant submits a loan application
- Think about the steps that the user takes
  - as the user does the activity
- Write no more than one step on each card
  - do not make it too complex

Format of a User Story

- Title: 2–3 words
- Acceptance test
- Priority: 1–2–3 (1 most important)
- Story points (e.g., the number of days of ideal development time)
- Description: 1–2 sentences (a single step towards achieving goal)

Title: Enter Player Info		
Acceptance Test: enterPlayerInfo1	Priority: 1	Story Points: 1
Right after the game starts, the Player Information dialog will prompt the players to enter the number of players (between 2 and 8). Each player will then be prompted for their name, which may not be an empty string. If Cancel is pressed the game exits gracefully.		

Example: Acceptance Test for a Story

- Create Receipt

  Keep a running receipt with a short description of each scanned item and its price.

- Test

  Setup: The cashier has a new customer

  Operation: The cashier scans three cans of beans ($0.99 each), two pounds of spinach ($0.59/lb), and a toothbrush ($2.00)

  Verify: The receipt has all of the scanned items and their correctly listed prices.

Priorities

- High
  - "Give us these stories to provide a minimal working system."
- Medium
  - "We need these stories to complete this system."
- Low
  - "Bells and whistles? Which stories can come later?"

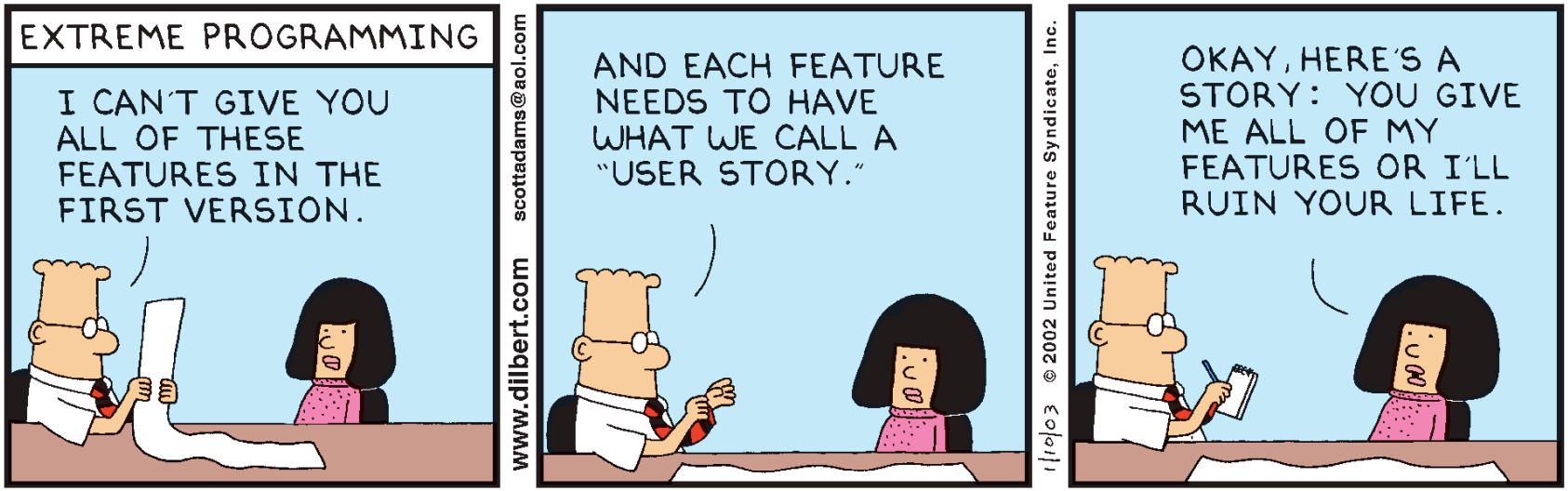
Story Points

- Unit of measure
  - the overall size of a user story, feature, or other piece of work.
  - NOT amount of time or the number of people
- Relative values
  - raw value of a story point is unimportant
- Estimating story points
  - choose a medium-size story and assign it, e.g., 5
  - estimate other stories relative to that

Criteria for User Stories

- Independent: of each other as much as possible
- Negotiable: understandable to the customer, not a contract
- Valuable: must provide value to the customer
- Estimable: to a good approximation
- Small: so as to fit within an iteration
- Testable: in principle, even if there isn't a test for it yet

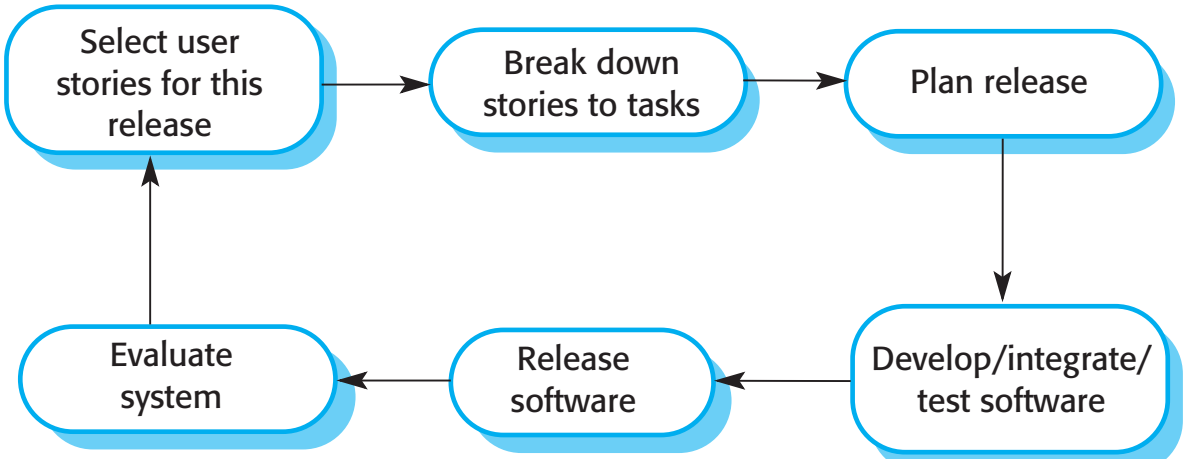
Need Educated Customer



Planning Game[1]

XP is an Iterative Process

- Iteration = two week cycle (1-3 weeks)



1. Plan each iteration in iteration meeting at the start of iteration
2. Iteration is going to implement set of user stories
3. Divide work into tasks small enough to finish in a day
4. Each day, programmers work in pairs to finish tasks

How to Plan

- Make a list
  - everything you have to do
- Decide when you will do each thing
  - How long will it take?
- Make sure you have what you need for each task
  - e.g., dependencies between different tasks

How Long Will It Take?

- Compare with other tasks
  - you have previously done
- Break into smaller steps
  - when too hard to estimate
- Get several people to estimate
  - to consider different aspects

Planning Game

- Customer writes user stories.
  - only provide enough detail to estimate time to implement the story
- Programmers estimate time to do each story.
  - If story is too big, customer splits it
- Customer chooses stories to be implemented that match velocity.
  - project velocity is amount of work done in the previous iteration(s)

Example: Coming up with the Plan

- Desired Features: a user story is given by a customer
- Estimate Size: 30 story points
- Divide by Velocity: 5 story points / two week iteration
- Derive Duration: 6 iterations
- Iteration/Release Plan: 3 months

Estimating Size

- Story point: unit of measure
  - the overall size of a user story, feature, or other piece of work.
  - NOT amount of time or the number of people
- Ideal time:
  - the amount of time, stripped of all peripheral activities
  - E.g., soccer game = 90 minutes
- Elapsed time:
  - the amount of time that passes on the clock
  - E.g., soccer game = 2 hours
- Velocity
  - a measure of how much work is getting done on your project.
  - sum of the estimates of the user stories finished during the iteration

Estimating Story Points

- Choose a medium-size story and assign it a "5"
  - based on how hard to implement it
  - not relate to actual hours; only relative
- Estimate other stories relative to that
  - twice as big
  - half as big
  - almost but not quite as big
  - a little bit bigger
- Only values

0 1 2 3 5 8 | 13 20 40 100

← Near-term iteration "stories" a few iterations away "epic" →

Estimating Ideal Days

- Ideal days vs. elapsed time
  - supporting current release
  - meetings, demonstrations
  - sick time, phone calls, personal issues
  - ...
- When estimating ideal days, assume
  - the story being estimated is the only thing you will work on
  - everything you need will be on hand when you start
  - there will be no interruptions

Ideal Days vs. Story Points

- Both are units of predicted effort
- Story points:
  - pure measure of size (focus on feature, not person)
  - helps drive cross-functional behavior
  - estimation is typically faster in the long run
- Ideal days:
  - my ideal days are not your ideal days (focus on person)
  - easier to explain outside of team
  - estimation is typically faster at first

Deriving an Estimate for a User Story

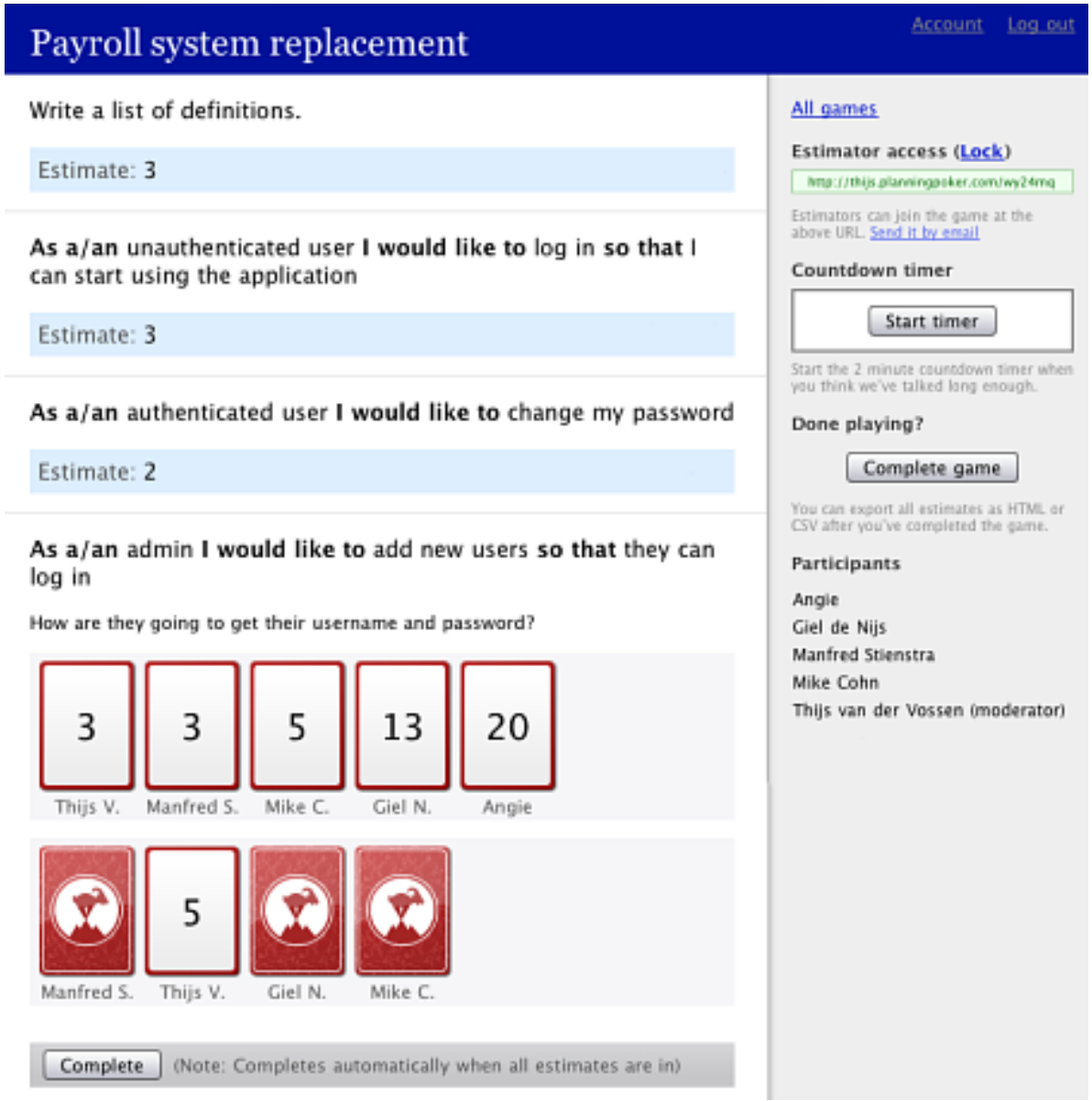
- Expert opinion
  - based on (extensive) experience
  - one expert not be enough (need to consider all aspects)
- Analogy
  - relative to (several) other user stories
  - triangulation: little bigger than “3” and a little smaller than “8”
- Disaggregation
  - break up into smaller, easier-to-estimate pieces or tasks.
  - need to make sure you do not miss any tasks.
- Planning poker
  - combines expert opinion, analogy, disaggregation
  - avoids the cognitive bias of anchoring

Planning Poker Deck

Playing Planning Poker

- Include all players on the development team:
  - but less than 10 people overall
  - programmers, testers, requirements analysts, UI designers, . . .
- Moderator (usually the product owner or analyst)
  - read the description (e.g., user stories) and answer any questions
- Each estimator privately selects a card with their estimate
- All cards simultaneously turned over
- Re-estimate
- Repeat until converge

Planning Poker: http://planningpoker.com



Velocity

- A measure of a team's rate of progress.
  - how much work is getting done on your project
- Summing up the story points assigned to the user stories
  - that the team completed during the iteration
- Assumption
  - team produces in future iterations at the rate of past velocity
- If velocity changes dramatically for more than one iteration
  - need to re-estimate and re-negotiate the release plan

Example: Release Plan

- Desired Features: a user story given
- Estimate Size: 30 story points
- Divide by Velocity: 5 story points / 2 week iteration
- Derive Duration: 6 iterations
- Release Plan: 3 months

Example: Not Working as Fast as Planned?

- Desired Features: a user story given
- Estimate Size: 30 story points
- Divide by Velocity: 3 story points / 2 week iteration
- Derive Duration: 10 iterations
- Release Plan: 5 months

Example: Not Working as Fast as Planned?

- Desired Features: a user story given
- Estimate Size: 18 story points
- Divide by Velocity: 3 story points / 2 week iteration
- Derive Duration: 6 iterations
- Release Plan: 3 months

Velocity Corrects Estimation Errors

- The velocity should change, not each story point estimate
  - to re-estimate and re-negotiate the release plan for future releases
- Since all user stories are estimated relative to each other

Prioritization

- Driven by customer, in conjunction with developer
  - which user stories to be implemented?
- Choose features to fill up velocity of iteration
  - a feature to a broad base of customers or users
  - a feature to a small number of important customers or users
  - the cohesiveness of the story in relation to other stories
- Cohesiveness example
  - "Zoom in" a high priority feature
  - "Zoom out" not a high priority feature but relative to "Zoom in"

Priorities (Revisited)

- High
  - "Give us these stories to provide a minimal working system."
- Medium
  - "We need these stories to complete this system."
- Low
  - "Bells and whistles? Which stories can come later?"

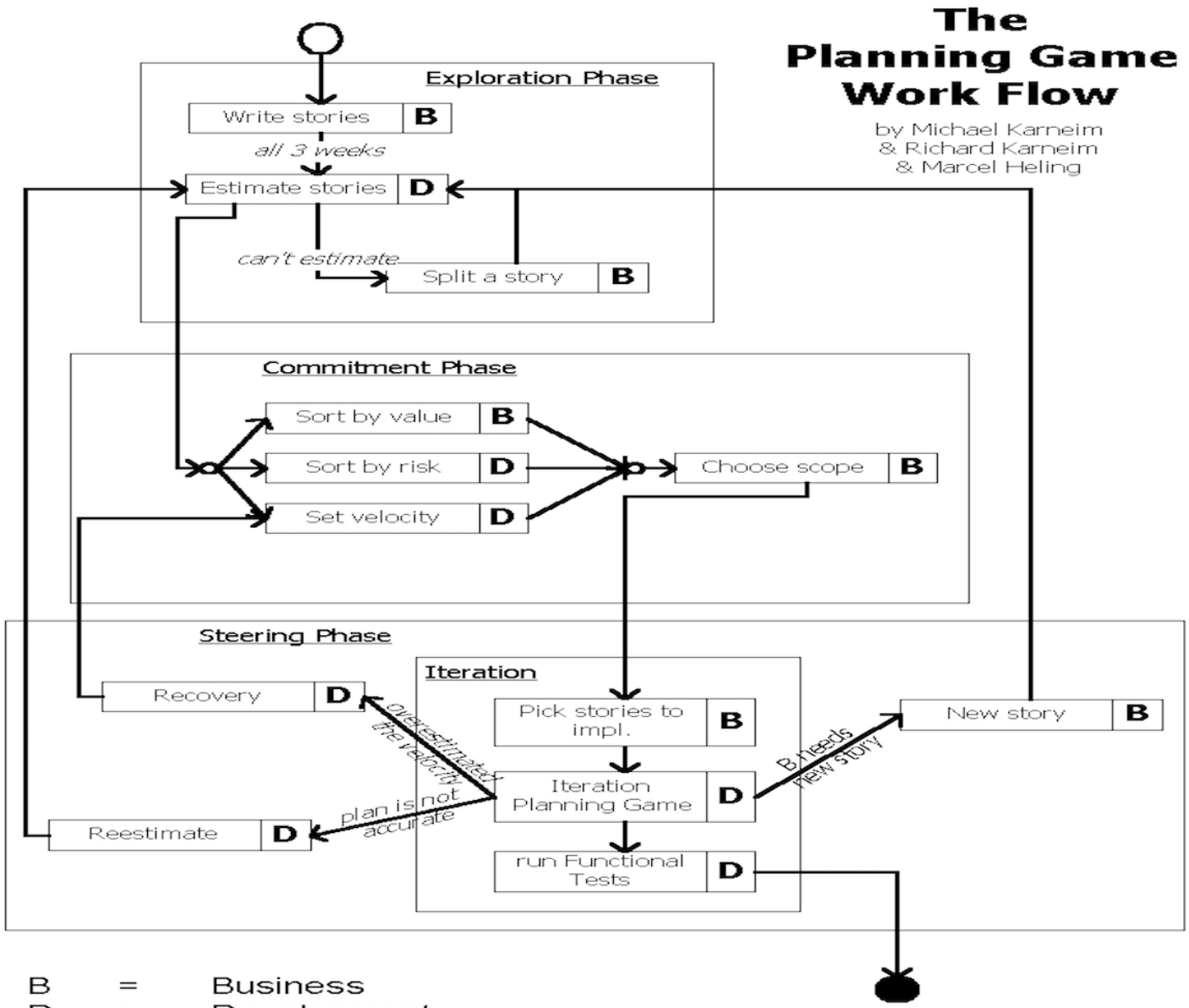
Example: Planning Game (2)

Customer	Desired Features:	a user story given
Developer	Estimate size (too big)	100 story points
Customer	Split the user story	8 user stories
Developer	Estimate Size:	20 story point each
	Divide by Velocity:	5 story points / 2 week iteration
	Derive Duration:	8 * 4 iterations
	Release Plan (small releases):	8 * 2 months
Customer	Choose stories to be implemented first	

Iteration Planning

At the beginning of each iteration

- Customer selects user stories
  - with estimates that total up to the velocity from the last iteration.
- User stories are broken down into the programming tasks
  - developers sign up to do the tasks
- Developers estimate how long their tasks will take to complete
  - ideal days or story points
- Velocity used to determine if the iteration is overbooked or not
  - too much: the customer must choose user stories to be put off
  - too little: another story can be accepted


The Planning Game Work Flow
by Michael Karneim & Richard Karneim & Marcel Heling
Exploration Phase
Write stories
B
all 3 weeks
Estimate stories
D
can't estimate
Split a story
B
Commitment Phase
Sort by value
B
Sort by risk
D
Set velocity
D
Choose scope
B
Steering Phase
Iteration
Recovery
D
Pick stories to impl.
B
New story
B
Overestimated the velocity
Iteration Planning Game
D
B needs new story
plan is not accurate
Reestimate
D
run Functional Tests
D
B = Business
D = Development

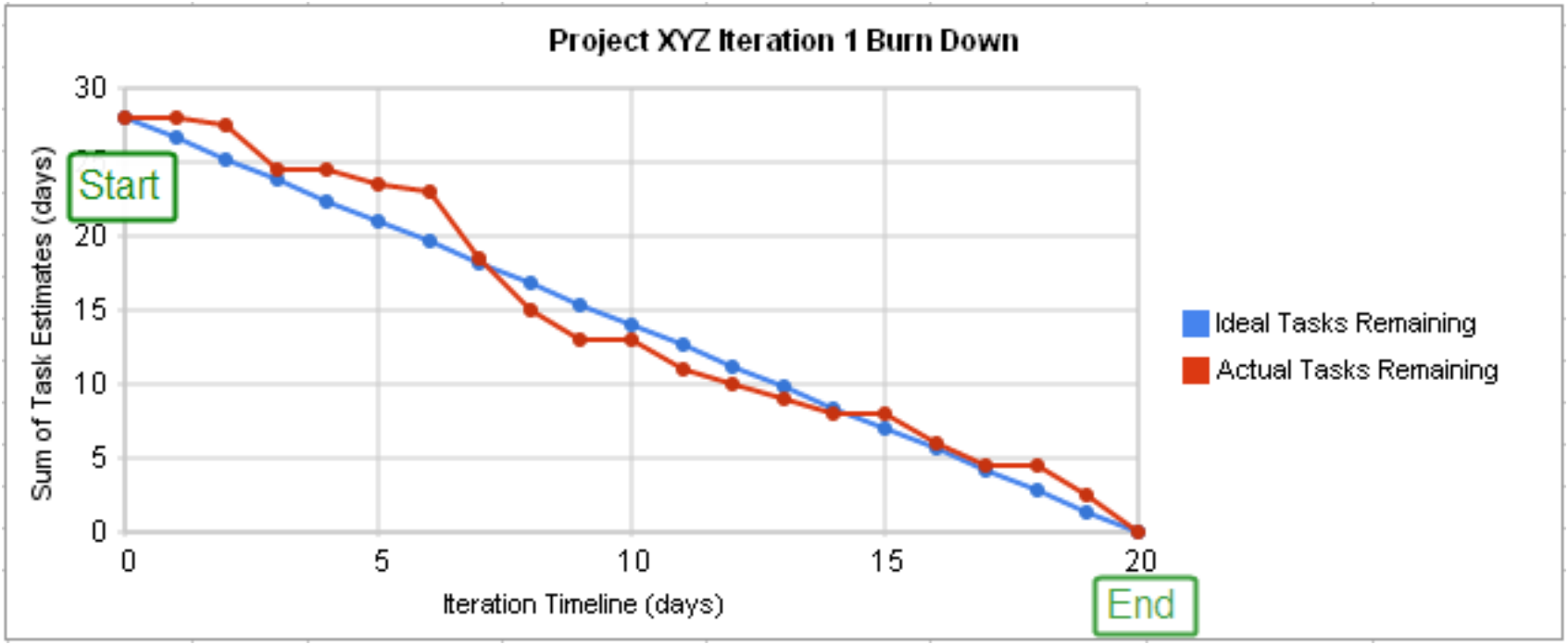
Project Tracking

- Keep track of when each task is finished
  - must be objective (e.g., "wrote a program" not "learned language")
  - different tasks are evaluated differently
- Compare plan to reality
  - estimated vs. actual completion date
- Tasks are either finished or not
  - forbid "90% finished" – decompose tasks
  - in XP, programming task is not finished until all tests run correctly

Your project

Update documents on GitLab to reflect completion dates.

Burn Down Chart



- Shows the amount of work remaining at the start of each iteration

  Vertical axis: number of story points or hours remaining
  Horizontal axis: iterations (or days)

- Visual indicator of how quickly a team is moving toward its goal

Difficulties in Scheduling

- Productivity is not proportional to the number of people
  - tasks cannot be fully partitioned to individuals in practice
  - require communication among subtasks
- Adding manpower to a late software project makes it later
  - called Brooks' law
  - communication overheads and training
- The unexpected always happens
  - always allow contingency in planning

Unit Tests and Refactoring

Simplicity

- Add one feature (user story) at a time
  - Programmers only worry about one iteration at a time
- Do not worry about future stories
  - Customer can change future iterations
- Make program as simple as possible
  - The simplest thing that could possibly work

Recover Simplicity

- Adding a new feature
  - tends to make the code less simple
- Refactoring the code
  - recover simplicity back
- Safe refactoring
  - requires to have a rigorous set of unit tests

Refactoring

- Changing the structure of program without changing its behavior
  - make code easier to understand
  - make code cheaper to modify
- Some refactoring
  - rename, rename, rename
  - extract method and local variable
  - encapsulate field
  - ...
- Automated refactoring support (e.g., Eclipse)

Working Software

- All software has automated (unit) tests
- All tests pass, all the time
- Never check in broken code to the repository

How to Work on a Task

- Get latest version of the code. All tests pass.
- Write test first. It fails.
- Write code to make test pass. Now all tests pass.
- Refactoring (clean up)
- Check in your code (plus smoke testing)

One Key Practice

- Write tests first, then write code
  - tests will initially fails
- Various names
  - Test-first programming
  - Test-driven development
- Is it testing or designing?
  - does not find bugs, but prevents them
  - does not measure quality, but improves it

Why Test?

- Improve quality: find faults
- Measure quality
  - Prove there are no bugs? (Is it possible?)
  - Determine if software is ready to be released
  - Determine what to work on
  - See if you made a mistake
- Learn the software

XP Specification and Testing

- Talk about specification when writing unit tests
  - write tests and then code
  - document code
- Customer (team) writes functional tests (acceptance test)
  - tests are derived from user stories
  - tests need to be documented to ensure traceability to user stories

XP and Test Coverage

- Measure coverage periodically
  - at end of iteration?
  - use automated tools
- Use as feedback
  - To write more tests
  - To improve how you write tests

Summary

- Core XP practices
  - pair programming, user stories, planning game, testing, refactoring
- Homework 3 (Due 10/2):
  - More Java, JUnit, and Maven
- Reading
  - `https://en.wikipedia.org/wiki/Agile_software_development`
  - `http://www.extremeprogramming.org`

Questions?