Software Design

Core XP Practices

Slides are adapted from various software engineering classes offered by Prof. Tao Xie at UIUC

This lecture's goals

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

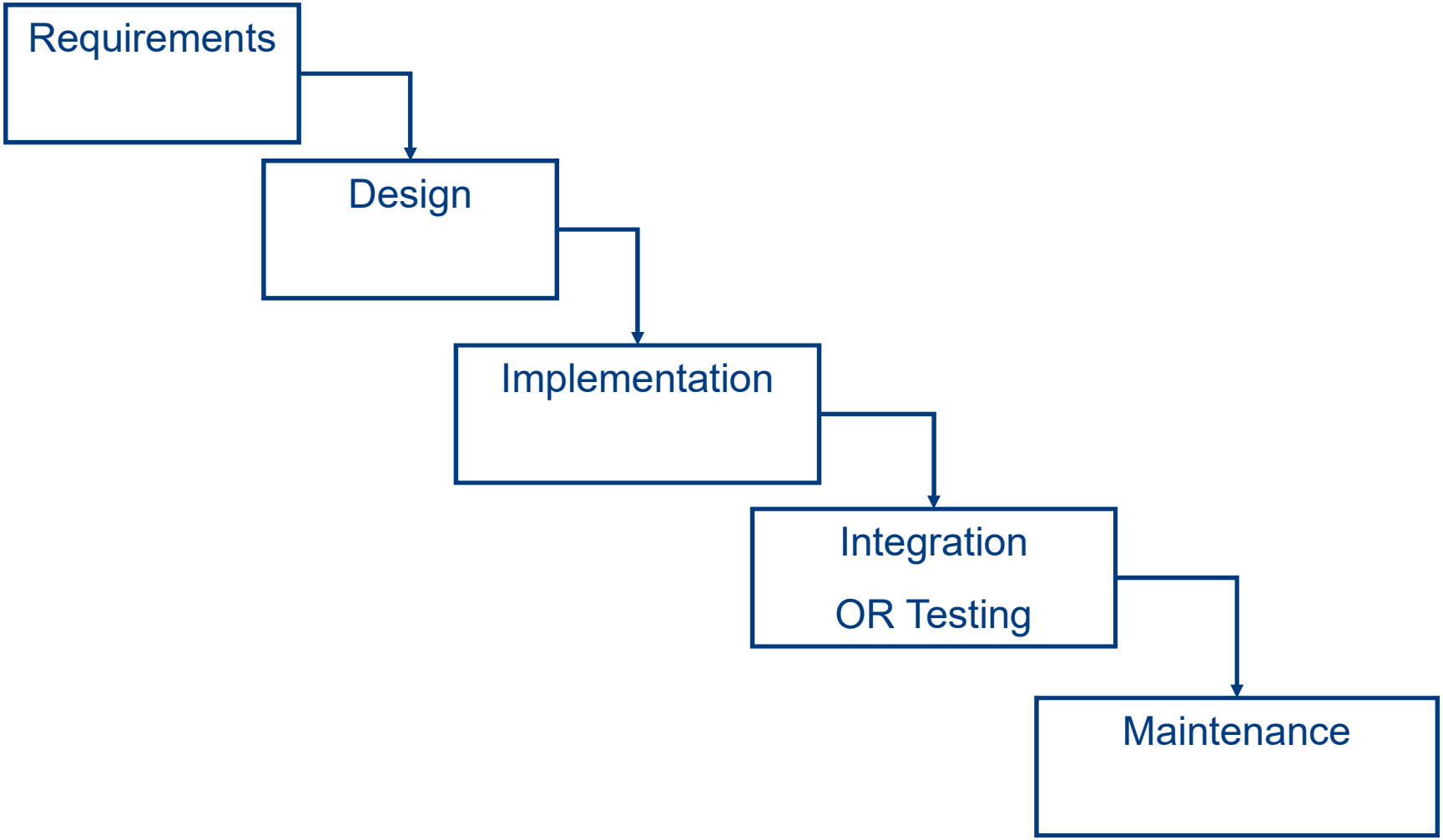
Software development processes

- 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

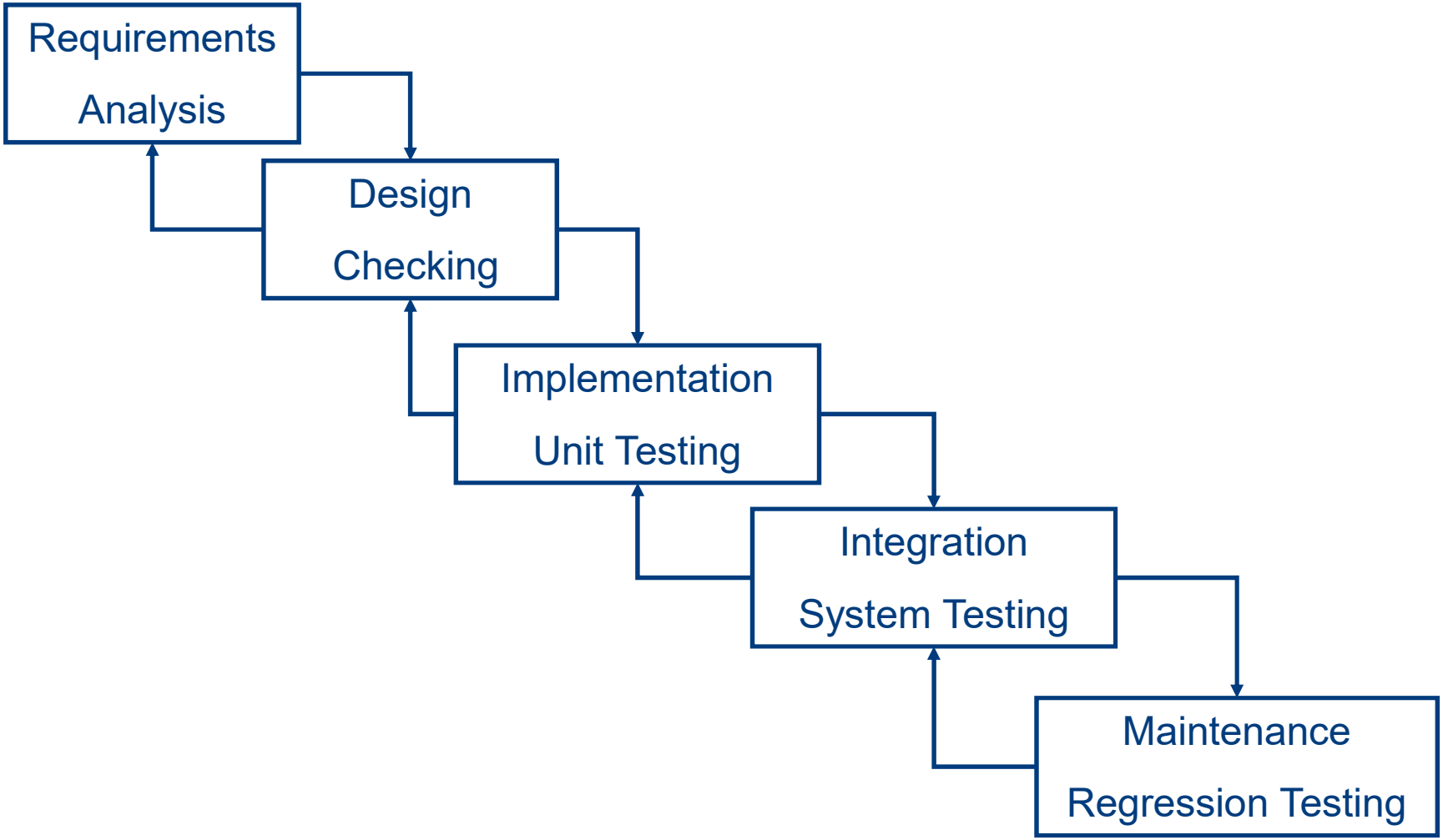
Waterfall process activities

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

Theoretical waterfall model



Modified waterfall model

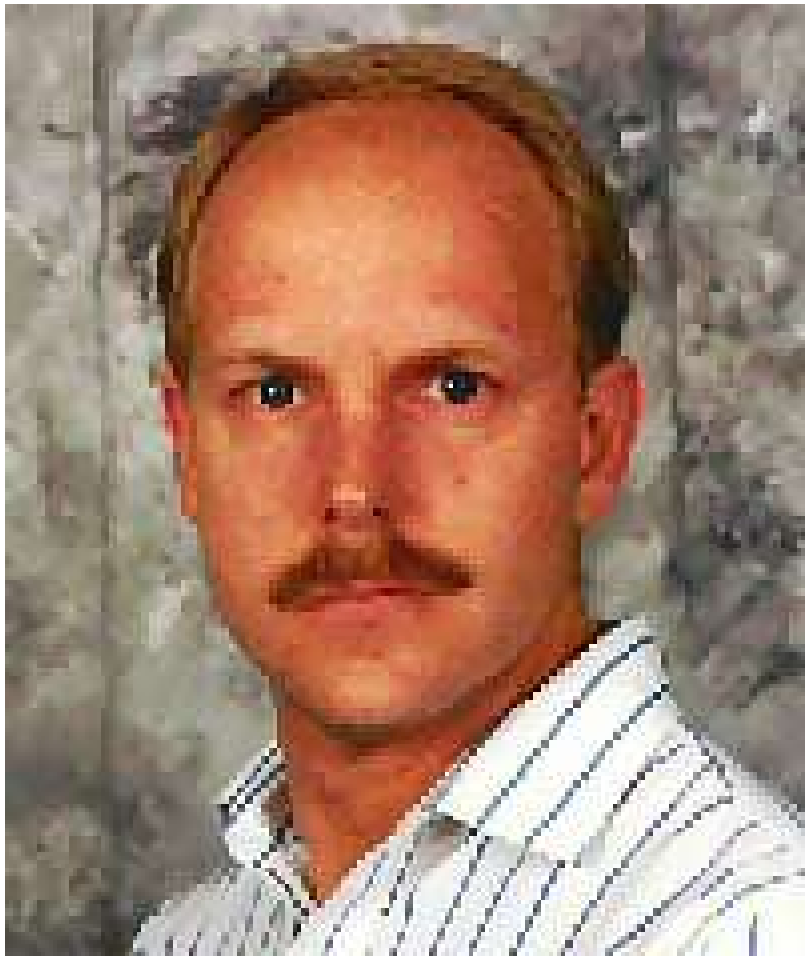


eXtreme Programming (XP)

- Radically different from waterfall
- Big ideas
  - Don’t write much documentation
    - Working code is the main written product
  - Implement features one by one
  - Release code frequently
  - Work closely with the customer
  - Communicate a lot with team members

Main figure

Kent Beck



XP: Some key practices

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

XP is an iterative process

- Iteration = two week cycle (1-3 weeks)
- Plan each iteration in an iteration meeting that is held at the start
- Iteration is going to implement set of user stories
- Divide work into tasks small enough to finish in a day
- Each day, programmers work in pairs to finish tasks

Group Discussion: Pros/Cons in Team Work vs. Solo Work

- Requirement: get into a group of three neighbor students
- share and discuss respective answers based on your current/past team/solo work experiences either at school or outside of school

5 minutes for discussion

- Instructor will call for volunteer groups and sometimes randomly pick groups

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."

Driver: types or writes

Navigator: observer (looking for tactical & strategic defects)

- Periodically switch roles of driver and navigator
  - possibly every 30 minutes or less
- Pair coding, design, debugging, testing, etc.

Pairs (should) rotate

Pair Programming

This is NOT Pair Programming

The Benefits of Pair Programming

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 about half the time
    - 58% of elapsed time
  - Pair programmers are happier programmers
    - Pairs enjoy their work more (92%)
    - Pairs feel more confident in their work products (96%)

Expected Benefits of Pair-Programming

- Higher product quality
- Improved cycle time
- Increased programmer satisfaction
- Enhanced learning
- Pair rotation
  - Ease staff training and transition
  - Knowledge management/Reduced product risk
  - Enhanced team building

Issues: Partner Work

Expert paired with an Expert

Expert paired with a Novice

Novices paired together

Professional Driver Problem

Culture

Issues: Process

- Used in eXtreme Programming
- Used in the Collaborative Software Process
- Pair programming can be added to any process

How does this work?

- Pair Pressure
  - Keep each other on task and focused
  - Don't want to let partner down
  - "Embarrassed" to not follow the prescribed process

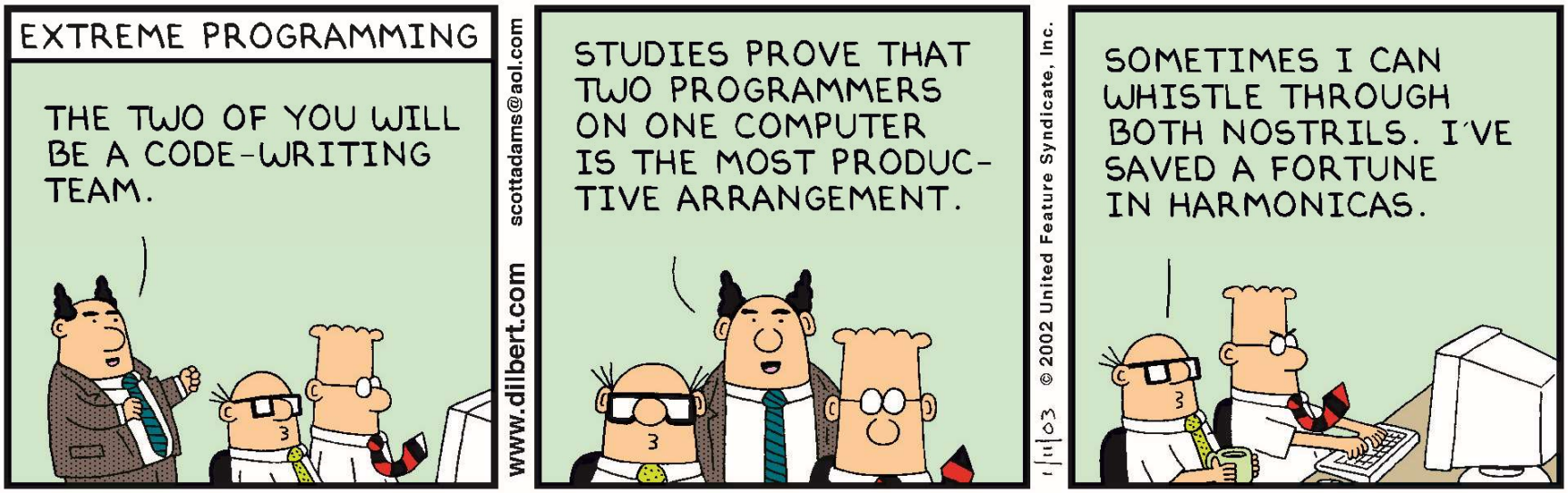
- Pair Negotiation
  - Distributed Cognition: "*Searching Through Larger Spaces of Alternatives*"
    - Have shared goals and plans
    - Bring different prior experiences to the task
    - Different access to task relevant information
    - Must negotiate a common shared of action

- Pair Courage
  - "if it looks right to me and it looks right to you – guess what? It's probably right!"

How does this work (part two)?

- Pair Reviews
  - "Four eyeballs are better than two"
  - Continuous design and code reviews
  - Removes programmers' distaste for reviews
    - 80% of all (solo) programmers don't do them regularly or at all
- Pair Debugging
  - Explaining the problems to another person → "Never mind; I see what's wrong. Sorry to bother you."
- Pair-Learning
  - Continuous reviews → learn from partners techniques, knowledge of language, domain, etc.
  - Take turns being the teacher and the student minute by minute

Pairs means working together



XP is an iterative process

- Iteration = two week cycle (1-3 weeks)
- Plan each iteration in an iteration meeting that is held at the start
- Iteration is going to implement set of

  user stories

- Divide work into tasks small enough to finish in a day
- Each day, programmers work in pairs to finish tasks

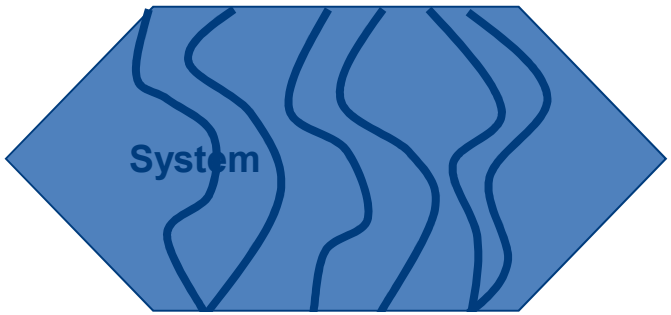
What are User Stories?

A user story represents

- ◊ a feature customers want in the software

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

- Written by our customers (communication w/ 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 he/she does the activity
- Write no more than one step on each card

Format of a User Story

- Title: 2-3 words
- Acceptance test
- Priority: 1-2-3 (1 most important)
- Story points (can mean #days of ideal development time)
- Description: 1-2 sentences (a single step towards achieving the 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.		

Ex. Acceptance Test for a Story

Create Receipt

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

Setup: The cashier has a new customer

Operation: The cashier scans three cans of beans @ $.99, two pounds of spinach @ $.59/lb, and a toothbrush @$2.00

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

XP is an iterative process

- Iteration = two week cycle (1-3 weeks)
- **Plan** each iteration in an iteration meeting that is held at the start
- Iteration is going to implement set of user stories
- Divide work into tasks small enough to finish in a day
- Each day, programmers work in pairs to finish tasks

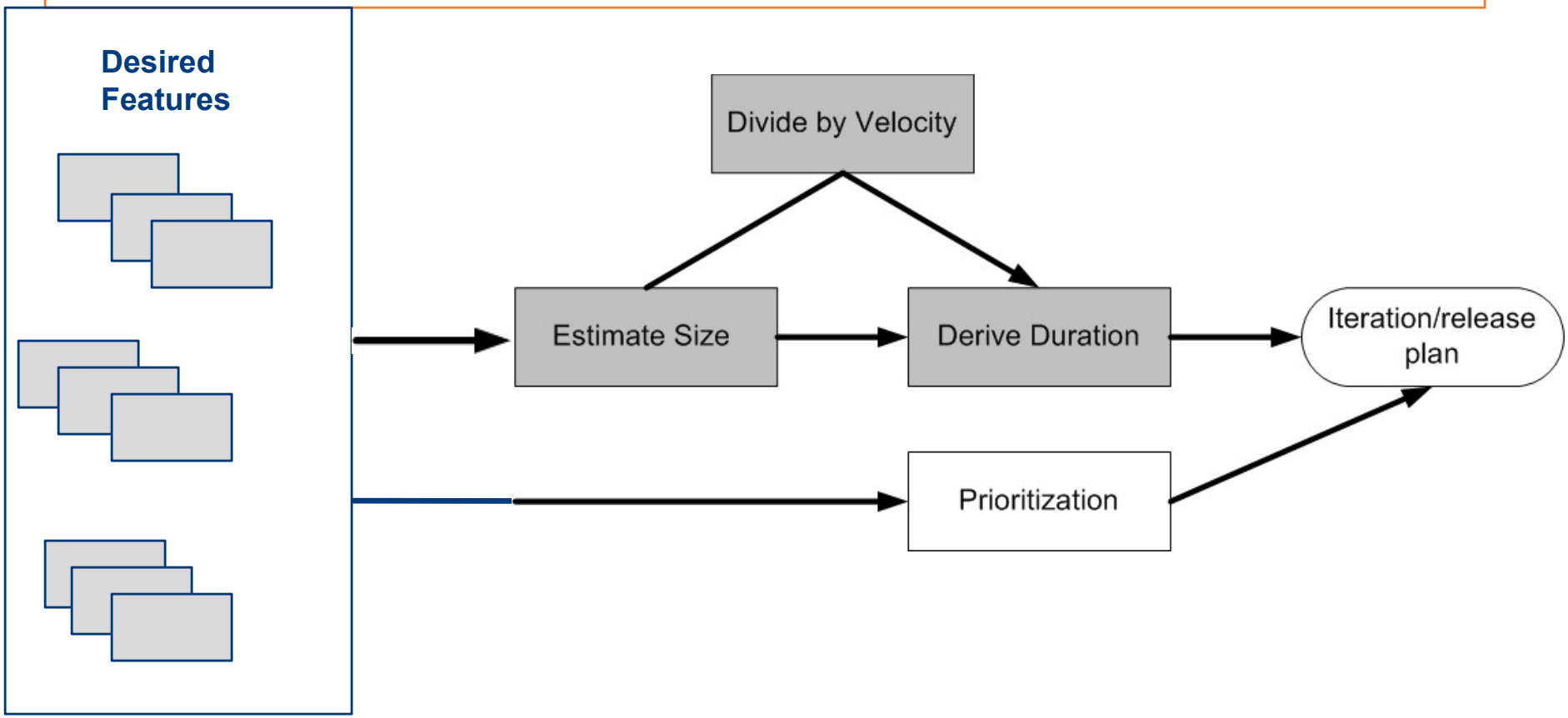
Estimating size: concepts

- Story point: unit of measure for expressing the overall size of a user story, feature, or other piece of work. The raw value of a story point is unimportant. What matters are the relative values.
  - Related to how hard it is and how much of it there is
  - NOT related to amount of time or # of people
  - Unitless, but numerically-meaningful
- Ideal time: the amount of time "something" takes when stripped of all peripheral activities
  - Example: American football game = 60 minutes
- Elapsed time: the amount of time that passes on the clock to do "something"
  - Example: American football game = 3 hours
- Velocity: measure of a team's rate of progress

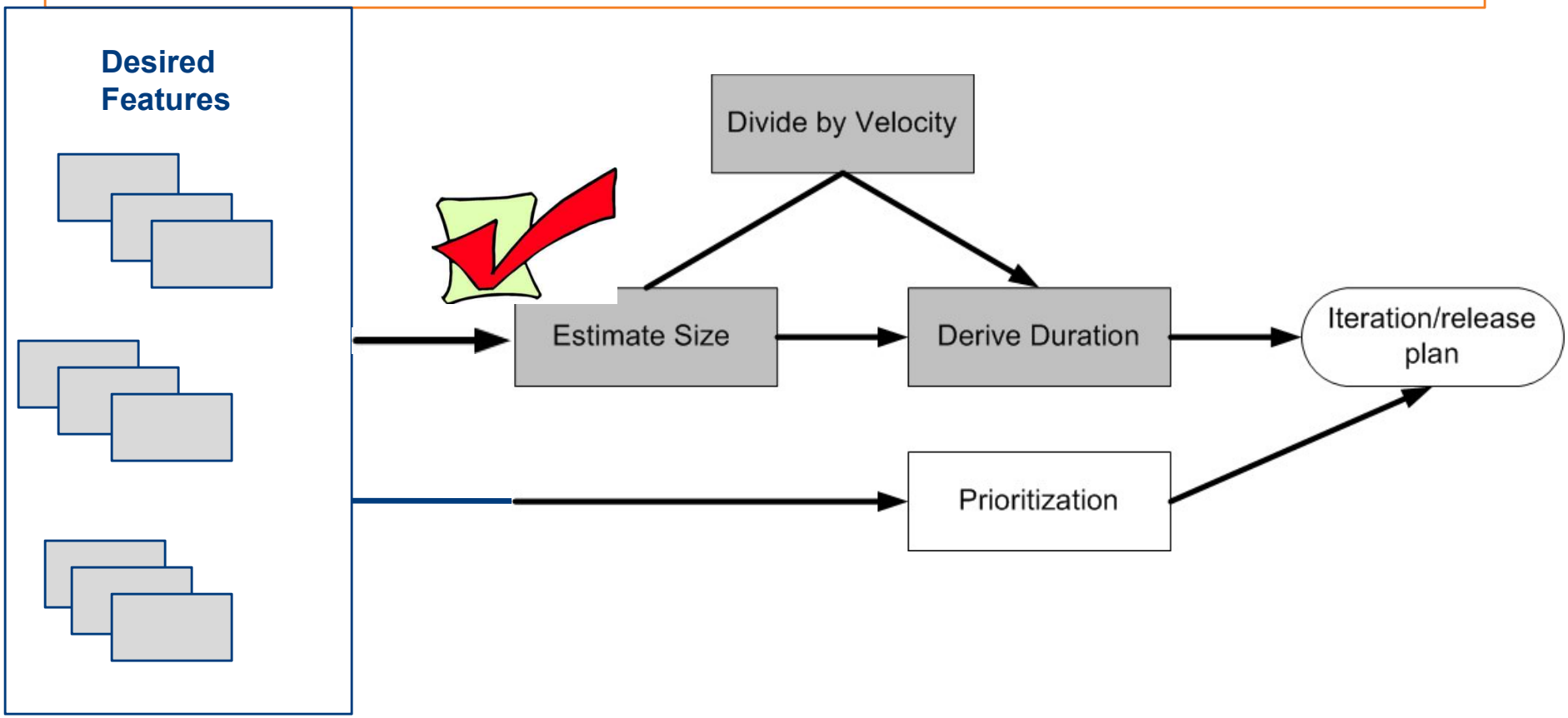
Priorities

- 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?”

Coming up with the plan



Coming up with the plan



Estimating story points

- Choose a medium-size story and assign it a “5”
- 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 in software development
  - Supporting current release
  - Sick time
  - Meetings
  - Demonstrations
  - Personal issues
  - Phone calls
  - . . . .
- When estimating ideal days, assume:
  - The story being estimated is the only thing you'll work on
  - Everything you need will be on hand when you start
  - There will be no interruptions

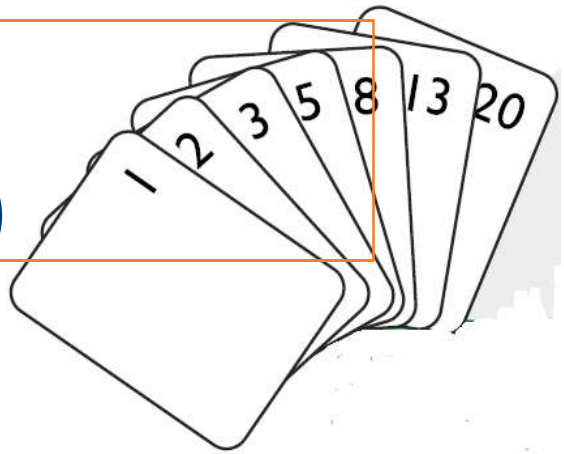
Ideal days vs. Story points

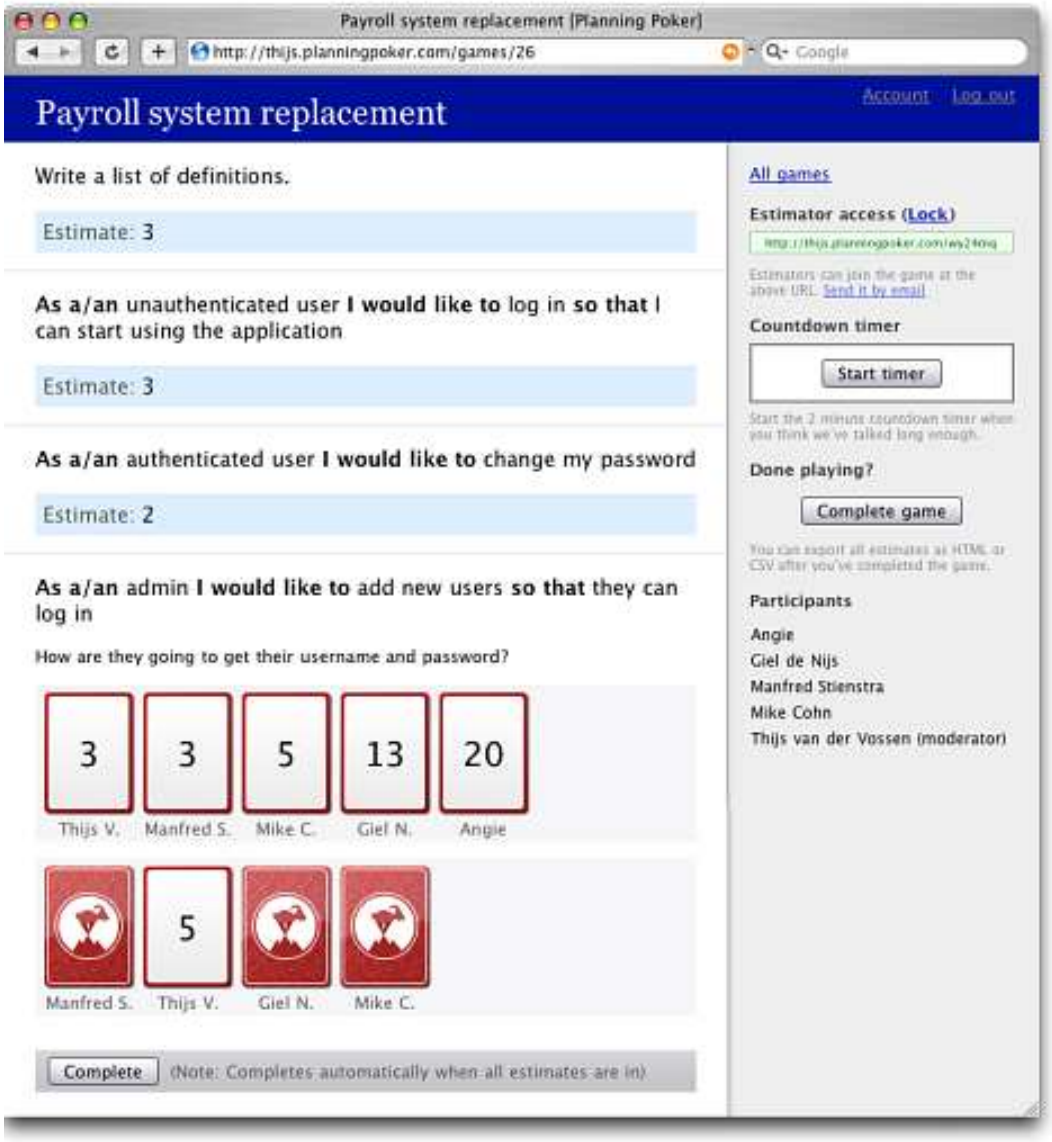
- Favoring story points:
  - Help drive cross-functional behavior
  - Do not decay (change based on experience)
  - Are a pure measure of size (focus on feature, not person)
  - Estimation is typically faster in the long run
  - My ideal days are not your ideal days (focus on person and their speed ☹)
- Favoring ideal days:
  - Easier to explain outside of team
  - Estimation is typically faster at first

Deriving an estimate for a user story

- Expert opinion
  - Rely on gut feel based on (extensive) experience
  - Disadvantage for agile: need to consider all aspects of developing the user story, so one expert will likely not be enough
- Analogy
  - Relative to (several) other user stories
  - Triangulation: little bigger than that “3” and a little smaller than that “8”
- Disaggregation
  - Break up into smaller, easier-to-estimate pieces/tasks.
  - Need to make sure you don’t miss any tasks.
  - Sanity check: does the sum of all the parts make sense?
- Planning poker
  - Combines expert opinion, analogy, disaggregation

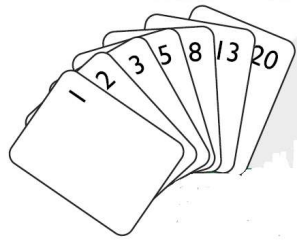
Planning Poker (http://planningpoker.com)





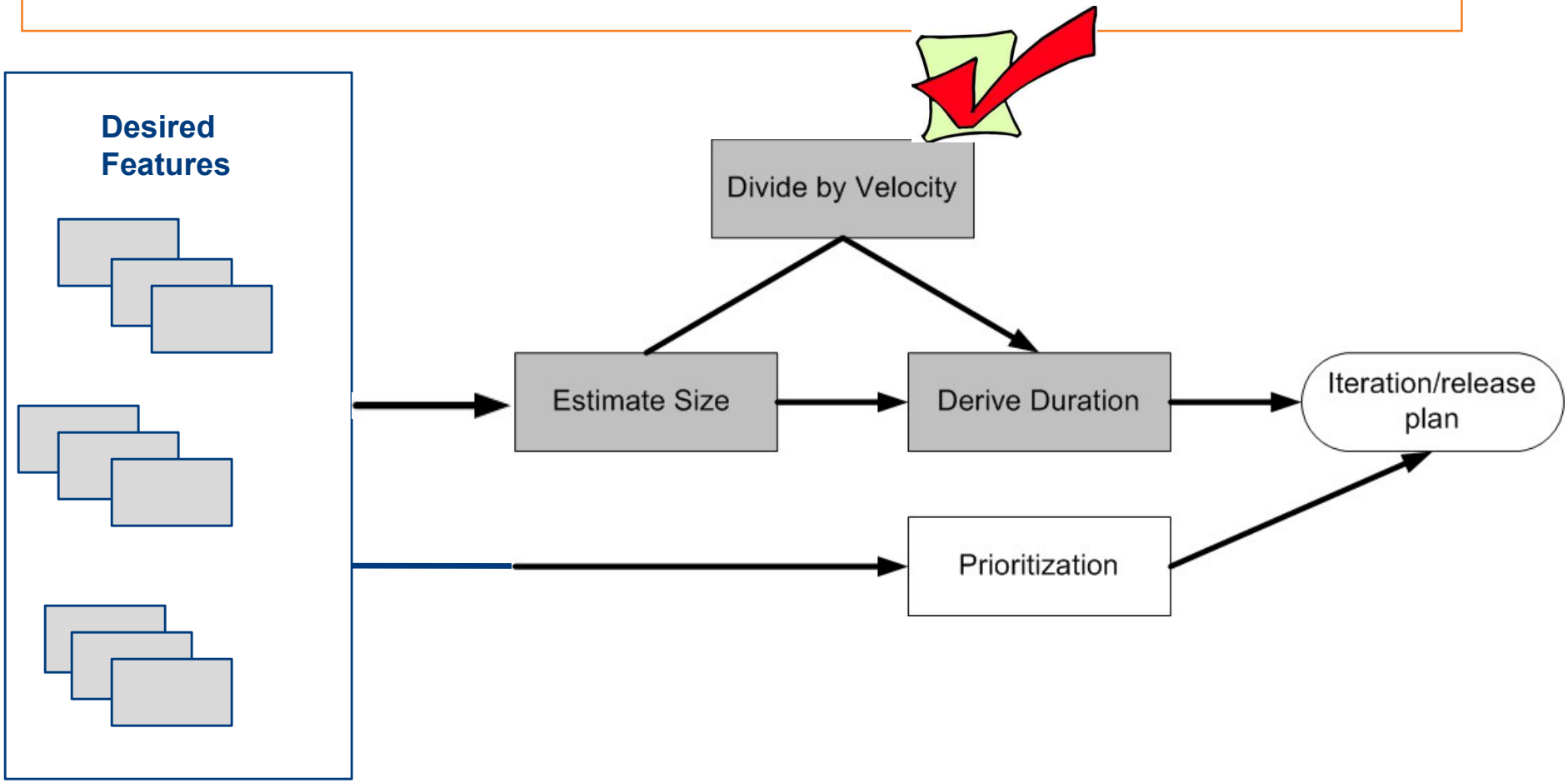
Playing Planning Poker

- Include all players on the development team (but less than 10 people overall):
  - Programmers
  - Testers
  - Database engineers
  - Requirements analysts
  - User interaction designers . . .



- Moderator (usually the product owner or analyst) reads the description and answers any questions
  - The team is given an opportunity to ask questions and discuss to clarify assumptions and risks.
- Each estimator privately selects a card with their estimate
  - Each individual lays a card face down representing their estimate for the story. Units used vary - they can be days duration, ideal days or story points. During discussion, numbers must not be mentioned at all in relation to feature size to avoid anchoring.
- All cards simultaneously turned over
- Re-estimate
- Repeat until converge
  - The developer who was likely to own the deliverable has a large portion of the "consensus vote"

Coming up with the plan

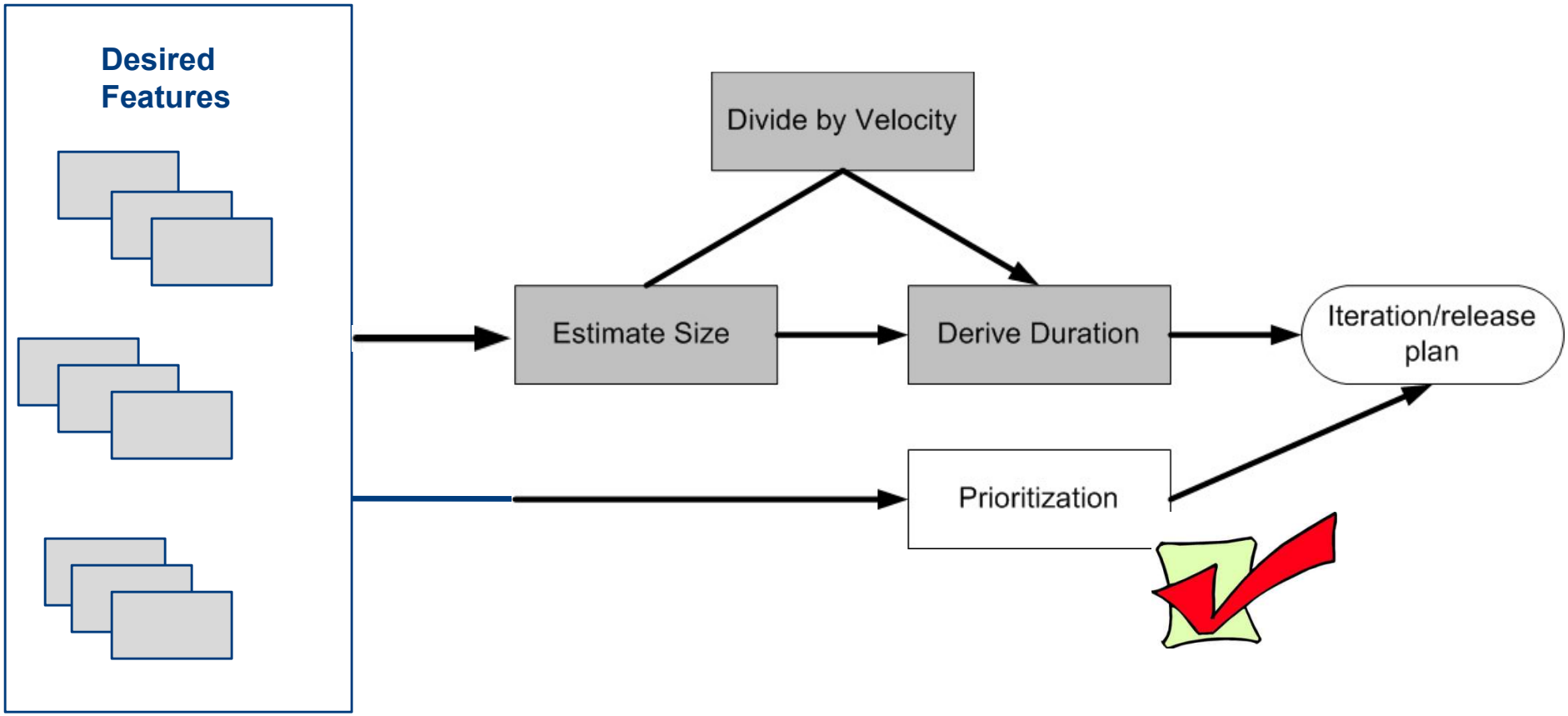


Velocity

- Velocity is a measure of a team's rate of progress.
- Velocity is calculated by summing the number of story points assigned to each user story that the team completed during the operation.
- We assume that the team will produce in future iterations at the rate of their past average velocity.
  - "Yesterday's weather"

http://agilesoftwaredevelopment.com/blog/pbielicki/predicting-team-velocity-yesterday-weather-method

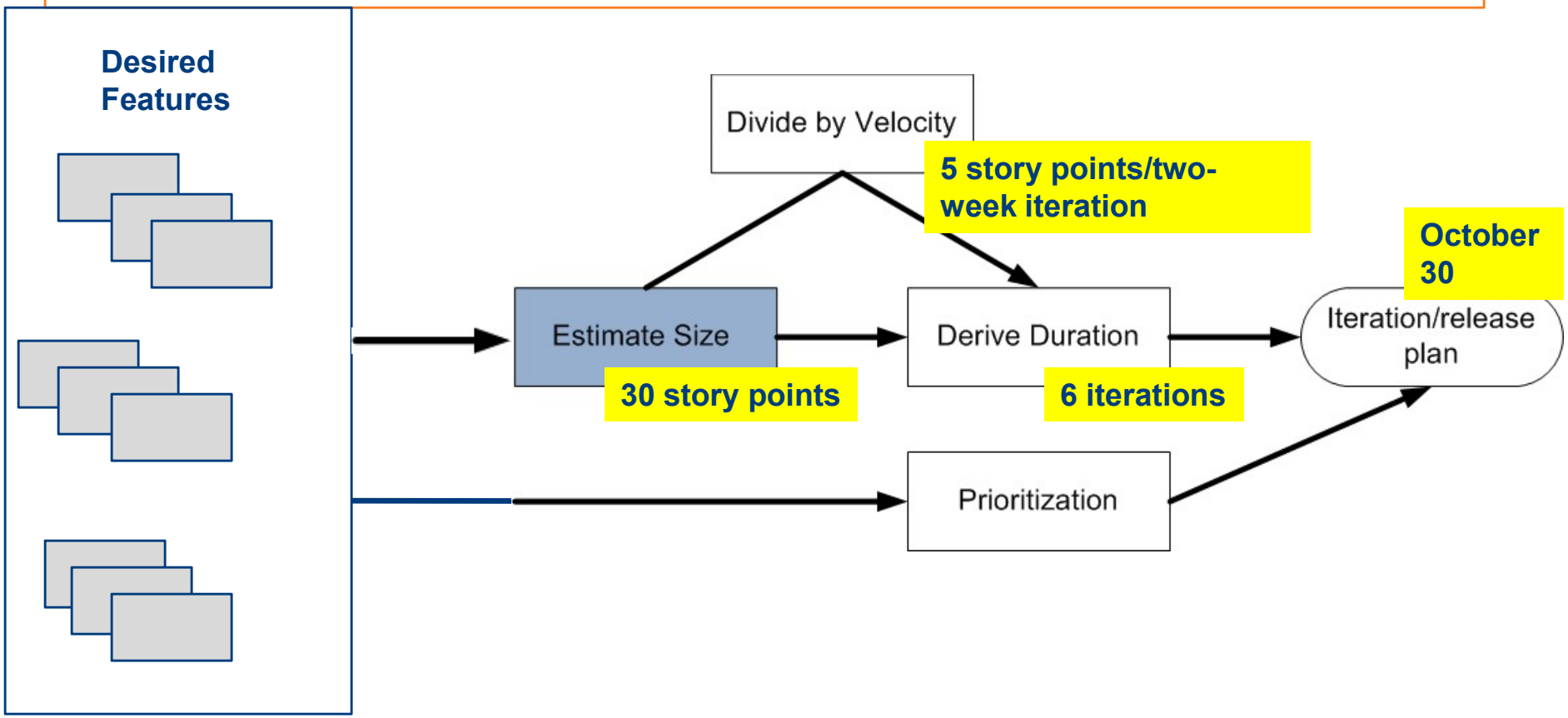
Coming up with the plan



Prioritization

- Driven by customer, in conjunction with developer
- Choose features to fill up velocity of iteration, based on:
  - Desirability of the feature to a broad base of customers or users
  - Desirability of a feature to a small number of important customers or users
  - The cohesiveness of the story in relation to other stories. Example:
    - "Zoom in" a high priority feature
    - "Zoom out" not a high priority feature
      - But it becomes one relative to "Zoom in"

Coming up with the plan



Planning game

- Customer writes user stories
- Programmers estimate time to do each story
- If story is too big, customer splits it
- Customer chooses stories to match project velocity
- Project velocity is amount of work done in the previous iteration(s)

Planning

- Programmers only worry about one iteration at a time
- Customer can plan as many iterations as desired, but can change future iterations

Simplicity

- Add one feature (user story) at a time
- Don’t worry about future stories
- Make program as simple as possible
- The simplest thing that could possibly work

Need educated customer

XP works best when

- Educated customer on site
- Small team
- People who like to talk
- All in one room (including customer)
- Changing requirements

Unit tests and refactoring

- Because code is as simple as it can be, adding a new feature tends to make it less simple
- To recover simplicity, you must refactor the code
- To refactor safely, you should have a rigorous set of unit tests

Working software

- All software has automated (unit) tests
- All tests pass, all the time
  - Never check in broken code
- 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.
  - Refactor (clean up)
  - Check in your code

One key practice

- Write tests first, then write code
- Various names
  - Test-first programming
  - Test-driven development
- Is it testing or designing?

- Degree to which you stick to it for MP?

Why test?

- Improve quality - find bugs
- 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

What is a test?

- Run program with known inputs, check results
  - Tests pass or fail
- Tests can document bugs
- Tests can document code
- Some terminology
  - Failure, error, fault, oracle

Who should test?

- Developer? Separate “quality assurance” group?
- Programmer? User? Someone with a degree in “testing”?

When to write tests

- During requirements analysis
- During architectural design
- During component design
- During coding
- After all coding

Timing

- Standard practice:
  - Worry about testing after you build the system
- Testers tradition:
  - Plan tests early, before code is written
- XP:
  - Write tests early, before code is written

XP Testing

- Write tests before code
- A design technique, not a testing technique
- Doesn’t find bugs, but eliminates them
- Doesn’t measure quality, but improves it

What kind of tests?

- Programmer tests / non-programmer tests
- Developer / Tester
- Unit tests / Integration tests / Functional tests / System tests
- Automated tests / Manual tests
- Regression tests
- Exploratory testing

New bugs or old bugs?

- Regression tests – test to make sure that everything that worked in the past still works

- Exploratory testing – look for new bugs
  - Name also used to contrast scripted testing

Regression tests: good

- Should be automated
- Set of tests that are rerun every time the software is changed
- Makes sure that things that are fixed stay fixed
- Each new bug results in an addition to the regression tests

Regression tests: can be bad

- Can take a long time to run
  - Select a subset
  - Remove obsolete tests
- Can be expensive to maintain
  - Changes to program can invalidate tests
  - Fix or delete?
- If it will never fail, why test it?

What kind of tests?

- Manual
  - Good for exploratory
  - Good for testing GUI
  - Manual regression testing is BORING
- Automatic
  - Test is a program
  - Test is created by a tool that records user actions

Test automation

- Tests are code or scripts (which is code)
- Real projects can often have more test code than production code
- Test code is boring
  - Build some complex data values
  - Run a function
  - Check the result

xUnit testing tools

- Programmer's testing tools
- Automated testing!
- Unit testing, but also integration testing and functional testing
- Regression testing

- Test-first design
- Each code unit requires several tests

JUnit

- Unit testing framework for Java
- Test is a method annotated with @Test (use JUnit 4, not JUnit 3), check assert