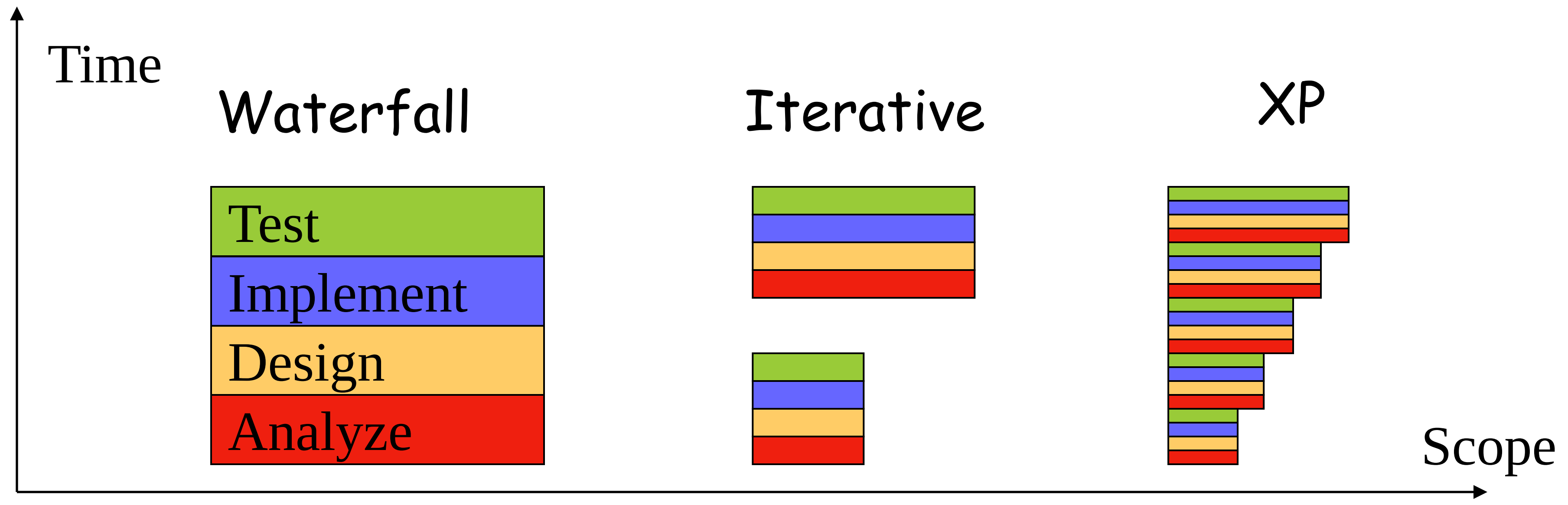
AGILE METHODOLOGIES: XP

Part II

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 active 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

No need to have all stories in first iteration

EXAMPLE: ACCOUNTING SOFTWARE

CEO: “I need an accounting software using which I can create a named account, list accounts, query the account balance, and 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 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 get list of

Can I delete if a 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 ge[illegible] list of

Can I delete if a 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

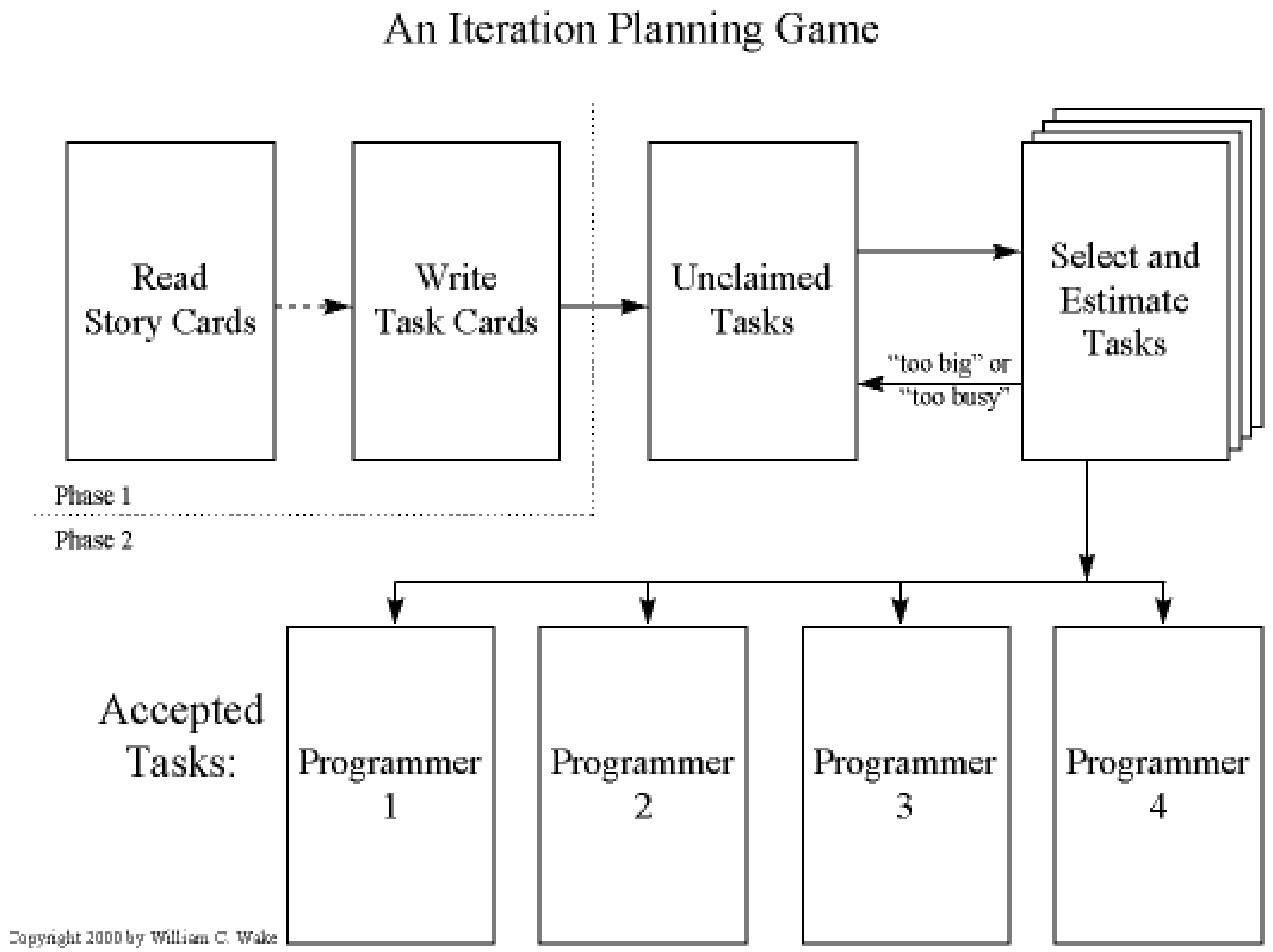
TASKS

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 subteam

Pick tasks => find completion date

Pick completion date, pick stories until you fill the budget

Customer might have to re-prioritize stories

TEST-DRIVEN DEVELOPMENT

Write unit tests before implementing tasks

Unit test: concentrate on one module

- Start by breaking acceptance tests into units

Think about names and calling conventions

Example of a test

```
addAccount("checking");
if(balance("checking") != 0) throw …;
try { addAccount("checking");
      throw …;
} catch(DuplicateAccount e) { };
```

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
- Courage to change 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

- Ensures that you will have a solid test suite

SIMPLICITY (KISS)

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; }

  … 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"

CONTINUOUS INTEGRATION

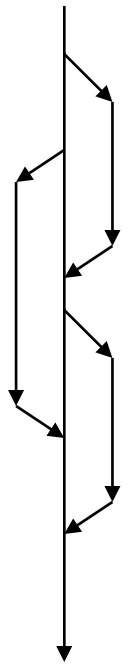
Integrate your work after each task.

- 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



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

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.
www.dilbert.com
scottadams@aol.com
© 2002 United Feature Syndicate, Inc.
1/9/03

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

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

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

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