CSc 131 – Computer Software Engineering

XP – Extreme Programming

Acknowledgements

- Roger Pressman: "Software Engineering: A Practitioner's Approach", ISBN-10: 0073375977

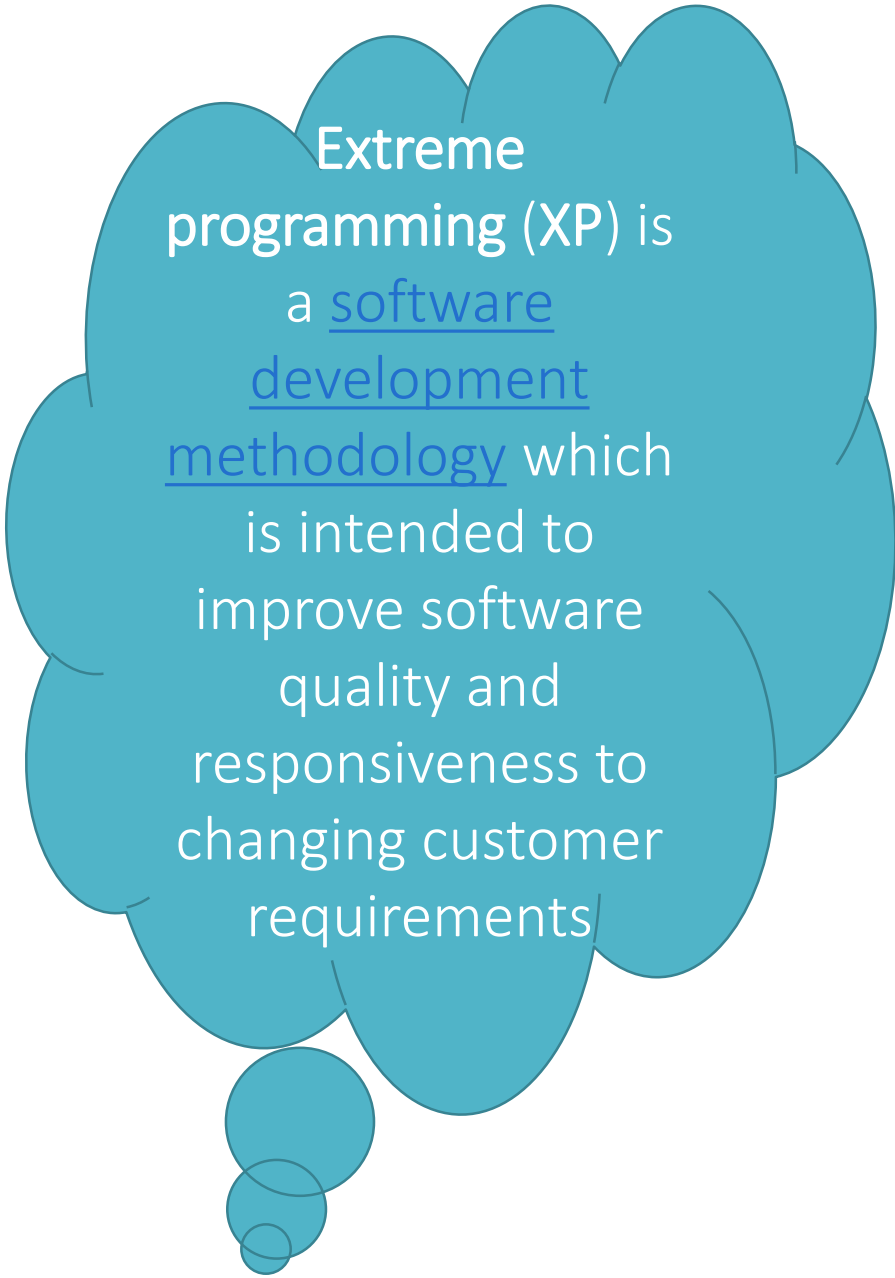
- These slides is also inspired by various courses available on-line that combine software engineering

  Ruzica Piskac's course at Yale

- Some materials were adapted from my previous CSC 230 course taught in Spring 2015

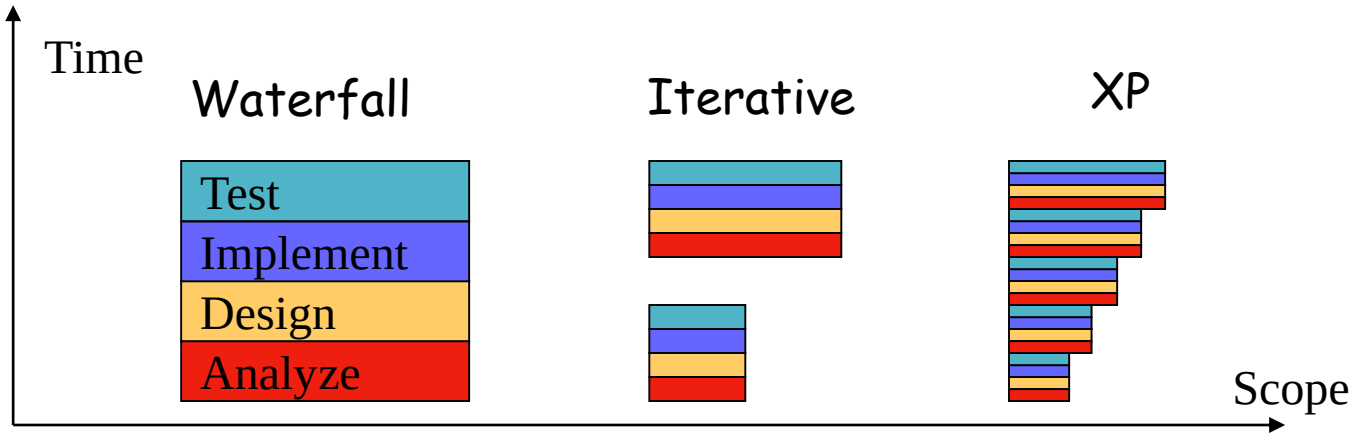
What is Extreme Programming?

- An agile development methodology
- Created by Kent Beck in the mid 1990's
- A set of 12 key practices taken to their "extremes"
- A mindset for developers and customers



Extreme Programming (XP)

XP: like iterative but taken to the *extreme*



XP - The 12 Practices

The Planning Game

Small Releases

Metaphor

Simple Design

Testing

Refactoring

Pair Programming

Collective Ownership

Continuous Integration

40-Hour Workweek

On-site Customer

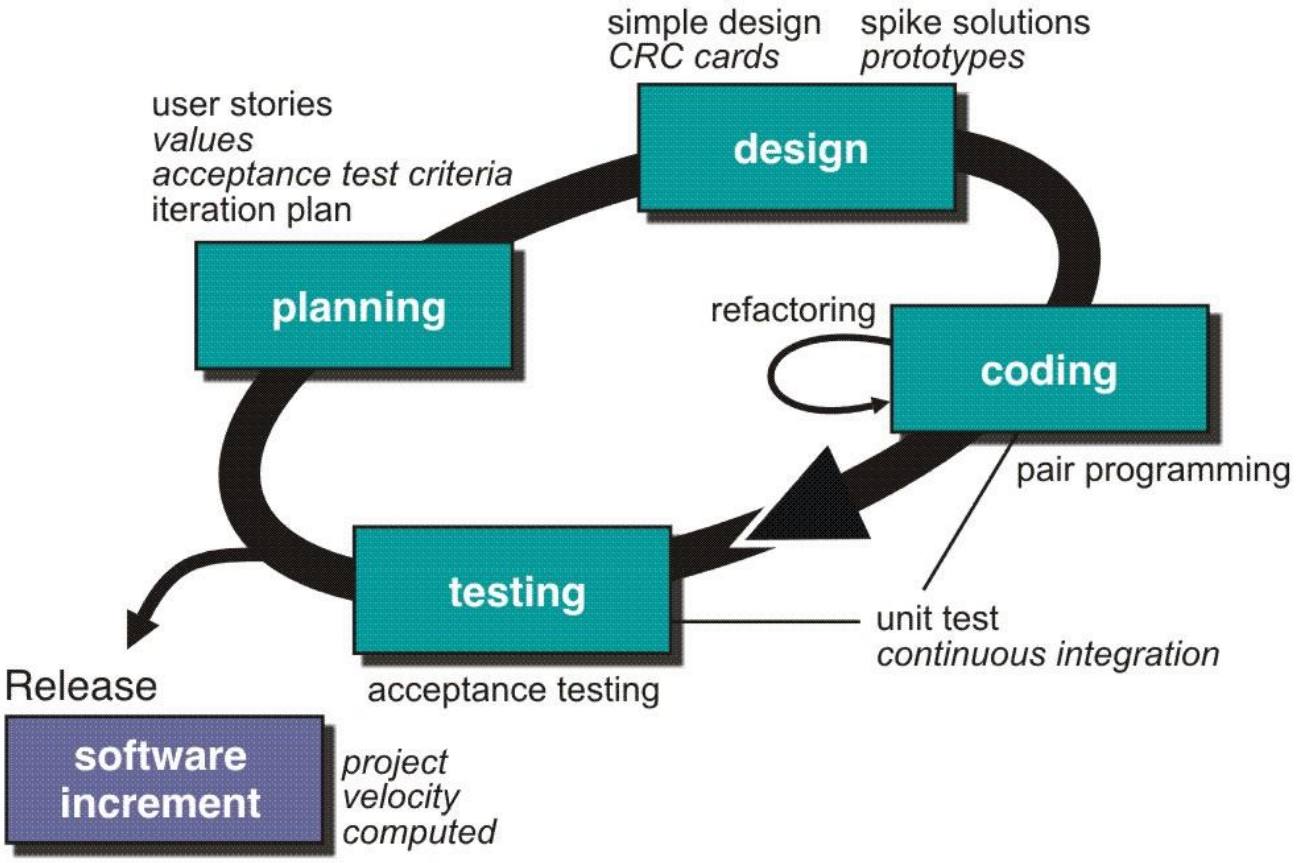
Coding Standards

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?

Extreme Programming (XP)



CSc 131 – Computer Software Engineering

XP – Extreme Programming

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?

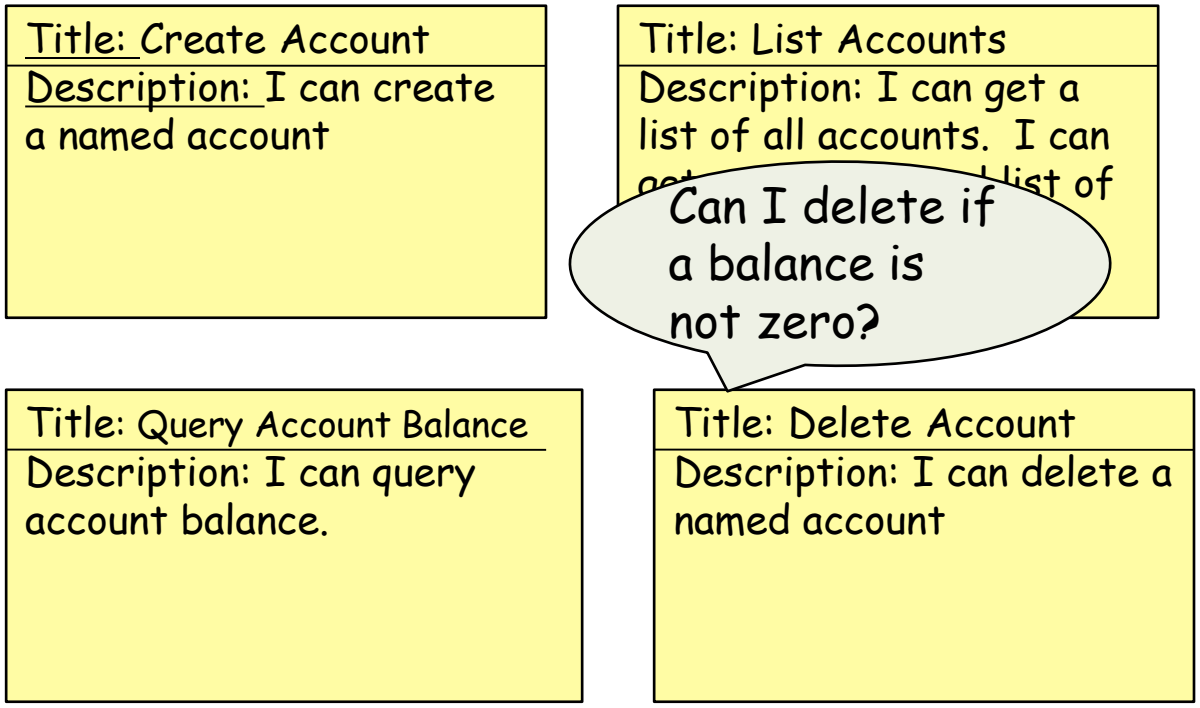
Title: Create Account
Description: I can create a named account

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

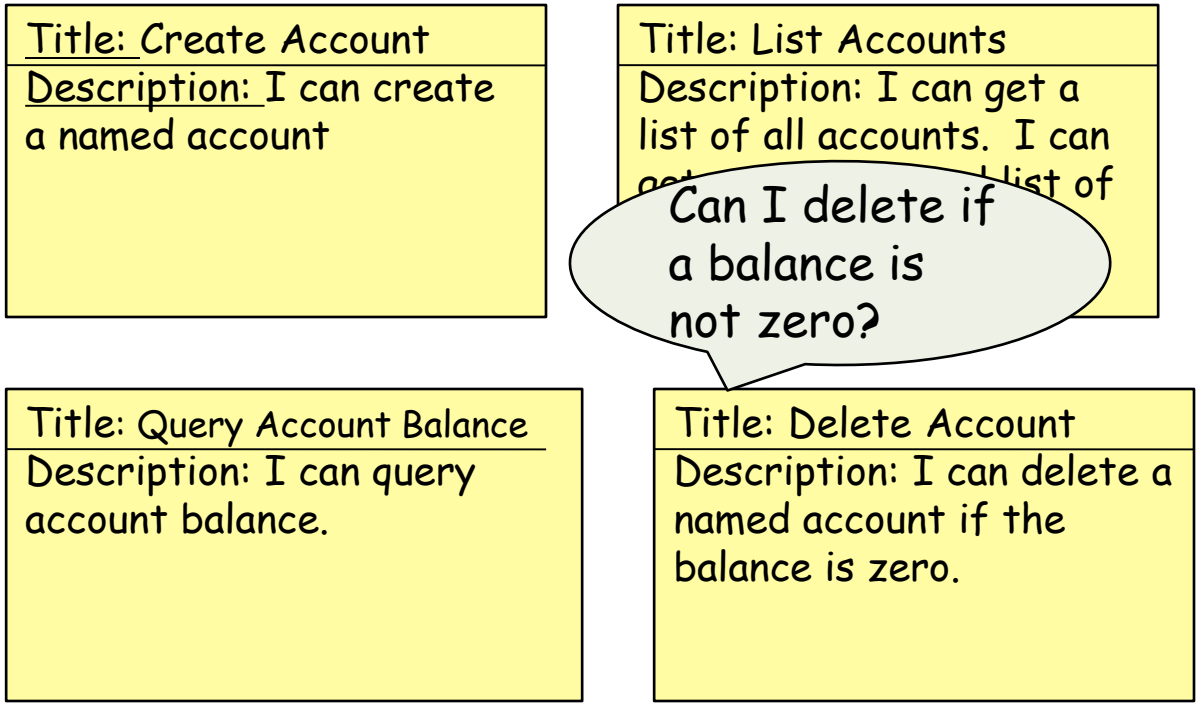
Title: Query Account Balance
Description: I can query account balance.

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

User Stories



User Stories



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?

Not a user story

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

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” (create/list stories)
2. If I now try to create “checking” again, I get an error (duplication !)
3. If now I query the balance of “checking”, I must get 0. (initialization)
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 (delete/list stories)

…

Automate Acceptance Tests

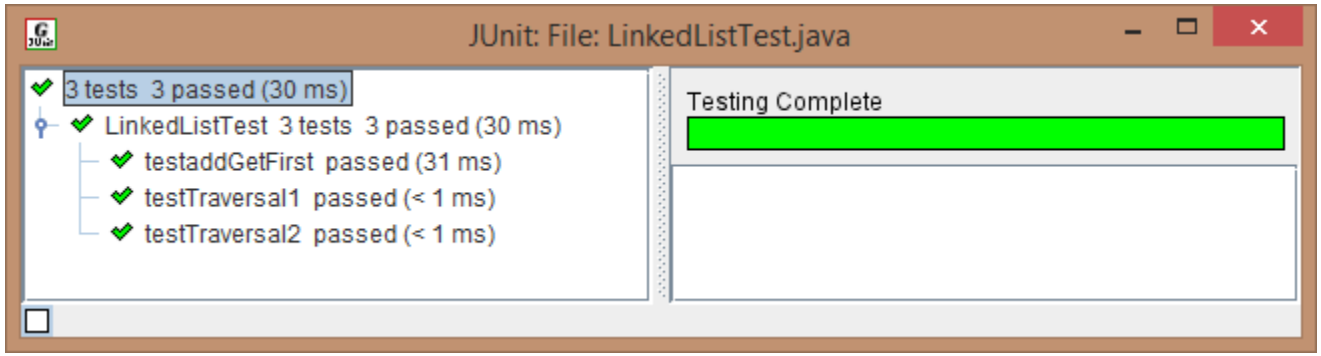
Customer can write and later (re)run tests

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

Example:



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

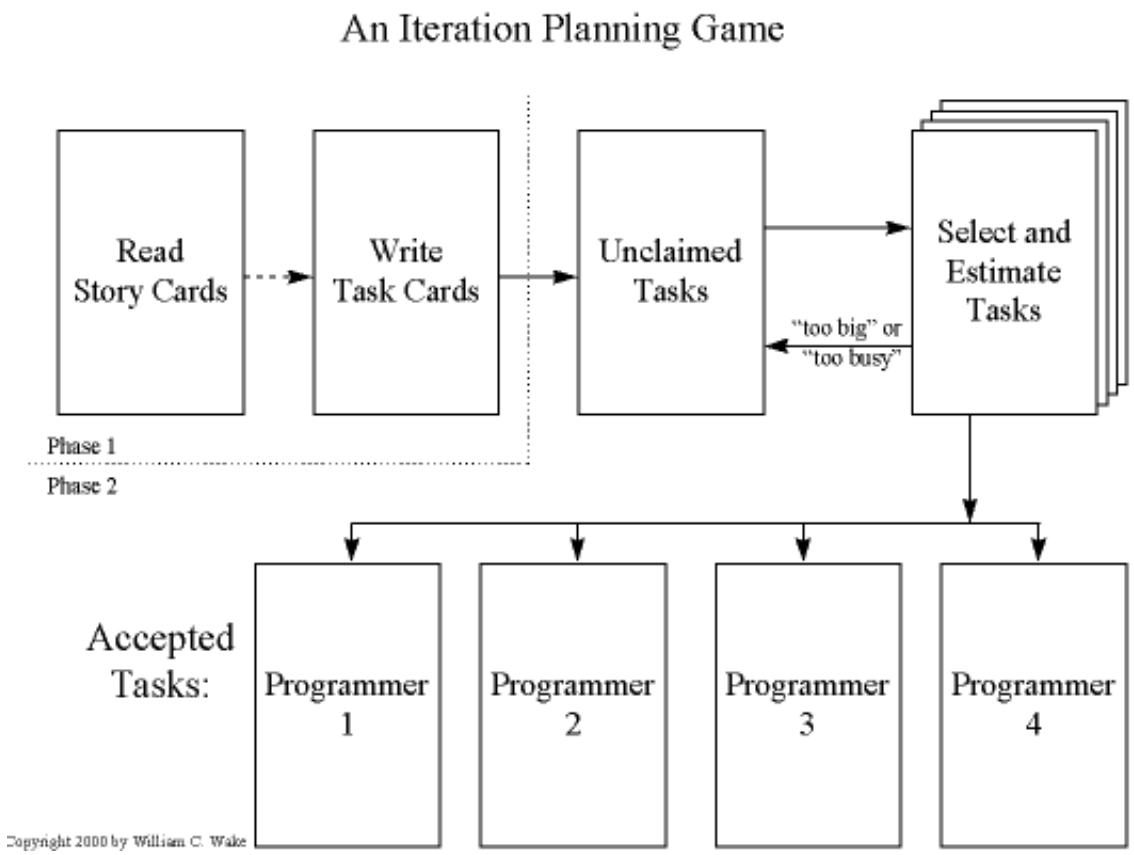
Tasks

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

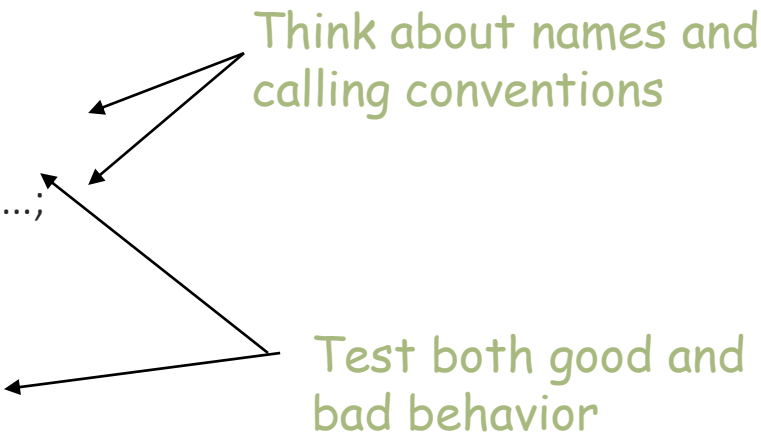
Write unit tests before implementing tasks

Unit test: concentrate on one module

Start by breaking acceptance tests into units

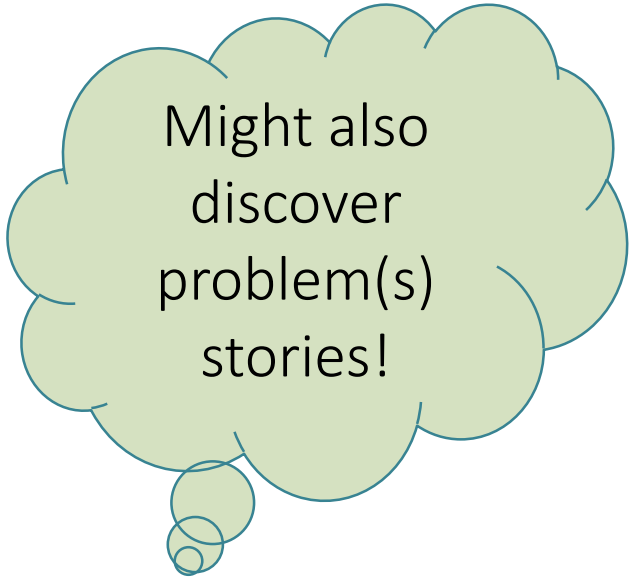
Example of a test

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



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

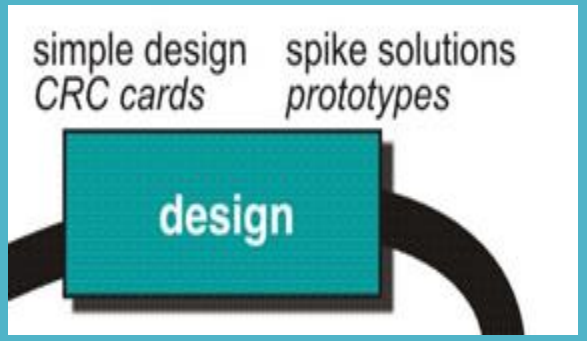


Test-Driven Development. Bug Fixes

- 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

CSc 131 – Computer Software Engineering

XP – Extreme Programming



Simplicity (KISS) – Keep it simple s...

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

Only implement features delivering value to the users

Avoid implementing **unnecessary features** confusing the users, putting constraints on and letting grow the code base

Save time for improving the existing code base and increasing the amount of tests

Minimize the risk of putting enormous effort into an app which eventually won't be used.

Class Responsibility Collaboration cards

- A CRC card is an index card that is used to represent the responsibilities of classes and the interaction between the classes.
- CRC cards are an informal approach to object oriented modeling.
- The cards are created from use-case scenarios, based on the system requirements.

A CRC card

Class name:	
SuperClasses:	
SubClasses	
Responsibility	**Collaborator**

A CRC card – Example (Front)

Front:

Class Name: Patient	**ID:** 3	**Type:** Concrete, Domain
Description: An Individual that needs to receive or has received medical attention		**Associated Use Cases:** 2

Responsibilities	Collaborators
Make appointment	Appointment
Calculate last visit	
Change status	
Provide medical history	Medical history

A CRC card – Example (Back)

Attributes:

Amount (double)

Insurance carrier (text)

Relationships:

Generalization (a-kind-of): Person

Aggregation (has-parts): Medical History

Other Associations: Appointment

Some advantages of CRC Cards

User participation increased: Users are actively involved in defining the model and thus increases their participation in understanding the system and creating it.

Breaks down communication barriers: Users and developers work togther side-by-side to create the CRC model.

Simple and Straightforward: CRC cards are easy to fill out and provide the essence of the system in a simple and straightforward manner.

Portable: CRC cards are easy to use and can be used anywhere.

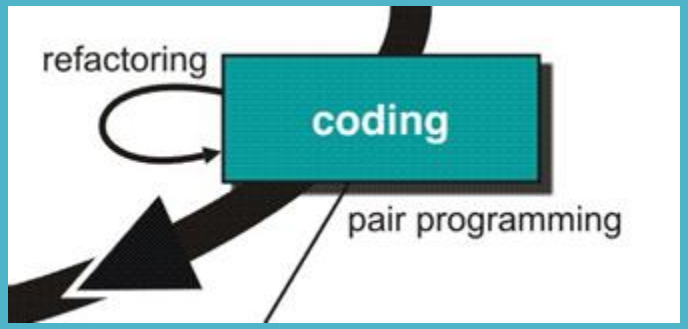
Class Diagrams: Class diagrams can be generated based on the CRC cards.

Life Cycle: CRC cards are useful throughout the life cycle.

Transition: CRC cards eases the transition from process orientation to object orientation.

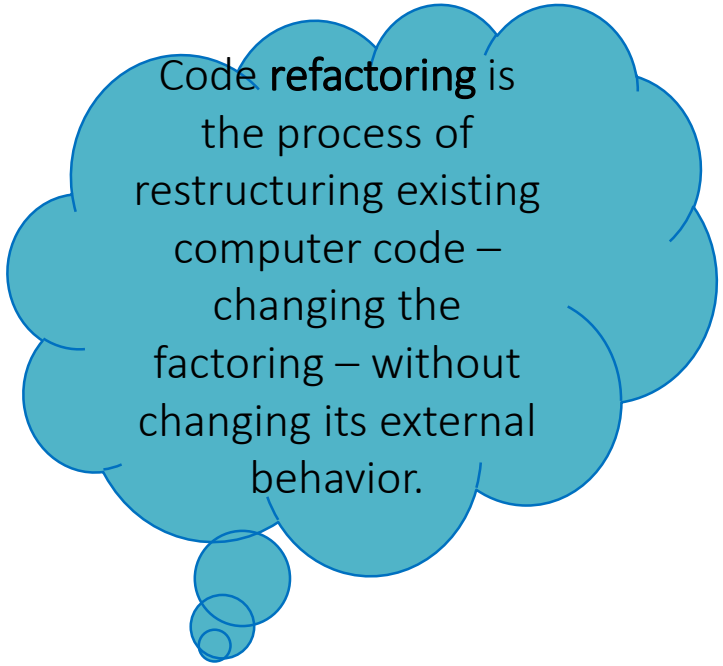
CSc 131 – Computer Software Engineering

XP – Extreme Programming



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

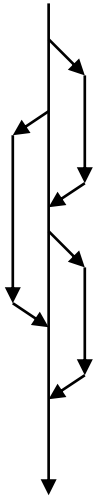
Integrate your work after each task.

- 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



Hudson: Continuous integration (**CI**) is the practice, in software engineering, of merging all developer working copies with a shared mainline several times a day.

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

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

Automate Unit Tests

Tests should be automated

To ensure they are run after each release

Example:

Result	Test Name	Project
Passed	ExpiresInSecondsTest	LoggingTests
Passed	LogLevelTest	LoggingTests
Passed	IsValidTest	LoggingTests
Passed	LoadTest	LoggingTests
Passed	RetrievedOnTest	LoggingTests
Passed	LogEnabledTest	LoggingTests
Passed	ExpiredTest	LoggingTests
Passed	IsWithinRange	LoggingTests
Passed	IsValidTest	LoggingTests
Passed	LoadTest	LoggingTests
Passed	IsValidTest	LoggingTests
Passed	ClientLogEntryConstruc	LoggingTests

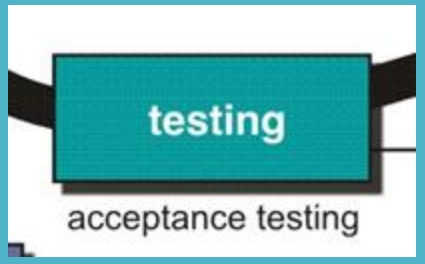
Automate Unit Tests (Sample Unit Test)

https://sacct.csus.edu/bbcswebdav/pid-2210871-dt-content-rid-7961889_1/courses/2155-CSC2001-50277/CsusStudentTest%281%29.java

```
// This Junit is used to validate students's lab 5
// It is also used to demostrate the functionality of Junit as student is first learning it.
// Author: Doan Nguyen
// Date: 6/11/15
import org.junit.Assert;
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class CsusStudentTest {

   /** Fixture initialization (common initialization
    *  for all tests). **/
   CsusStudent instance;
   Csc20Student instance2;
   @Before public void setUp() {
        instance = new CsusStudent("John Doe", 123, "123 Somewhere", "415-555-1212", "johndoe@somewhere.com");
        instance2 = new Csc20Student("John Doe", 123, "123 Somewhere", "415-555-1212", "johndoe@somewhere.com",true,15);
        instance = instance2;
        instance2 = (Csc20Student) instance;
   }

   // test getName
   @Test public void testGetName() {
         String expResult = "John Doe";
         String result = instance.getName();
         assertEquals(expResult, result);
   }

   // test getAddress
   @Test public void testgetAddress() {
         String expResult = "123 Somewhere";
         String result = instance.getAddress();
         assertEquals(expResult, result);
   }
```

CSc 131 – Computer Software Engineering

XP – Extreme Programming



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

CSc 131 – Computer Software Engineering

XP – Extreme Programming

Conclusion

What's Different About XP

No specialized analysts, architects, programmers, testers, and integrators

every XP programmers 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)

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