Week 6
More Software Engineering

Paul Chew

CS 212 – Spring 2004

Announcements

- Part 2B is due on Monday, Mar 15; we expect grades for Part 2A to be done this week
- Sections are being held next week (for questions on Part 2, Parsing, and Code Generation)
  - no meeting for W evening, Mar 3
  - M afternoon, Mar 8
  - M evening, Mar 8
  - W evening, Mar 10
- Regrades must be requested using CMS; regrade requests via email will be ignored
- Send email (to me) if you would like help finding a partner
- If you turn in Part 2B early, we can test it to make sure it compiles
  - We send email reporting compile-test results

Recall

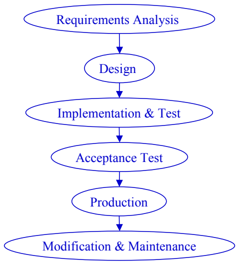
- Example programming-language abstractions
  - Procedural abstraction (static methods)
  - Data abstraction (classes & their methods)
  - Type abstraction (type hierarchy)
  - Iteration abstraction (Iterators)
  - Polymorphic abstractions (Java Collections Framework)
- An *abstraction* is distinct from its *implementation*
  - Can substitute one implementation for another without disturbing the “using programs”
- An abstraction has meaning only when it is *specified*
- *Validation*: Does the implementation match the specification?
  - *Verification* vs. *Testing*
  - For testing, need *drivers* and *stubs*
- *Debugging* is the usually the *most* time-consuming part of programming
  - Use *defensive programming*

Programming in the Large

- Last class, we mostly discussed how to implement a single module
- How do we design and implement a large program consisting of many modules?
- Topics
  - Models for software development
  - Requirements analysis
  - Data models
  - Program Design
  - Design patterns

Models for Software Development

- Waterfall model



Requirements Analysis

Design

Implementation & Test

Acceptance Test

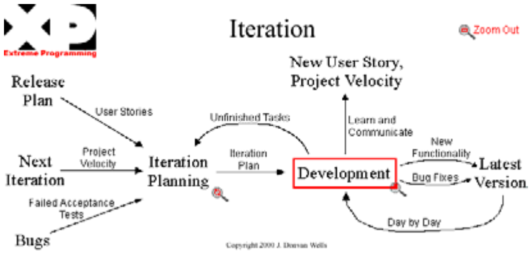
Production

Modification & Maintenance

- This model is idealized
  - True development is never entirely sequential
  - There is feedback from each stage of the process
- There are many other models for software development
  - XP, RUP, CMM, SCRUM, FDD

Another Model for Software Development

- This is a diagram from a website promoting *extreme programming* (http://www.extremeprogramming.org/)



Other Features of *Extreme Programming*

- All code is written in response to a *user story* (describes requirements on 4x6 card)
- Start with smallest set of useful features; release early and often
- Simple design: use simplest possible design that gets the job done
- Continuous testing
  - Tests are written *before* programming
  - When the tests are passed, the job is done
- Continuous integration: new code is added daily, but *all* tests must be passed
- *Pair programming*: two programmers at one machine

Pair Programming

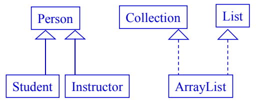
- Two programmers share one computer
  - One is the *driver*
    - Controls keyboard and mouse
    - Does all the writing of code
  - The other is the *observer*
    - Watches and guides
    - Focuses on strategic issues (e.g., how this module fits with others)
    - Is usually the *better* or *more experienced* programmer
- Claim: pair programming is *more productive* than having two separate programmers
- I've never tried it, but you might want to try this with your group

Requirements Analysis

- Requirements analysis consists of
  - Functional requirements
    - What is the program supposed to do?
    - How should the program respond to errors?
  - Performance requirements
    - How fast?
    - How much storage?
  - Determine delivery schedule
- It helps to create a *data model*
  - A diagram showing relations between important entities
  - The entities are mostly classes, but they don't have to be

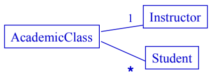
Data Models

- A data model defines
  - The kinds of data being manipulated
  - How they relate to one another
  - This can be shown as a graph
- *UML* (*Unified Modeling Language*) is one technique for diagramming data models
  - Each "class" is shown in a box with its (important) fields and methods
- In UML:
  - An open-headed arrow shows inheritance
  - A dashed open-headed arrow shows "implements an interface"

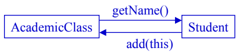


UML

- Composition
  - Edges without arrow-heads are used to show containment
  - The edge is labeled to show how many
    - 0..1 (0 to 1)
    - 1 (exactly one)
    - * (zero or more)



- Arrows with a closed head (and labeled with a method name) show who calls who



- Goal is to have a convenient picture showing relations between objects
  - Just a few pieces of UML were presented here
  - There are several books on the topic of UML
  - There are several other data modeling schemes

Program Design

- Design goals
  - Meet functional and performance requirements
  - The components are all good abstractions
  - The structure is relatively easy to implement and maintain
- Design is usually done iteratively
  - Select a target abstraction to implement
  - Identify useful helper abstractions (i.e., decompose the problem)
  - Specify behavior for the helpers
  - Sketch implementation plan for the target
  - Iterate

Top-Down vs. Bottom-Up Design

- Top-Down Design
  - Start with what is wanted
  - Determine what is needed to achieve it
- Bottom-Up Design
  - Start with what is implementable
  - Determine how these can be put together to achieve goal
- Top-Down design is usually more effective for all but small programs
- A rule to keep in mind
  - Avoid implementing an abstraction until its design is complete

Evaluating a Design

- A team conducts a *Design Review*
- Design Review: evaluating functionality
  - Explain how design captures the data model
  - Do a *walk-through* on symbolic test-data
  - Do this for entire design, and for individual modules or groups of modules
- Design Review: evaluating program structure
  - Each abstraction should be *coherent*
    - A specification with lots of &&'s or lots of ||'s might indicate a single procedure that is trying to handle several abstractions
  - Abstraction interfaces should be no wider than necessary

Testing

- Unit testing
  - Testing of a single module
  - If a unit fails to match its specification then it is considered to be incorrect
  - There are tools for unit testing
    - DrJava includes facilities for using JUnit (http://www.junit.org)
    - Simplifies the process of writing unit tests
- Integration testing
  - Testing of the entire program
  - Failure here may imply that the specifications are incorrect
  - Integration testing is usually harder than unit testing

Design Patterns

- These are (object-oriented) solutions to recurring design problems
- There are several books on this topic
  - *Design Patterns* by Gamma, et al.
  - *Java Design Patterns* by James W. Cooper
- There are two such patterns for traversing a tree (e.g., your AST)
  - the *Interpreter Pattern*
  - the *Visitor Pattern*

Coding Quality

- Pareto's Law
  - Due to Vilfredo Pareto, late 1800's
  - An 80/20 rule that shows up often
    - 80% of complaints are about 20% of the products
    - 80% of the decisions are completed during 20% of a meeting
- Software version: 80% of software defects occur in just 20% of the modules
- NSA study [Drake, IEEE Computer, 1996] on 25 million lines of code
  - 70-80% of problems were due to 10-15% of modules
  - 90% of all defects were in modules containing 13% of the code
  - 95% of *serious* defects were from just 2.5% of the code