Software Engineering

Lecture 5
CS 212 – Fall 2007

Software Engineering

- Engineering
  ABET: "the profession in which a knowledge of the mathematical and natural sciences gained by study, experience, and practice is applied with judgment to develop ways to utilize, economically, the materials and forces of nature for the benefit of mankind."
- Software Engineering
  IEEE: "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."
- Is Software Engineering a really a type of Engineering?

Engineering vs. Software Engineering

Engineering	Software Engineering
Ability to build from prefab components	Re-use is encouraged, but is not always practiced; designs are often "from scratch"
Uses metrics (i.e., measurements, as in physical units)	No clear physical units (although there is a concept of *software metrics*)
Tolerances are important	No real equivalent

Programming in the Large

- How do we design and implement a large program consisting of many modules?

- Topics
  - Abstraction
  - Specifications
  - Models for software development
  - Requirements analysis
  - Data models
  - Program Design

Decomposition

- Basic paradigm:

  "divide and rule"
  – Machiavelli
- Program is *decomposed* into smaller programs
- Decomposition goals
  - Each subproblem is at same level of detail
  - Each subproblem can be solved independently
  - The solutions can be combined to solve the original problem

Abstraction

- Abstraction:
  ignore certain details so that things that are different can be treated as if they are the same
  - Example abstractions
    - concept of a "file"
    - concept of "people"
    - concept of "mammals"
- High-level programming languages are based on abstractions
  - Strings
  - Lists
  - Dictionaries
  - BigNums
  - Matrices

Abstraction vs. Implementations

- An *abstraction* is distinct from its *implementation*
  - Can use it without knowing how it is implemented
  - Can implement it without knowing how it is used
  - Can substitute one implementation for another without disturbing the "using programs"
- Implies need for a description of the abstraction
  - A *specification*
- We'll talk about
  - Abstracting operations
  - Abstracting data
  - Abstracting types

Procedural Abstraction

- Almost all programming languages allow/encourage this
  - Called a procedure, a function, a subroutine, or (in Java) a static method
- Specification
  - Can be done formally or informally
  - An informal method might require comments at the beginning of each procedure describing what the procedure does
- Properties of good procedural abstractions
  - Minimally constraining
    - More freedom for the implementer
  - Simplicity
  - Generality
- Examples
  - A sort procedure that works on any int[ ]
  - A search procedure that works on any sorted int[ ]

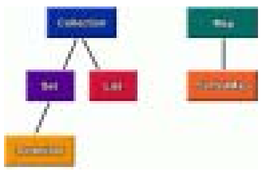
Data Abstraction

- Allows creation of new data types
  - Data abstraction = (object, operations)
  - This idea forms the basis for *object oriented programming*
- Details of the representation (i.e., the implementation) should be hidden
- Types of operations
  - Creators
    - new object "from scratch"
  - Producers
    - new objects from existing objects
  - Mutators
    - modify the object's state
  - Observers
    - provide information (without modification)

Abstraction Applied to Types

- Java allows type hierarchies (i.e., using *extends*)
- A supertype can be an abstraction of its subtypes
- Example:
  - Java Collections Framework (Set, List, and Map are subtypes of Collection)
- Benefits of hierarchy
  - Relationships among a group of types make it easier to understand the group as a whole
  - Code can be written in terms of supertype, but will work for all subtypes
  - Will still work when new subtypes are defined

Java Collections Framework



- Collection
  - isEmpty();
    contains(element);
    add(element);
    remove(element);
    iterator();
- Set
  - Same as Collection, but no duplicates allowed
- SortedSet
  - Same as Set, but includes first(), last(), and "ranges"
- List
  - Same as Collection, but includes positional access (access by index), search, list iteration, and "ranges"

Other Abstractions

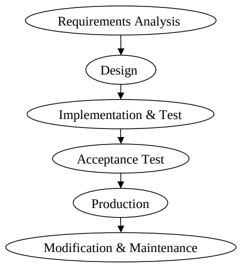
- Iteration Abstraction
  - Many data abstractions are *collections* of elements
  - Common need: "visit" each element of a collection
- In Java, this is accomplished by an operation that returns an Iterator (i.e., a subtype of the Iterator interface)
- Polymorphic Abstractions
  - *Polymorphic* ⇒ works for many different types
  - Example
    - The Java Collections Framework (ArrayList, HashSet, TreeSet, HashMap, …)

Specifications

- An *abstraction* has meaning only when it is specified
- A specification is a contract
  - Implementers agree to provide an implementation that satisfies the specification
  - Clients agree to rely on the specification and not the implementation
- Good specifications should be
  - Restrictive
    - Rule out implementations that are unacceptable to users (e.g., too slow, too much space, missing operations)
  - General
    - Don't rule out desirable implementations
  - Clear
    - Must be easy to understand

Models for Software Development

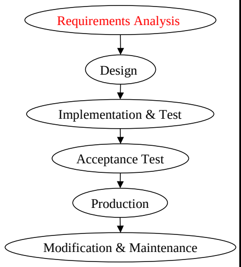
- Waterfall model



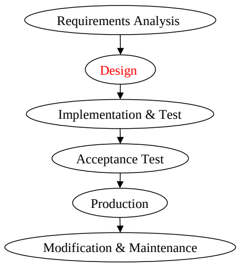
- This model is idealized
  - True development is never entirely sequential
  - There is feedback from each stage of the process
- There are many other models for software development
  - XP, RUP, CMM, SCRUM, FDD

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
    - When does the "customer" need it
    - How much time can we devote to it?
  - Additional requirements
    - Example: A game should be "fun"



Software 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:

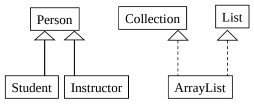
  Avoid implementing an abstraction until its design is complete

Data Models

- As part of the design, 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
- A data model defines
  - The kinds of data being manipulated
  - How they relate to one another
- One way to describe a data model is to use a graph (e.g., UML)

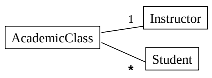
UML

- 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”

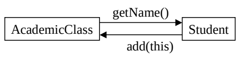


More UML

- 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

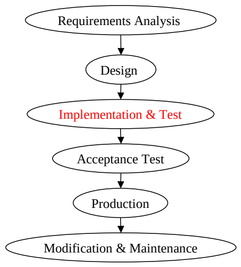


- Goal is to have a convenient picture showing relations between objects
  - The examples here showed just a few parts of UML
  - There are books on UML
  - There are several other data modeling schemes

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

Implementation & Testing



- Implementation:
  - Often the *least expensive* task
- Code Review:
  - Systematic examination of code intended to find and fix bugs at an early stage of development
- Testing:
  - Unit testing vs. integration testing
  - Glass-box testing vs. black-box testing

Another Testing Tool: Profiling

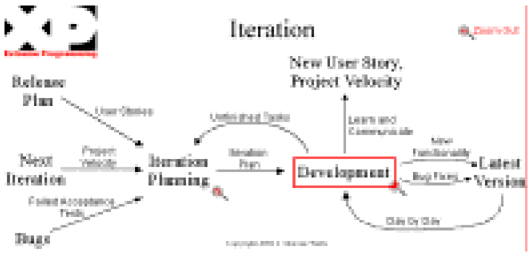
- People are notoriously bad at predicting the most computationally expensive parts of a program
  - Rule of thumb (Pareto Principle): 80% of the time is spent in 20% of the code
  - No use improving the code that isn’t executed often
  - How do you determine where your program is spending its time?
- Part of the data produced by a *profiler* (Python)

```
 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   2521    0.227    0.000    1.734    0.001 Drawing.py:102(update)
   7333    0.355    0.000    0.983    0.000 Drawing.py:244(transform)
   4347    0.324    0.000    4.176    0.001 Drawing.py:64(draw)
   3649    0.212    0.000    1.570    0.000 Geometry.py:106(angles)
     56    0.001    0.000    0.001    0.000 Geometry.py:16(__init__)
 343160    9.818    0.000   12.759    0.000 Geometry.py:162(_determinant)
   8579    0.816    0.000   13.928    0.002 Geometry.py:171(cross)
   4279    0.132    0.000    0.447    0.000 Geometry.py:184(transpose)
```

- Java has a built-in profiler (hprof); there are many others

Another Model for Software Development

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



Some Features of *Extreme Programming*

- All code is written in response to a *user story* (4x6 card describing requirements)
- 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

The Tower of Babel (1563). Pieter Bruegel the Elder