Week 5
Software Engineering

Paul Chew
CS 212 – Spring 2004

Announcements

- Part 2A is due on Monday
- Sections are being held this week (for questions on Part 2, Parsing, and Code Generation)
  - W evening, Feb 24
  - M afternoon, Mar 1
  - M evening, Mar 1
- Part 1 has been graded & most regrades are done
- In Part 2, the production statement -> expression ; will be more useful for Part 3 when we introduce functions
- If you turn in Part 2B early, we can test it to make sure it compiles
  - We send email reporting compile-test results

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 type of Engineering?

Engineering vs. Software Engineering

- Engineering
  - Ability to build from prefab components
  - Uses metrics (measuring systems, as in physical units)
  - Tolerances are important
- Software Engineering
  - Limited re-use; designs are often "from scratch"
  - Not really (although there is a concept of *software metrics*)
  - No real equivalent

Handling Large Programs

- 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[ ]

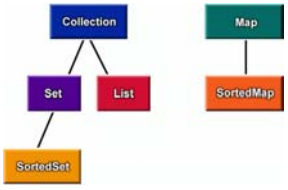
Data Abstraction

- 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., 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*)
- Can define a supertype whose properties are an abstraction of the subtype properties
- 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"

JCF Implementations

		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

- Vector and Hashtable appeared in earlier versions of Java
- They have been "retrofitted" to implement collections interfaces

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 (Vector, HashSet, TreeSet,...)

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
  - Users 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

Validation

- Validation:
  - Increase confidence that a program works is it is supposed to
  - Can be done via
    - Verification (i.e., proof)
      - Formal or informal
    - Testing
      - Black-box testing
      - Glass-box testing

Black-box testing: test cases are generated based on the specification without regard to implementation

Glass-box testing: examine implementation and attempt to test all possible "paths" through the program

Tools for Testing

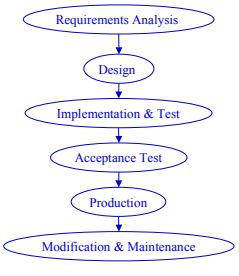
- Goal: automate as much of the testing as we can
  - Some parts can't be automated
    - Process of developing test cases is difficult and usually cannot be automated
  - But we can automate the testing process itself
    - Need *drivers* and *stubs*
- A *driver*
  - Calls the unit being tested and keeps track of how it performs
- A *stub*
  - Simulates a program-part that is *called* by the unit being tested
- Both can interact with a file or with a person
  - Example: a driver can read calling parameters from a file and save test results to another file

Debugging

- Debugging typically consumes *more time* than programming
- While debugging
  - A good starting goal: Find a *simple input* that causes the problem to occur
  - Look at intermediate values to localize the responsible code-region
- Goal: design, write, and document your programs in a way that makes them easy to test and debug
- *Defensive programming*
  - Include code that checks for errors
  - Include code whose only purpose is to examine intermediate results
  - Retain checks in the final version unless they are prohibitively expensive
    - The cost of an error is likely to be higher than the cost of the checks

Programming in the Large: Software Life Cycle

- Waterfall model



- This model is *idealized*
  - True development is never entirely sequential
  - There is feedback from each stage of the process

Other Views of the Software Life Cycle

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