Object-oriented software construction

Analysis/Requirements: What am I doing?
Design: Which classes and methods?
Implementation: How do I do it?
Testing: Does it work?
Maintenance: I just turn it in, right?

Software Construction I

- Programming is not just coding and debugging!
- Analysis
  - English description of what system models, to meet a requirement or specification
  - Usually involves working with non-programmer to develop detailed specifications
- Design: Measure twice, cut once!
  - divide & conquer: system is composed of smaller subsystems which in turn may be composed of even smaller subsystems
  - OOP: system is decomposed into a set of cooperating objects
  - Pseudocode: describe tasks, subtasks, and how they relate
  - hand-simulation: desk-check pseudocode by stepping through it without using a computer
  - Often use diagrams to better communicate the structure of the system
  - Often flowcharts in procedural languages and UML diagrams in OOP

Software Construction II

- Implementation
  - Pick a language
  - emphasize good problem decomposition, well structured design, and readable, well-documented programming style
  - may need to backtrack and redesign
- Testing
  - submitting input data or sample user interactions and seeing if program reacts properly
  - typically done in stages, starting with individual components and working up to subsystems, and eventually the entire program
  - bugs: errors in code or design
  - debugging: process of removing program bugs
Software Construction III

- Maintenance
  - keep the program working and current
  - Good documentation is a must, as is a good initial design

- Software development not usually as linear as waterfall model
  - Spiral method: go back during implementation to change the design, go back during testing and debugging to change implementation
  - Incremental development (accretion): emphasis on designing, implementing, and testing a small version of the system, and then repeating these steps for larger and larger versions of the system
  - Extreme programming: emphasis on teams, flexibility, and writing test code first!

- What makes a program “good” or a “horror”?

What makes code easy to test/debug/maintain?

- Abstraction: The ability to view a complex operation in a simplified form
  - Good classes are abstract representations of objects in the problem domain
  - Good utility/library classes have a “functional” problem domain

- Method Cohesion: the degree to which a method implements a single function
  - Each method should do a well specified task with no surprises or side-effects
  - Methods should have high cohesion

- Class Cohesion: the degree to which the methods of a class deal with a specific abstraction
  - Classes should have high cohesion

- Coupling: the degree to which a method depends upon the specific implementation of other methods
  - Methods that share global data in an array field, for instance, are “coupled”. Neither method can change the way in which the data is stored without necessitating code change in the other method.
  - Classes should have loose coupling

Pre/Post conditions

- Precondition: A condition that must be true about the state of the system (input parameters, instance variables, etc.) before executing a method
  - Example: say we want a letter from a-z. We use type char for the input parameter. Our precondition is that the character is from a-z and not punctuation or some other legal value of type char.

- Postcondition: condition that must always be true just after the execution of a routine.
  - Example: the result of a factorial operation must always be greater than one

- Checking input from users is generally a necessary step in satisfying method preconditions
Defensive programming

* Enforce pre/postconditions!

```java
public static double divide (int numerator, int denominator) {
    if (denominator == 0) {
        System.out.println("divide error: zero denominator");
        System.exit(1);
    }
    return (double) numerator/denominator;
} // end divide
```

* Assertion statements simplify test code
  – Include -ea or --enableassertions runtime/VM flag to enable

```java
public static double divide (int numerator, int denominator) {  // requires -ea VM flag
    assert denominator != 0;
    return (double) numerator/denominator;
} // end divide
```

**OO Design: Defining the classes**

* Identify all the nouns in the problem description / analysis / requirements
  – Each of these is a potential class or object instance

* Refine the list to identify the classes
  – Some of the nouns may mean/represent the same thing
  – Some nouns used may be beyond the scope of the problem
  – Some of the nouns may represent objects, not classes
  – Some of the nouns may represent values that can be stored in a primitive variable and thus do not require a class.

* For each putative class
  – Determine what the class knows (instance variables): generally nouns
  – Determine what the class does (methods): generally verb phrases

* Construct UML diagrams

* Refactoring: Come back and improve your design anytime!

**Documentation**

* Internal code documentation should consist of:
  * For each class header
    – A description of the abstract function of the class
  * For each method header
    – A description of the abstract function of the method (often, this is just the “pseudo-code” description of the method) and parameters
    – Pre-conditions (if any)
    – Post-conditions (if any)
  * In addition to header, internal documentation may require:
    – Descriptions of variables, but only if a variable’s name cannot be selected to make its role crystal clear
    – An in-line comment heading each block of code that performs a high level step in the routine’s pseudo-code algorithm
Development: Implementation & Test

* For most programmers, these two steps are inseparable and cyclic.
* For each class:
  - Determine the instance variables
  - Write pseudocode for the methods
  - Write test code for the methods
  - Implement the methods
  - Test each as you develop them
  - Test the class fully
  - Debug and reimplement as needed
  - Move on to the next class.
* Develop in iteration cycles
* Keep it simple-stupid (KISS)
* Use “stubs” for top-down design, develop bottom-up where necessary

Requirements for Text Battleship

* We are going to build a text-based version of the game “Battleship”.
* Our gameboard is going to be a 2d 5x5 grid that prints as follows:
  - Rows are alphabetic: A-E
  - Columns are numeric: 1-5
  - ‘~’ represent unknown sea
  - ‘.’ represent previously targeted miss
  - ‘x’ represents a previously targeted hit
  - ‘*’ represents a previously targeted vessel that has been sunk.
* Input will simply be a coordinate for the next shot (e.g., A2 or E1)
* There will be two ships, size 2 (2x1 or 1x2), and size 3 (3x1 or 1x3).
* The player loses the 5th time that they make a clean miss (“o”).
* The player wins if they sink all of the ships before they loose.
  - Show the surviving ship locations! (use ‘o’ in display)