Software Engineering

- Program = Algorithm + Data Structures
- Software Lifecycle
- Systematic approach to problem solving/program development

**Lifecycle**

- Software is not physical, doesn’t wear out
- 70% of life is spent in maintenance
- GOAL: ease of modification

**program = algo + data**

- program is An Embodiment of a solution
- what is done = algorithms
- what receives actions = data (objects)
- Abstract Data Type = collection of data and operations valid on the data
Coupling & Cohesion

- Coupling - degree of interdependence of modules (objects)
- Cohesion - degree of singleness of purpose of a module
- Both contribute to modularity
- Modularity leads to ease of modification

Phases

- Analysis / Specification
- Design / Verification
- Implementation
- Test

Analysis

- What will we build?
- Increase precision and detail in spec.
  - ex: cold v. 50°F
- Who are the users? What input data?

Design

- What are the classes?
- Who needs what data?
  - Data flow
- What does a method do?
  - Pre/post-conditions
**Contract**

```java
/**
 * Sets value of denominator to den.
 * pre: den > 0
 * post: denominator == den
 */
setDenominator(den)
```

Say WHAT is done, not HOW.

**Verification**

- Basis for testing
- Assertions
- Loop Invariants

**Assertion**

```java
while (count > 0) {
    // something happens here
    Assert.assertTrue(count <= 0);
```

- claim about the state of computation at a certain point in the algorithm

**What can you assert?**

```java
int aNumber = 64;
int theRoot = Math.sqrt(aNumber);
Assert.assertTrue(a);
List aList = new List();
Assert.assertTrue(b);
aList.fill(25); // fill with 25 rnd nums
Assert.assertTrue(c);
aList.sort(); // sort the list
Assert.assertTrue(d);
```
Demo JUnit with Eclipse

Loop Invariants
- loop terminates
- true before & after each iteration
- this includes BEFORE the loop starts
- this includes AFTER the loop ends

```java
// compute the sum of item[i], i=0..n-1
// n >= 1
int sum = 0;
int j = 0;
while (j < n) {
    sum += item[j];
    ++j;
}
```

sum = 0 and j = 0 initially.
1: sum = 0 + item[0] and j = 1.
2: sum = sum + item[1] and j = 2
   = 0 + item[0] + item[1] and j = 2.
finally: sum = sum + item[n-1] and j = n.
invariant: sum = item[0] +...+ item[j-1]

WHY?
Invariant: \( \text{sum} = \text{item}[0] + \ldots + \text{item}[j-1] \)

Why?

Before any loop: \( \text{sum} = 0 \) and \( j = 0 \).
Inv: \( \text{sum} = \text{item}[0] + \text{item}[0-1] = 0 \), true.

After an iteration? Assume \( j = k \).
Then \( \text{sum} = \text{item}[0] + \ldots + \text{item}[k-1] \)
Loop occurs, after which \( j = k + 1 \), but we have added \( \text{item}[k] \) to \( \text{sum} \), so
\( \text{sum} = \text{item}[0] + \ldots + \text{item}[k-1] + \text{item}[k] \)
Since \( j = k + 1 \) then \( k = j-1 \) so
\( \text{sum} = \text{item}[0] + \ldots + \text{item}[j-1] \), inv is true.

- Loop terminates?
- Loop invariant matches the semantics / documentation?

Invariant doesn’t match

- Suppose while \( (j <= n) \) instead?
- Inv: \( \text{sum} = \text{item}[0] + \ldots + \text{item}[n] \) because after loop terminates, \( j == n+1 \).
- This is an off by one error.

Good Solution?

- A priori: must be correct
- Traditionally: operational costs such as fast, low memory requirements, low end-user costs (training)
- Expanded: total costs over all phases are appropriate (book: minimal)
Solution Efficiency

- IMPORTANT cost. Could be deal breaker.
- algorithm's time/space complexity
- data structure choice

Object Oriented Design

Abstract Data Type

- suppression of detail irrelevant to domain of discourse
- procedural: WHAT not HOW (method)
- data: hide organizational details (array)

- collection of data + valid operations
- encapsulates data and operations
- hides data and operations that are not needed by clients
Verb-Noun Analysis

- Given a specification of a system, mark all the nouns, and separately all verbs
- Nouns are potential objects
- Verbs are potential methods

OOP

- Class = ADT, rules for creating objects
- Data and methods are encapsulated via object
- Objects can inherit properties
- Polymorphism delays message/method binding to run-time

Programming Issues

- Don’t mistake Moccasins For Expensive Shoes
- Debugging, Modularity, Modifiability, Fail-Safe, Style

Debugging

- Fault isolation. JUnit helpful here.
- Tracing execution. Debuggers, scaffolding.
- Easier if code is modular and readable
Modularity
- Use classes, methods, private fields
- Minimize coupling
- Maximize cohesion

Modifiability
- Use methods!
- Use parameters!
- Use named constants!
- Don't depend on internal details of other modules!

Fail-Safe
- Validate input data and parameters
- Methods should check invariants
- Methods should enforce preconditions

Ease of Use
- User should know what is happening at all times (e.g., prompt for inputs, display outputs clearly)
- What is simple should be easy. What is complex should be possible.
Style
• Readability is key.
• INDENT consistently.
• Match braces.
• Use vertical and horizontal white space to show which statements belong together.
• Use the “FORMAT” item in Eclipse

Procedural Abstraction
• specify WHAT is done, not how

Style
• Use JavaDoc.
• Web-Cat will check these items.