CS1530, LECTURE 11

IMPLEMENTATION

WHAT IS IMPLEMENTATION?

- Turning the design of a system into code
- What many people think software engineering consists of
- And indeed it is important! We will spend much of the rest of the term discussing implementation details and techniques

WHAT DOES IT MEAN FOR AN IMPLEMENTATION TO BE "GOOD"?

CHARACTERISTICS OF A "GOOD" IMPLEMENTATION

- Readability
- Maintainability
- Performance
- Traceability
- Correctness
- Completeness

NOT ALL CHARACTERISTICS ARE OF EQUAL WEIGHT!

- Especially not across domains
- **High-frequency trading bot?** Performance might be more important than correctness!
- **Government contracting?** Traceability might be more important than performance!
- **Prototype for a startup cat rental company?** Completeness might be more important than traceability!
- **An extremely large-scale system?** Maintainability might be more important than traceability!
- **Medical device?** Correctness might be more than maintainbility!

SOFTWARE ENGINEERING = TRADE-OFFS

- We will have an entire lecture on trade-offs later on in the term
- Trade-offs happen based on what quality attributes are most important for the system
- Examples:
  - Usability is often at the cost of performance/security
  - Size of code is often a trade-off with speed of execution/ compilation
  - Storing data on disk - trade-off between speed and data storage
  - Optimizing compilers - faster code, more chance of error

PROGRAMMING STYLE

- Often, languages have conventions which are universal or close
  - Class names are in PascalCase
  - Variable names and methods are in camelCase
  - Constants (final variables) are in ALL_CAPS_SNAKE_CASE
  - This looks weird but is valid Java:

```
public class MONKEY_SIM {
    public void EatBanana(int NumBananas) {
        BANANA_LAND.eat(NumBananas * bananaMult);
    }
}
```

SPECIFICS FOR INDIVIDUAL COMPANIES/TEAMS

- No recursion - avoids performance issues and analysis difficulty
- No prefix or postfix incrementation/decrementation
- No modification of loop variables inside of a loop
- All switch statements must have a default
- No fall-through statements in cases (case without a break)
- No // comments on same line as code
- All variables except index variables must be 2 or more chars
- All overridden methods should include @Override annotation

MORE GUIDELINES

- Maximum number of characters on a line (usually 80 or 100)
- Indentation - spaces or tabs? Size?
- Maximum number of lines in a method
- Should we use particular keywords/constructs/features? (e.g. avoiding reflection or sun.misc.Unsafe)
- What kind of documentation of methods should we use?
- What kind of commenting should be used? How much?

MOST IMPORTANT THING - BE CONSISTENT

- Be sure to follow rules of team
- If team does not have explicit rule, try to use the conventions of the language

COMMENTS

- Very important in a language like Java or C++
- Less important in a language like Ruby or Python
  - At TTM, we were taught never to use comments in production code unless absolutely necessary
  - They were a code smell that our code wasn't clear enough!

CLASSES OF COMMENTS

- Repeat of code
- Explanation of code
- Marker in code
- Summary of code
- Description of intent
- External Reference

REPEAT OF CODE

- THIS IS HORRIBLE PLEASE DON'T DO THIS
- Why? Waste of time, distracting

```
// Add 1 and 3 and put the result in variable b
int b = 1 + 3;
```

EXPLANATION OF CODE

- This is good! Especially when code is very complex

```
// The following code goes through each element
// of the MeowList and performs Satler-Bulganov
// normalization of its Quacks, finally summing
// them up for Essential Complexity.

int s = 0;
for (Meow m : meowList) {
   int satlerBulganov = m.quack * 300 - Math.PI;
   satlerBulganov += Math.acos(17);
   s += satlerBulganov;
}
return s;
```

MARKER IN CODE

- This is used often: TODO, CLEANUP, HOTSPOT, etc.

```
// TODO - clean up this method. Better var names?
public int messyMethod(int a) {
    int b = a - 2;
    int c = (b * a) + 1;
    c++;
    int d = --c;
    int e = ++d + c++;
    return e;
}
```

SUMMARY OF CODE

- This is good! Give a high-level overview of what the code is doing. Note that this is different than explaining the code, which describes something in detail. JavaDoc is a good example.

```
/**
 * Given a set of Brumbles, returns the sum of
 * their chumbles.
 * @param ArrayList<Brumble> list of Brumbles
 * @return int - Sum of all Brumble chumbles
 */
public int brumbleChumbleSum(ArrayList<Brumble> x) {
    int sum = 0;
    for (Brumble b: x) {
        sum += b.chumble;
    }
}
```

EXTERNAL REFERENCE

- Also good! A link to where more information can be found.

```
// This method uses a ConcurrentHashMap as multiple
// threads interact with it.  See:
// https://stackoverflow.com/questions/510632/whats-
the-difference-between-concurrenthashmap-and-
collections-synchronizedmap
// for details
```

DEBUGGING

- Something you've all done!
- Locating and fixing errors in code
- Debugging is scientific - develop a hypothesis, then test it!
- Be sure to test the null hypothesis as well!
- Example: 2, 4… ? What is the next number?

PHASES OF DEBUGGING

- **Reproduction (stabilization)** - Find and reproduce the error
- **Localization** - Determine which part of the code caused it
- **Correction** - Modify code to fix error (without breaking anything else!)
- **Verification** - Ensure that your code modifications did in fact fix it and did not break anything else

TOOLS YOU CAN USE

- diff
- Interactive debuggers
- Profilers
- System.out.println(“x = “ + x); // print debugging

DEFENSIVE PROGRAMMING

- Like defensive driving - assume everything calling your method/class/object is reckless and assume your code is full of errors
- Check for preconditions and postconditions

```
public int meow(Cat c, int numTimes){
   if (c == null || c.invalid()) {
     throw new BadCatException();
   }
if (numTimes < 0) {
  throw new InvalidTimeException();
}
try {
   for (int j = 0; j < numTimes; j++) {
      c.meow();
   }
} catch (MeowException mex) {
. . .
```

PERFORMANCE OPTIMIZATION

- Rule 1: Don’t do it.
- Rule 2 (for experts only): Don’t do it yet.

REFACTORING

- *Modifying code without modifying behavior. Examples:*
- Rewriting code so duplicate code is removed
- Splitting up a method from one into multiple methods
- Splitting up a single large class into multiple ones
- Replacing switch statements with polymorphism
- Changing algorithms
- Move methods to different class
- Removing magic numbers (use constants instead of values in code)

MANDEL'S GOLDEN RULES FOR USER INTERFACES

- *Place the user in control*
- *Reduce the user's memory load*
- *Design consistent user interface*

- *More info: http://theomandel.com/wp-content/uploads/2012/07/Mandel-GoldenRules.pdf*

PLACE THE USER IN CONTROL

- Accomodate users with different skill levels and needs (train vs car)
- Allow interfaces via multiple methods (e.g. GUI, command line, mouse, API, etc.)
- Let users customize the interface
- Provide meaningful help, entry, exit, etc. messages and signposts
- Be forgiving! Allow undo and provide immediate feedback

REDUCE THE USER'S MEMORY LOAD

- Be able to save specific settings
- Remind users of data - don't force them to memorize
- Show users data and provide visual cues
- Organize data (important data = larger, in different colors, etc.)
- Provide defaults

DESIGN CONSISTENT USER INTERFACE

- Don’t have Control-C mean “cancel” in one part of your system and “copy” in another
- Similar error messages should look similar and have similar wording
- Use standard terminology across the system
- Try to use industry standards (e.g. in Emacs, copy is Meta-W, cut is Control-W and paste is Control-Y… one reason many people find it hard to use!)