1 Specification for methods and classes

- All methods should have specifications – such specifications should describe the method’s behavior thoroughly.
- Should not constrain the implementation in any way
- Should be the most minimal description that guarantees a certain result for all possible inputs
- The description should be general enough that unimplemented behaviors (and inputs) may be handled in the future without changing the method description
- Should describe whether the input is modified and whether the object called on (this) is modified
- Do the methods allow the object to be changed (mutable) or is the object unchangeable (immutable, like String)

Example

/* Finds first instance in input char[] by creating a new String */
static int findFirstInstanceOfString(char[] c1, char c2) { ... }

/* Returns first index of c within buf, or -1 if c is not in buf */
static int indexOf(char[] buf, char c) { ... }

2 Data abstraction

- How does the object appear to the rest of the world?
- Most important principle is data hiding – only make visible what you must absolutely make visible
- Some people go as far as saying that all fields in all classes should be private with access and set methods to allow manipulation (so, for a variable private int foo, there would be a method getFoo and a method setFoo)
- May want to still provide access and set methods, but aggregate them
- A “representation invariant” can help you test and debug your objects – a method that checks to see whether guarantees about your object hold true (for example, a class that represents fractions could require that its denominator is not 0)

Example

/* Represents a fraction with a numerator and denominator */
class Fraction {
    public int numer, denom; /* everyone can change these */
}
/* Represents a fraction with integer numerator
* and positive integer denominator */
class Fraction {
    private int numer, denom;

    /* Checks the representation invariant */
    boolean repOk() {
        return (denom > 0);
    }

    /* Sets the numerator */
    public void setNumer(int numer) {
        this.numer = numer;
    }

    /* Sets the denominator, if the desired value is positive */
    public void setDenom(int denom) {
        if (denom > 0) {
            this.denom = denom;
        }
    }

    /* get methods not shown */
}

Example

/* Represents a point pair in 3D space */
class PointPair {
    private double x1, y1, z1, x2, y2, z2;

    /* would be tedious to have get and set for each variable */

    /* Sets the starting point of this point pair to [x1, y1, z1] */
    public setStartPoint(double x1, double y1, double z1) {
        this.x1 = x1;
        this.y1 = y1;
        this.z1 = z1;
    }

    /* other methods not shown */
}

Example

/* A Min Binary Heap of n elements */
public class MinHeap {
    /* stores the heap */
    private Vector h;

    /* Bad comment: */
    /* Grabs h[0], moves h[size-1] to h[0] and reheapifies */
public Node extractMin() {
    ...
}

/* Better comment: */
/* Removes and returns the minimum Node stored by this heap */
public Node extractMin() {
    ...
}

3 Design

- Start with high level idea – what are we trying to solve?
- Approach the design top-down – only break up the problem into big pieces (of which you know the slightly smaller components), and write them
- The code the high level pieces should be very abstract and vague – make method calls to methods you haven’t written yet, but have specified the behavior of
- Then, within those methods, break them into components
- Group appropriate methods that share information into classes
- Group methods that all function the same into the same general methods (and use overloading to indicate that multiple methods take different arguments but function the same)
- Be lazy on purpose – postpone writing any code that does anything of substance for a while – simply write methods with empty method bodies unless the methods will be less than 5 lines of code (in which case, you may as well write them)
- Sometimes it turns out to be useful to not override methods from parent classes, but simply use them directly (TrReader)
- Write the code with the assumption that every program you use for the next 5 years will use the code, so by making it general (and well-defined)
- Follow these principles and your programs will seem to write themselves

Example

/* snippet of code from CtfServer */
while (!gameOver) {
    while (!(roundOver || gameOver)) {

        processAllPlayerInput();

        // calculate new positions
        if (!paused) {
            updatePlayArea();
            broadcastGameInfo();
            broadcastTick();
        }
    }
}
// wait
long sleeptime = nextTick.getTime() - System.currentTimeMillis();
if (sleeptime > 0) {
    try {
        Thread.sleep(sleeptime);
    } catch (InterruptedException e) {
        if (CtfConstants.DEBUG)
            e.printStackTrace();
    }
}

// set new tick times
thisTick.setTime(System.currentTimeMillis());
nextTick.setTime(thisTick.getTime()+tickLength);
if (roundOver) {
    endOfRound();
}
}
sendToAllPlayers("gameover");
System.exit(0);

4 Style

• Not required, but helps you (and others) read your code

• When if statements are used, and there are other conditional blocks before and/or afterwards, use
  braces even if the body of the if is just one statement (same holds for else while do for)

• Write comments that actually describe something useful, or don’t write the comment (if your code is
  self documenting, then you won’t have a problem, and many comments become unnecessary)

• Don’t use “magic numbers” – define all constant values at the top of your program as public static final
  variables in CAPS and then use them within your program

• Use the for and while loop idioms for iteration on arrays and lists to avoid off-by-one errors and
  improve clarity

• Don’t use if statements to return booleans – just return the boolean value itself

• Use short variable names for counters (like i j k …) and descriptive (but not too long) variable
  names for other things

Example

/* what? */
/* returns whether x & x-1 is 0 */
public boolean foo(int x) {
    return ((x & (x-1)) == 0);
}

/* aha */
/* returns whether x is a power of 2 */
public boolean foo(int x) {
    return ((x & (x-1)) == 0);
}

Example

/* useless comments */

/* bar is a method that takes in an integer value x and an integer
* value y and add them, computing their sum, and returns the result of
* the addition */
public void bar(int x, int y) {
    return x + y; // adds x and y and returns the resulting sum
}

Example

private byte[] lengthInBytes(String s) {
    int length = s.length();
    byte h[] = new byte[37]; /* what? */
    ... 
}

public static final int LENGTH_OF_HEADER = 37;

private byte[] lengthInBytes(String s) {
    int length = s.length();
    byte h[] = new byte[LENGTH_OF_HEADER]; /* aha */
    ... 
}

Example

/* for loops */
int [] x = new int[SIZE];

for (int i = 0; i < x.length; i++) {
    ... 
}

/* while loops */
List y = ...;

while (y != null) {
    ... 
    y = y.next;
}

5 Testing

- Glass-box testing vs. Black-box testing
In black-box testing, simply pretend as if the representation invariant of a class holds true and that the class will behave as describe only using its public methods – execute those methods with expected input and see that the output matches the specification.

In glass-box testing, explicitly test the internal workings of the program – try to write code to travel along all paths of the program (for each if statement, try to provide input that tests both possible condition values, testing both branches).

Code coverage in testing is often important – tests that ensure that every line (or almost every line) of code was used to do something in some test case.

Regression testing uses a set of test cases that are re-run after some time to ensure that old test cases are not broken in a new version of the program.

6 Debugging

In Java, the debugger is sometimes useful (especially with threads), but not always useful.

Write `toString()` and `equals()` methods for all classes to make output and comparison convenient.

Print out where you are in the program at the beginning of your debugging messages.

Use a Debug method or DEBUG constant to determine whether or not to output debugging information (and put it somewhere reachable by other parts of your code).

Example

```java
class A {
    public void foo() {
        util.Debug("[A] in foo()");
    }
}

class B {
    public void foo() {
        util.Debug("[B] in foo()");
    }
}

class util {
    public static final boolean DEBUG = true;

    public static void Debug(String s) {
        if (DEBUG) {
            System.err.println(s);
        }
    }
}
```

7 Performance

- Method calls are slower than field accesses.
- Object creation is very slow.
• With that in mind, try to reduce the number of objects you create during runtime
• Reduce the number of temporary variables you use
• Don’t use Strings! Concatenation of strings is very expensive because the strings must be recopied and have a new string created to hold the result.
• Use integer primitives instead of objects when possible
• Store things of known fixed sizes in arrays instead of Vectors
• Use declared constants as flags instead of Strings
• Declare classes that will not be overridden as final to increase the speed of method calls
• Remember that asymptotic improvements will make a much greater difference in the performance of your program
• Use the optimizer – javac -O ...