THOUGHTS ON A4

Lecture 19
CS2110 – Fall 2014

A4 and A5


A5. We have begun to populate the FAQ note for A5 on the CMS. Will put more info there later today, and after that, whenever necessary. Look at it often.

Get started on A5 soon! Get it done EARLY.

Before posting a question about A5 on the Piazza:
- See whether it is already answered on the Piazza!
- Look in the Java API specs for the answer
- Google your question

About Testing

Example conversation after finding a bug, a typo (an x for a y), in someone’s BoundingBox.getCenter. After it was fixed, everything worked.

Did you test all the BoundingBox methods before moving on to BlockTree? No.

How long do you think it takes to write a JUnit testing class to test all the methods in BoundingBox? I don’t know.

When I, Gries, developed the solution, it took me 20 minutes to build such a BoundingBox tester, and I found 2 errors/typos.

How many hours did you and consultants spend looking for the error when the GUI didn’t work, looking mainly at BlockTree.contains, BlockTree.overlap? Probably 6-7 hours, more

Keep things simple

/** Return true iff this bounding box overlaps with box */ public boolean overlaps(BoundingBox box) {
    double x0 = 0;
    double x1 = this.getWidth();
    double y0 = 0;
    double y1 = this.getHeight();
    if (box.contains(x0, y0) || box.contains(x1, y0) || box.contains(x0, y1) || box.contains(x1, y1)) {
        return true;
    } else {
        return false;
    }
}

About Testing

Fact: We all make mistakes, both simple typos and logical mistakes.

Fact: We all have the urge to move on and write more code, without proper testing of what is already written.

SO Continually discipline yourself to write and test code incrementally. Make sure that basic methods are correct before moving on to write code that calls those methods.

Keep things simple and beautiful

There are two ways of constructing a piece of software: One is to make it so simple that there are obviously no errors. The other is to make it so complicated that there are no obvious errors. — Tony Hoare

Inside every large program, there is a small program trying to get out. — Tony Hoare

When I’m working on a problem, I never think about beauty. I think only how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong. — R. Buckminster Fuller
Keep things simple and beautiful

When your work starts to get long and complicated, STOP, reflect, look for different approaches.

Simplify your work by avoiding useless clutter

When do two rectangles overlap?

Rectangles and do not overlap if one is to the right of the other or one is below the other. Otherwise, they overlap.

```java
/** Return true if this box overlaps with box. */
public boolean overlaps(BoundingBox box) {
    if (upper.x < box.lower.x) return false;
    if (box.upper.x < lower.x) return false;
    if (upper.y < box.lower.y) return false;
    if (box.upper.y < lower.y) return false;
    return true;
}
```

BlockTree.overlaps: too much case analysis

```java
public boolean overlaps(BoundingBox thisD, BlockTree t, Vector2D d) {
    if (upper.x < box.lower.x) return false;
    if (box.upper.x < lower.x) return false;
    if (upper.y < box.lower.y) return false;
    if (box.upper.y < lower.y) return false;
    return true;
}
```

BlockTree.overlaps: too much case analysis

```java
public boolean overlaps(Vector2D thisD, BlockTree t, Vector2D d) {
    if ( block == null) {
        if (t.block != null) {
            return Block.overlaps(block, thisD, t.block, d);
        } else {
            return overlaps(thisD, t.left, d) || overlaps(thisD, t.right, d);
        }
    } else {
        if (!box.displaced(thisD).overlaps(t.box.displaced(d)))
            return false;
        else return true;
    }
}
```

What else is wrong with this?

Unnecessary clutter: “this.”

No spaces around operators, after if, before {.

BlockTree.overlaps: too much case analysis

```java
public boolean overlaps(BoundingBox thisD, BlockTree t, Vector2D d) {
    return true;
}
```

BlockTree.overlaps: too much case analysis

```java
public boolean overlaps(Vector2D thisD, BlockTree t, Vector2D d) {
    if (this==null || t==null) return false;
    if (!this.box.overlaps(thisD, t.box, d)) return false;
    if (isLeaf() && t.isLeaf()) return true;
    if (isLeaf()) return t.overlaps(d, this, thisD);
    if (t.isLeaf()) {
        if (left.box.getArea() > right.box.getArea()) {
            return left.overlaps(thisD, t, d) || right.overlaps(thisD, t, d);
        } else {
            return right.overlaps(thisD, t, d) || left.overlaps(thisD, t, d);
        }
    } else {
        if (box.displaced(thisD).overlaps(t.box.displaced(d)))
            return false;
        else return true;
    }
}
```

A beautiful overlaps

```java
public boolean overlaps(Vector2D thisD, BlockTree t, Vector2D d) {
    // If the blocks don’t overlap, return false.
    if (!box.displaced(thisD).overlaps(t.box.displaced(d)))
        return false;
    // If the blocks overlap
    if (isLeaf() && t.isLeaf()) return true;
    // Recurse on the longer of this and t
    if (box.getLength() > t.box.getLength())
        return left.overlaps(thisD, t, d) || right.overlaps(thisD, t, d);
    else
        return right.overlaps(thisD, t, d) || left.overlaps(thisD, t, d);
}
```
A beautiful overlaps

Suppose \( t \) contains 2 blocks and depth of this tree is \( d \).

\[
\text{Worst case: total of } d \text{ recursive calls}
\]

Why is recursing on longer better? 

We provide intuition

public boolean overlaps(Vector2D thisD, BlockTree t, Vector2D d) {
    if (!box.displaced(thisD).overlaps(t.box.displaced(d)))
        return false;
    if (isLeaf() && t.isLeaf()) return true;
    if (box.getLength() > t.box.getLength())
        return left.overlaps(thisD, t, d) || right.overlaps(thisD, t, d);
    else
        return t.left.overlaps(d, this, thisD) || t.right.overlaps(d, this, thisD);
}

Recurse on shorter? Need more case analysis

public boolean overlaps(Vector2D thisD, BlockTree t, Vector2D d) {
    if (!box.displaced(thisD).overlaps(t.box.displaced(d)))
        return false;
    if (isLeaf() && t.isLeaf()) return true;
    if (one of the trees is a leaf) take care of this case
        // Takes up to \( d \) recursive calls
    if (box.getLength() < t.box.getLength())
        return left.overlaps(thisD, t, d) || right.overlaps(thisD, t, d);
    else
        return t.left.overlaps(d, this, thisD) || t.right.overlaps(d, this, thisD);
    // Suppose \( t \) contains 2 blocks and depth of this tree is \( d \).
    // \( 2d \) recursive calls: \( d \) for left.overlap and \( d \) for right.overlap.
}

Summary

1. Code and test incrementally. Don’t write a call on a method unless that method has been checked thoroughly.
2. Use already written methods –don’t reinvent the wheel.
3. Strive for clarity, simplicity, brevity.
4. Avoid unnecessary clutter and case analysis.
5. Use returns in functions to avoid case analysis. See Code Style Guidelines.
6. Don’t accept your first “correct” method as the final one. Like an essay in English, it may need reorganizing, rethinking, reworking.