Lecture 7: Testing

9/11/2020

A New Way

How Does a Programmer Know That Their Code Works?

- Running main and seeing if the code behaves as expected
- The Autograder
- In the real world, programmers believe their code works because of **tests they write themselves**
    - Knowing that it works for sure is usually impossible
    - This will be our new way

Sorting: The McGuffin for Our Testing Adventure

- To try out this new way, we need a task to complete
    - Let's try to write a method that sorts arrays of Strings

The New Way

- We will write our test for TestSort first

Ad Hoc Testing vs. JUnit

Ad Hoc Test

```
public class TestSort {
    /** Test the Sort.sort method */
    public static void testSort() {
        String[] input = {"i", "have", "an", "egg"};
        String[] expected = {"an", "egg", "have", "i"};

        Sort.sort(input);

        for (int i = 0; i < input.length; i += 1) {
            if (!input[i].equals(expected[i])) {
                System.out.println("Mismatch in position " + i);
                return;
            }
        }

        if (java.utils.Arrays.equals(input, expected)) {
            System.out.println("Error! There seems to be a problem with
Sort.sort.");
        }
    }

    public static void main(String[] args) {
```

```
        testSort();
    }
}
```

JUnit: A Library for Making Testing Easier

- Ad Hoc Testing is tedious, use JUnit library instead

```
public class TestSort {
    /** Test the Sort.sort method */
    public static void testSort() {
        String[] input = {"i", "have", "an", "egg"};
        String[] expected = {"an", "egg", "have", "i"};

        Sort.sort(input);

        org.junit.Assert.assertArrayEquals(expected, input);
    }

    public static void main(String[] args) {
        testSort();
    }
}
```

Selection Sort

Back to Sorting: Selection Sort

- Selection sorting a list of N items:
  - Find the smallest item
  - Move (or swap it) it to the front
  - Selection sort the remaining N-1 items (without touching front item!)
- Let's try implementing this
  - I'll try to simulate as closely as possible how I think students might approach this problem to show how TDD helps

```
public class Sort {
    /** Sorts strings recursively */
    public static void sort(String[] x) {
        // Find the smallest item
        // Move it to the front
        // Selection sort the rest
        int smallestIndex = findSmallest(x);
        swap(x, 0, smallestIndex);
    }

    /** Swap item a with b */
    public static void swap(String[] x, int a, int b) {
```

```
        String temp = x[a];
        x[a] = x[b];
        x[b] = temp;
    }

    /** Returns the index of the smallest String in x */
    public static int findSmallest(String[] x) {
        int smallestIndex = 0;
        for (int i = 0; i < x.length; i += 1) {
            int cmp = x[i].compareTo(x[smallestIndex]);
            if (cmp < 0) {
                smallestIndex = i;
            }
            return smallestIndex;
        }
    }
}

public class TestSort {
    /** Test the Sort.sort method */
    public static void testSort() {
        String[] input = {"i", "have", "an", "egg"};
        String[] expected = {"an", "egg", "have", "i"};

        Sort.sort(input);

        org.junit.Assert.assertArrayEquals(expected, input);
    }

    /** Test the Sort.findSmallest method */
    public static void testFindSmallest() {
        String[] input = {"i", "have", "an", "egg"};
        int expected = 2;

        int actual = Sort.findSmallest(input);
        org.junit.Assert.assertEquals(expected, actual);

        String[] input2 = {"there", "are", "many", "pigs"};
        int expected2 = 1;

        int actual = Sort.findSmallest(input2);
        org.junit.Assert.assertEquals(expected2, actual2);
    }

    public static void testSwap() {
        String[] input = {"i", "have", "an", "egg"};
        int a = 0;
        int b = 2;
        String[] expected = {"an", "have", "i", "egg"};

        Sort.swap(input, input, b);
        org.junit.Assert.assertArrayEquals(expected, input);
    }
```

```
    public static void main(String[] args) {
        testSwap();
        testFindSmallest();
        testSort();
    }
}
```

The Evolution of Our Design

- Created testSort
- Created a sort skeleton
- Created testFindSmallest
- Created findSmallest
- Created testSwap
- Created swap
- Changed findSmallest
- Now we have all the helper methods we need, as well as tests that make us pretty sure that they work
    - All that's left is to write the sort method itself.

```
public class Sort {
    /** Sorts strings recursively */
    public static void sort(String[] x) {
        sort(x, 0);
    }

    /** Sorts x starting at position start */
    private static void sort(String[] x, int start) {
        if (start == x.length) {
            return;
        }
        int smallestIndex = findSmallest(x, start);
        swap(x, start, smallestIndex);
        sort(x, start + 1);
    }

    /** Swap item a with b */
    public static void swap(String[] x, int a, int b) {
        String temp = x[a];
        x[a] = x[b];
        x[b] = temp;
    }

    /** Returns the index of the smallest String in x. Starting at start
*/
    public static int findSmallest(String[] x, int start) {
        int smallestIndex = start;
        for (int i = start; i < x.length; i += 1) {
            int cmp = x[i].compareTo(x[smallestIndex]);
            if (cmp < 0) {
                smallestIndex = i;
```

```
            }
            return smallestIndex;
        }
    }
}

public class TestSort {
    /** Test the Sort.sort method */
    public static void testSort() {
        String[] input = {"i", "have", "an", "egg"};
        String[] expected = {"an", "egg", "have", "i"};

        Sort.sort(input);

        org.junit.Assert.assertArrayEquals(expected, input);
    }

    /** Test the Sort.findSmallest method */
    public static void testFindSmallest() {
        String[] input = {"i", "have", "an", "egg"};
        int expected = 2;

        int actual = Sort.findSmallest(input, 0);
        org.junit.Assert.assertEquals(expected, actual);

        String[] input2 = {"there", "are", "many", "pigs"};
        int expected2 = 2;

        int actual2 = Sort.findSmallest(input2, 2);
        org.junit.Assert.assertEquals(expected2, actual2);
    }

    /** Test the Sort.swap method */
    public static void testSwap() {
        String[] input = {"i", "have", "an", "egg"};
        int a = 0;
        int b = 2;
        String[] expected = {"an", "have", "i", "egg"};

        Sort.swap(input, input, b);
        org.junit.Assert.assertArrayEquals(expected, input);
    }

    public static void main(String[] args) {
        testSwap();
        testFindSmallest();
        testSort();
    }
}
```

The Evolution of Our Design

- Created testSort
- Created a sort skeleton
- Created testFindSmallest
- Created findSmallest
- Created testSwap
- Created swap
- Changed findSmallest
- Now we have all the helper methods we need, as well as tests that make us pretty sure that they work
    - All that's left is to write the sort method itself.
- Modified findSmallest

Reflections on the Process

And We're Done!

- Often, development is an incremental process that involves lots of task switching and on the fly design modification
- Tests provide stability and scaffolding
    - Provide confidence in basic units and mitigate possibility of breaking them
    - Help you focus on one task at a time
- In larger projects, tests also allow you to safely **refactor**! Sometimes code gets ugly, necessitating redesign and rewrites
- One remaining problem: Sure was annoying to have to constantly edit which tests were running. Let's take care of that

Simpler JUnit Tests

Simple JUnit

- New Syntax #1: `org.junit.Assert.assertEquals(expected, actual);`
    - Tests that expected equals actual
    - If not, program terminates with verbose message
- JUnit does much more
    - `assertEquals`, `assertFalse`, `assertNotNull`, etc.
    - Other more complex behavior to support more sophisticated testing

Better JUnit

- The messages output by JUnit are ugly
- New Syntax #2
    - Annotate each test with `@org.junit.Test`
    - Change all test methods to non-static
    - Use a JUnit runner to run all tests and tabulate results
        - Intellij provides a default runner/renderer. OK to delete `main`
        - Rendered output is easier to read, no need to manually invoke tests

Even Better JUnit

- It is annoying to type out the name of the library repeatedly

- New Syntax #3: To avoid this we'll start every test file with:
  - `import org.junit.Test;`
  - `import static org.junit.Assert.*`
- This will eliminate the need to type `org.junit` or `org.junit.Assert`

```
public class Sort {
    /** Sorts strings recursively */
    public static void sort(String[] x) {
        sort(x, 0);
    }

    /** Sorts x starting at position start */
    private static void sort(String[] x, int start) {
        if (start == x.length) {
            return;
        }
        int smallestIndex = findSmallest(x, start);
        swap(x, start, smallestIndex);
        sort(x, start + 1);
    }

    /** Swap item a with b */
    public static void swap(String[] x, int a, int b) {
        String temp = x[a];
        x[a] = x[b];
        x[b] = temp;
    }

    /** Returns the index of the smallest String in x. Starting at start
*/
    public static int findSmallest(String[] x, int start) {
        int smallestIndex = start;
        for (int i = start; i < x.length; i += 1) {
            int cmp = x[i].compareTo(x[smallestIndex]);
            if (cmp < 0) {
                smallestIndex = i;
            }
            return smallestIndex;
        }
    }
}

import org.junit.Test;
import static org.junit.Assert.*;

public class TestSort {
    /** Test the Sort.sort method */
    @org.junit.Test
    public void testSort() {
        String[] input = {"i", "have", "an", "egg"};
        String[] expected = {"an", "egg", "have", "i"};
```

```
        Sort.sort(input);

        assertArrayEquals(expected, input);
    }

    /** Test the Sort.findSmallest method */
    @org.junit.Test
    public void testFindSmallest() {
        String[] input = {"i", "have", "an", "egg"};
        int expected = 2;

        int actual = Sort.findSmallest(input, 0);
        assertEquals(expected, actual);

        String[] input2 = {"there", "are", "many", "pigs"};
        int expected2 = 2;

        int actual2 = Sort.findSmallest(input2, 2);
        assertEquals(expected2, actual2);
    }

    /** Test the Sort.swap method */
    @org.junit.Test
    public void testSwap() {
        String[] input = {"i", "have", "an", "egg"};
        int a = 0;
        int b = 2;
        String[] expected = {"an", "have", "i", "egg"};

        Sort.swap(input, input, b);
        assertArrayEquals(expected, input);
    }
}
```

Testing Philosophy

Correctness Tool #1: Autograder

- Idea: Magic autograder tells you code works
  - We use JUnit + jh61b libraries
- Why?
  - Less time wasted on "boring" stuff
  - Determines your grade
  - Gamifies correctness
- Why not?
  - Autograders don't exist in real world
  - Errors may be hard to understand
  - Slow workflow
  - No control if grader breaks/misbehaves

Autograder Driven Development (ADD)

- The worst way to approach programming:
    - Read and (mostly) understand the spec
    - Write entire program
    - Compile. Fix all compilation errors
    - Send to autograder. Get many errors
    - Until correct, repeat randomly
        - Run autograder
        - Add print statements to zero in on the bug
        - Make changes to code to try to fix bug
- This workflow is slow and unsafe!

Correctness Tool #2: Unit Tests

- Idea: Write tests for every "unit"
    - JUnit makes this easy!
- Why?
    - Build confidence in basic modules
    - Decrease debugging time
    - Clarify the task
- Why not?
    - Building tests take time
    - May provide false confidence
    - Hard to test untis that rely on others
        - e.g. how do you test `addFirst`?

Test-Driven Development (TDD)

- Steps to developing according to TDD:
    - Identify a new feature
    - Write a unit test for that feature
    - Run the test. It should fail
    - Write code that passes test
        - Implementation is certifiably good
    - Optional: Refactor code to make it faster, cleaner, etc.

A Tale of Two Workflows

- TDD is an extreme departure from the naive workflow
    - What's best for you is probably in the middle

Correctness Tool #3: Integration Testing

- Idea: Tests cover many units at once
    - Not JUnit's focus, but JUnit can do this
- Why?
    - Unit testing is not enough to ensure modules interact properly or that system works as expected
- Why not?
    - Can be tedious to do manually

    - Can be challenging to automate
    - Testing at highest level of abstraction may miss subtle or rare errors

Parting Thoughts

- JUnit makes testing easy
- You should write tests
    - But not too many
    - Only when they might be useful!
    - Write tests first when it feels appropriate
    - Most of the class won't require writing lots of tests (to save you time)
- Some people really like TDD. Feel free to use it in 61B.