8.3 -- Testing the controller in isolation

kiss-play-video.png Video Lecture (https://northeastern.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=7ad91eb3-65b1-4830-8b04-ab64001350a0)

All the tests in previous modules test two things: whether the model works correctly and whether the controller receives inputs and transmits the result produced by the model. A correct actual output transmitted by the controller reflects that *both* of them worked correctly. This is redundant, because we would have tests to separately verify the model's correctness. The following (strategically buggy) implementation of a controller will still pass these tests:

```
public class BuggyController {
  private final Readable in;
  private final Appendable out;

  public BuggyController(Readable in, Appendable out) {
    this.in = in;
    this.out = out;
  }

  /**
   * Method that gives control to the controller.
   *
   * @param model the model to use.
   * @throws IOException if something goes wrong appending to out
   */
  public void start(Calculator model) throws IOException {
    Objects.requireNonNull(model);
    int num1;
    int num2;
    Scanner scan = new Scanner(this.in);
    while (true) {
      String command = scan.next();
      switch (command) {
      case "+":
        num1 = scan.nextInt() + 10; // wrong input, bug
        num2 = scan.nextInt() - 10; // wrong input, but this bug nullifies above bug
        this.out.append(String.format("%d\n", model.add(num1, num2)));
        break;
      case "q":
        scan.close();
        return;
      default:
        throw new UnsupportedOperationException(command + " not supported");
      }
    }
  }
}
```

This bug appears contrived, but more such realistic cases will appear as the model becomes more complicated. As our tests pass, we will *incorrectly* conclude that our controller works correctly.

A better way to test the controller would be by isolating it. How do we know that the controller, *in isolation*, works correctly? It works correctly if it reads the inputs in the correct sequence, sends them to the model correctly, and transmits any outputs to the view correctly. This would mean our controller-model would work correctly, assuming that we have independently verified that the model works correctly when presented correct inputs.

In order to isolate the controller, we provide it with a "mock" model. A "mock" of an object is another object that "looks like the real object, but is simpler". In particular, the simplicity here implies that it would allow us to verify what was passed to it correctly. We begin by creating an explicit interface for the model for the above example (we should have one for the model in general). We then make our model object implement this interface.

```
public interface ICalculator {
  int add(int one, int two);
}
```

```
public class Calculator implements ICalculator {

  @Override
  public int add(int one, int two) {
    return one + two;
  }
}
```

We now change our controller and client code slightly.

```
public class BuggyController {
  private final Readable in;
  private final Appendable out;

  public BuggyController(Readable in, Appendable out) {
    this.in = in;
    this.out = out;
  }

  /**
   * Method that gives control to the controller.
   *
   * @param model the model to use.
   * @throws IOException if something goes wrong appending to out
   */
  public void start(ICalculator model) throws IOException {
    Objects.requireNonNull(model);
    int num1;
    int num2;
    Scanner scan = new Scanner(this.in);
    while (true) {
      String command = scan.next();
      switch (command) {
      case "+":
        num1 = scan.nextInt() + 10; // wrong input, bug
        num2 = scan.nextInt() - 10; // wrong input, but this bug nullifies above bug
        this.out.append(String.format("%d\n", model.add(num1, num2)));
        break;
      case "q":
        scan.close();
        return;
      default:
        throw new UnsupportedOperationException(command + " not supported");
      }
    }
  }
}
```

```
public static void main(String[] args) {
  try {
    // create the model
    ICalculator model = new Calculator();

    // create the controller
    Readable reader = new InputStreamReader(System.in);
    BuggyController control = new BuggyController(reader, System.out);

    // give the model to the controller and give it control
    control.start(model);
  } catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
  }
}
```

Now we create a "mock" model class.

```
public class MockModel implements ICalculator {
  private StringBuilder log;
  private final int uniqueCode;

  public MockModel(StringBuilder log, int uniqueCode) {
    this.log = log;
    this.uniqueCode = uniqueCode;
  }

  @Override
  public int add(int one, int two) {
    log.append("Input: " + one + " " + two + "\n");
    return uniqueCode;
  }
}
```

This mock does not actually add numbers: it merely logs the inputs provided to it, and returns a unique number provided to it at creation. We can now test the controller in isolation as follows:

```
@Test
public void testStartWithMock() throws IOException {
  StringBuffer out = new StringBuffer();
  Reader in = new StringReader("+ 3 4 + 8 9 q");

  StringBuilder log = new StringBuilder();
  ICalculator model = new MockModel(log, 1234321);
  BuggyController controller = new BuggyController(in, out);
  controller.start(model);
  assertEquals("Input: 3 4\nInput: 8 9\n", log.toString()); // input reaches model correctly
  assertEquals("1234321\n1234321\n", out.toString()); // output from model received correctly
}
```

What does this test do? It tests whether the inputs provided to the controller were correctly transmitted to the model, and the results from the model were correctly transmitted to the `Appendable` object by the controller. It *does not* test whether the controller-model combination produced the correct answer. However this test, along with those for the model, collectively verify the correctness of this combination. It is noteworthy that this test will correctly catch the contrived bug in `BuggyController` above.

As the complexity of the model and controller increases, testing using mock objects provides more substantial benefits. The tests are likely shorter and more focused.