8.2 -- Refactoring a simple example

The following is a simple program:

```
/**
 * Implementation of a simple calculator.
 */
public class SimpleCalculator {
  public static void main(String[] args) {
    int one;
    int two;
    Scanner scan = new Scanner(System.in);
    one = scan.nextInt();
    two = scan.nextInt();
    System.out.printf("%d", one + two);
    scan.close();
  }
}
```

The above is a simple program. It has a monolithic design, with `main` doing all the work. The only thing good about this program is that it works. In fact there is no way to even test this program using automated testing, because the main method contains all the logic.

Factoring out the model

kiss-play-video.png Video Lecture -- Part 1 of 4 (https://northeastern.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=aab4cb44-6590-4dbe-847e-ab6301845760)

We begin improving the design of this program by factoring out a model. Recall that the model is the part of the program that implements all its features. In the above program, the part that adds two numbers can be thought of as the model. We factor it out in its own class, and modify the `main` method to use it.

```
/**
 * Demonstrates a simple command-line based calculator.
 * In this example, the model is factored out.
 */
public class SimpleCalculator {

  public static void main(String[] args) {
    int one;
    int two;
    Scanner scan = new Scanner(System.in);
    one = scan.nextInt();
    two = scan.nextInt();
    Calculator model = new Calculator();
    System.out.printf("%d", model.add(one, two));
    scan.close();
  }
}

@Test
public void testAdd() {
  Calculator model = new Calculator();
  assertEquals(7, model.add(3,  4));
}
```

However other parts of the program still cannot be tested, because they are embedded inside the `main` method.

Factoring out the controller

kiss-play-video.png Video Lecture -- Part 2 of 4 (https://northeastern.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=0bb0f1ab-0158-4b93-901a-ab6301891ada)

In the above iteration, the `main` method still takes input from the user. This is the job of the controller. The following refactoring carves out the controller.

```
/**
 * Demonstrates a simple command-line based calculator.
 * In this example, the model and controller are factored out.
 */
public class SimpleCalculator {

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

    // create the controller
    Controller control = new Controller();

    // give the model to the controller and give it control
    control.start(model);
  }
}

/**
 * A controller for our calculator.  This calculator is still
 * hard-wired to System.in, making it difficult to test through JUnit.
 */
public class Controller {

  public void start(Calculator model) {
    Objects.requireNonNull(model);
    int one;
    int two;
    Scanner scan = new Scanner(System.in);
    one = scan.nextInt();
    two = scan.nextInt();
    System.out.printf("%d", model.add(one, two));
    scan.close();
  }
}
```

The code of the `main` method now looks better: it creates the model and the controller, links the latter to the former and then relinquishes control to the controller (by calling its `start` method). Since we divide all tasks of a program amongst model, views and controllers, the `main` method should do nothing more than set up the main components and pass control to the controller.

However this controller is not testable, because it uses a specific way of accepting inputs (`System.in`) and a specific way to transmit output (`System.out`). Also, showing outputs is the responsibility of the view, so this controller is really a controller mixed with a specific, static view.

Abstracting the controller

kiss-play-video.png Video Lecture -- Part 3 of 4 (https://northeastern.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=0795b203-e469-43d6-b1c5-ab6400036b34)

Ideally the controller should be independent of the specific source of the input, and should be able to work with a general-purpose view. We change the controller design by replacing `System.in` with a higher-level, general-purpose input source, and `System.out` with a higher-level, general-purpose output source.

Looking at the **documentation (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/System.html)** for `System.in` and `System.out`, we see that they are instances of `InputStream` and `PrintStream` respectively. These are two types of streams in Java (unrelated to the Stream API for functional programming constructs). A stream is a general-purpose source or sink of data. You may think of a stream as a "general pipe" that can be connected to a wide variety of actual data sources (keyboard, files, input buffers, etc.) and data sinks (console, files, output buffers, etc.).

Therefore we refactor the controller by allowing it to work with *any* `InputStream` and `PrintStream` objects.

Looking at the **documentation (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InputStream.html)** for the `InputStream` class we see that it allows relatively low-level reading (reading a byte, for example). The advantage of the `Scanner` class is that it provides a higher-level abstraction for reading data ("read a number", "read a word", etc.). Fortunately, the `Scanner` object can be coupled with an `InputStream` object, so that we can continue to use a `Scanner` object in the controller.

```
/**
 * A controller for our calculator.  This controller receives all its
 * inputs from an InputStream and transmits all outputs to a PrintStream
 * object.  The PrintStream object would be provided by a view
 * (not shown in this example).  This design allows us to test.
 */
public class Controller {

  private final InputStream in;
  private final OutputStream out;

  public Controller(InputStream in, OutputStream out) {
    this.in = in;
    this.out = out;
  }

  /**
   * Method that gives control to the controller.
   *
   * @param model the model to use.
   */
  public void start(Calculator model) throws IOException {
    Objects.requireNonNull(model);
    int one;
    int two;
    Scanner scan = new Scanner(this.in);
    one = scan.nextInt();
    two = scan.nextInt();
    this.out.printf("%d\n", model.add(one, two));
    scan.close();
  }
}
```

How can we use this from our `main` method?

```
/**
 * Demonstrates a simple command-line based calculator.
 * In this example, the model and controller are factored out.
 */
public class SimpleCalculator {

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

      // create the controller
      Controller control = new Controller(System.in, 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();
    }
  }
}
```

Since the controller can work with *any* `InputStream` and `PrintStream` objects, we can now write tests that provide it with suitable objects. Unlike the `main` method, we need an `InputStream` object that does not wait for the user to enter, but "pull" data from a pre-populated source.

> Do Now!
>
> Look at the documentation for the `InputStream` and `PrintStream` classes, and determine which subclasses can be helpful in the context of a non-interactive test.

We put our data in a simple `String` object, and convert it to an array of bytes. We then use the `ByteArrayInputStream` and connect it to the array of bytes.

```
String input = "3 4";
InputStream in = new ByteArrayInputStream(input.getBytes());
```

Similarly, we would like a `PrintStream` object that allows us to extract its contents, so that we can verify them with the expected output. This is easily done using strings as well. We connect a `PrintStream` object with a `ByteArrayOutputStream` object.

```
ByteArrayOutputStream bytes = new ByteArrayOutputStream();
PrintStream out = new PrintStream(bytes);
...
String output = bytes.toString(); //extract contents
```

We use this to (finally) write an automated test for our controller.

```
@Test
public void testStart() throws Exception {
  InputStream in = new ByteArrayInputStream("3 4".getBytes());
  ByteArrayOutputStream bytes = new ByteArrayOutputStream();
  PrintStream out = new PrintStream(bytes);
  CalcController controller4 = new Controller4(in, out);
  controller4.start(new Calculator());
  assertEquals("7\n", new String(bytes.toByteArray()));
}
```

Simplifying inputs and outputs even more

[kiss-play-video.png Video Lecture -- 4 of 4 (https://northeastern.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=1ba964a3-ed2f-4c87-bb57-ab640007aaf6)](https://northeastern.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=1ba964a3-ed2f-4c87-bb57-ab640007aaf6)

Streams are powerful abstractions, and contain many data-related features (such as the ability to buffer, for efficiency). However our controller does not need any such sophisticated features: it just needs to be able to read data one "token" at a time. Therefore we use an even simple abstraction of inputs: `Readable` objects. The documentation for the `Readable` interface shows that it allows a simpler API: a method that reads one or more characters. The corresponding interface for outputs is the `Appendable` interface.

We factor our controller to use these interfaces:

```
/**
 * A controller for our calculator. This calculator receives all its inputs
 * from a Readable ojbect and transmits all outputs to an Appendable object.
 * The Appendable object would be provided by a view (not shown here).
 * This design allows us to test.
 */
public class Controller {
  private final Readable in;
  private final Appendable out;

  public Controller(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 one;
    int two;
    Scanner scan = new Scanner(this.in);
    while (true) {
      String command = scan.next();
      switch (command) {
      case "+":
        one = scan.nextInt();
        two = scan.nextInt();
        this.out.append(String.format("%d\n", model.add(one, two)));
        break;
      case "q":
        scan.close();
        return;
      default:
        throw new UnsupportedOperationException(command + " is unsupported");
      }
    }
  }
}
```

How do we "hook up" our main method to such a controller? Since we already know that `System.in` is an `InputStream` object, we need something that will convert this into a `Readable` object. We use the `InputStreamReader` class for this purpose. Since a `PrintStream` is an `Appendable` we can pass `System.out` directly to this controller.

```
public class SimpleCalculator {

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

      // create the controller
      Readable reader = new InputStreamReader(System.in);
      Controller control = new Controller(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();
    }
  }
}
```

The `Appendable` object throws an `IOException`. In this example we merely catch it and print a message, but the controller should ideally throw a more appropriate exception.

> Do Now!
>
> Look at the documentation for the `Readable` and `Appendable` interfaces, and determine which of its implementations can be helpful in the context of a non-interactive test, as well as an interactive program.

The test looks similar to what we had before:

```
  @Test
  public void testStart() throws IOException {
    StringBuffer out = new StringBuffer();
    Reader in = new StringReader("3 4");

    Calculator model = new Calculator();
    Controller controller = new Controller(in, out);
    controller.start(model);
    assertEquals("7\n", out.toString());
  }
```

Receiving multiple inputs

Now that we have abstracted our controller, we can make its logic more complicated by having it receive several inputs and produce multiple outputs.

```
class Controller6 implements CalcController {
  final Readable in;
  final Appendable out;

  Controller6(Readable in, Appendable out) {
    this.in = in;
    this.out = out;
  }

  public void start(Calculator calc) throws IOException {
    Objects.requireNonNull(calc);
    int num1, num2;
    Scanner scan = new Scanner(this.in);
    while (true) {
      switch (scan.next()) {
        case "+":
          num1 = scan.nextInt();
          num2 = scan.nextInt();
          this.out.append(String.format("%d\n", calc.add(num1, num2)));
          break;
        case "q":
          return;
      }
    }
  }
}
```

This calculator repeatedly accepts prefix-expressions with the operator +, until the input "q" is given. We can test this controller by giving it several inputs, and verify the sequence of outputs that it produces.

```
@Test
public void testStart() throws Exception {
  StringBuffer out = new StringBuffer();
  Reader in = new StringReader("+ 3 4 + 8 9 q");
  Controller controller = new Controller(in, out);
  controller.start(new Calculator());
  assertEquals("7\n17\n", out.toString());
}
```