Discussion 3 handout

Requirements reminder

- Form a group of 2-4 classmates seated near enough to allow discussion. Part of the purpose of discussion sections is for you to learn to work *collaboratively* on technical problems (software is written by teams).
- Record the group's responses to each activity on a sheet of paper (the boxes on this handout highlight expected responses). Write the NetIDs of all group members at the top.
- Register your group in CMSX, then upload a photo of the work you completed in section by *Friday evening*.
- It is okay if you do not finish everything during section. Please work at the pace facilitated by your TA (discussion is not a race).

Group members (names & NetIDs)

1.
2.
3.
4.

Objectives

- Evaluate quality of specifications
- Practice writing thorough black-box tests

Prepration: BoundingBox project

Your TA will be demonstrating features of IntelliJ during portions of this section. In order to follow along, you should download the demo code for this activity and extract the Zip file to a known location on your computer. Open it as a project in IntelliJ just as you would an assignment.

Task 1: Evaluating specifications

Evaluate the following method specifications and decide whether you think they are sufficient. Do they contain all applicable clauses (preconditions, postconditions, side effects) in some form? Do they address all corner cases? Are they precise and relatively concise? Most importantly, as a client programmer, would you be comfortable calling these methods without being able to look at their implementation? Write your group's feedback, both good and bad, on your worksheet.

The first example is a static method from an old assignment:

```
/**
 * Return a string that contains the same characters as str, but with each
 * vowel duplicated.
 */
public static String duplicateVowels(String str) { ... }
```

The second example is from Java's `StringBuilder` class, which represents a *mutable* string:

```
/**
 * Returns the index within this string of the first occurrence of the
 * specified substring. The integer returned is the smallest value
 * <i>k</i> such that:
 * <pre>this.toString().startsWith(str, k)</pre>
 * is true.
 *
 * @param   str   any string.
 * @return  if the string argument occurs as a substring within this
 *          object, then the index of the first character of the first
 *          such substring is returned; if it does not occur as a
 *          substring, -1 is returned.
 * @throws  java.lang.NullPointerException if <code>str</code> is
 *          null.
 */
public int indexOf(String str) { ... }
```

The third example is from Java's `Calendar` class. The `roll` method is used when interacting with a date picker (for example, clicking the "next month" arrow in Google Calendar and seeing the same numbered day highlighted):

```
/**
 * Adds or subtracts (up/down) a single unit of time on the given time
 * field without changing larger fields. For example, to roll the current
 * date up by one day, you can achieve it by calling:
 * roll(Calendar.DATE, true).
 * When rolling on the year or Calendar.YEAR field, it will roll the year
 * value in the range between 1 and the value returned by calling
 * getMaximum(Calendar.YEAR).
 * When rolling on the month or Calendar.MONTH field, other fields like
 * date might conflict and, need to be changed. For instance,
 * rolling the month on the date 01/31/96 will result in 02/29/96.
 * When rolling on the hour-in-day or Calendar.HOUR_OF_DAY field, it will
 * roll the hour value in the range between 0 and 23, which is zero-based.
 *
 * @param field the time field.
 * @param up indicates if the value of the specified time field is to be
 * rolled up or rolled down. Use true if rolling up, false otherwise.
 */
public void roll(int field, boolean up) { ... }
```

Task 2: Bounding boxes and black-box testing

In a graphical design program (like PowerPoint), a *bounding box* is an axis-aligned rectangle in a 2D plane that encloses ("bounds") one or more 2D shapes. They may be used to operate on those shapes as a group and to optimize queries over large numbers of shapes.

As a class, **brainstorm** some operations ("behavior") that an object representing a bounding box might want to support. Assume the existence of a `Point` class representing points on the 2D plane.

-
-
-
-

Now open the "dis03" project in IntelliJ and look at the interface in "BoundingBox.java". Do these method declarations cover your desired behavior?

Consider the following method specification from the `BoundingBox` interface:

```
/**
 * Determine whether the specified point is contained strictly within the
 * interior of this box. Points on the boundary of the box are not considered to
 * be contained within its interior.
 *
 * @param p The point to perform the interior test on.  Non-null.
 * @return True if <code>p</code> is strictly within the interior of this box;
 *         otherwise false.
 */
boolean contains(Point p);
```

Open `BoundingBoxTest` under "tests" and add the JUnit 5 dependency. Your goal is to write a test suite thorough enough to catch all of the bugs in the `contains()` implementation in classes `BB1`–`BB6`, which implement the `BoundingBox` interface (some implementations are correct, but others are not).

Write a different test procedure for each "situation" you are testing (e.g. inside the box, outside the box, on an edge, on a corner, etc.), and be sure to add the `@Test` annotation above each procedure. Within each procedure, you may have multiple JUnit assertions involving different points or different boxes.

Each test should involve the creation of a `BoundingBox` and a `Point` and should use `assertTrue()` or `assertFalse()` to indicate the expected outcome for whether that point is contained within the box. Use the `makeBoundingBox()` method instead of invoking constructors directly so that you can easily swap out which class you are testing (if your test procedures only use the `BoundingBox` interface, and not class names, as the static type, they will not need to change).

Write as many test cases as you think are necessary to be confident that the method is implemented correctly, but try to avoid redundancy in your cases—each case should exercise the method in a different geometric regime. Summarize your group's test cases on your worksheet (indicate the rectangle, the point, and the expected result).

Bugs

The “dis03” project provides several classes implementing the `BoundingBox` interface, named `BB1` through `BB6`. They showcase how to implement the interface using different sets of fields to represent its state, but some have buggy implementations of `contains()`. Run your test suite for each implementation; which classes do you think are implemented correctly?

1.
2.
3.
4.

Submission

1. Open the assignment page for “Discussion activity 3” in CMSX
2. [Recorder] Find the “Group Management” section and invite each group member
3. [Others] Refresh the page and accept your invitation
4. [Recorder] Take a picture of your work and save as either a JPEG or a PDF file named “discussion_responses” (you do not need to submit your test code). *After all invitations have been accepted*, upload your picture along with your code as your group’s submission.
    - Recommended scanning apps: Microsoft Office Lens, Adobe Scan, Genius Scan, Evernote Scannable

Ensure that your group is formed and your work submitted before the Friday evening deadline.

Tips and reminders

- Discussion is not a race. Focus on the current activity being facilitated by your TA and engage your whole group to propose and explain ideas.
- Elect a recorder to maintain the “master” copy of your work (others are still encouraged to jot down ideas on scratch paper). Rotate this position each week.
- It is a violation of academic integrity to credit someone for work if they were not present when the work was done, and the whole group is accountable. Your CMS group must only include classmates who attended section with you on that day. Remember that our participation policies accommodate occasional absences without penalty.
- It is your individual responsibility to ensure that you are grouped on CMS and that your group’s work is submitted before the deadline. Log into CMS the day after section to ensure that everything is in order, and contact your groupmates if not. It would be prudent for multiple members to photograph the group’s work.
- Only one group member (typically the recorder) needs to upload a submission. But their submission must not be uploaded until after all group members have confirmed their membership in CMS (contact your TA if you have trouble grouping in CMS).

Challenge problem

If your entire group finishes early and are looking for more practice with this material, try completing this additional task. To be clear, we are not expecting anyone to submit work on challenge problems to CMSX, nor will your TA have time to discuss them in section. But you are welcome to discuss them on Ed or to bring questions and ideas to office hours.

Brainstorm additional state representations (i.e., fields and their invariants) that could be used to implement the `Interval` interface. Have each member of your group implement one representation from scratch, looking only at the signatures and specifications in “Interval.java” (not any of the sample implementations). Verify the correctness of your new implementations using your test suite.
