Eric Roberts
CS 106A

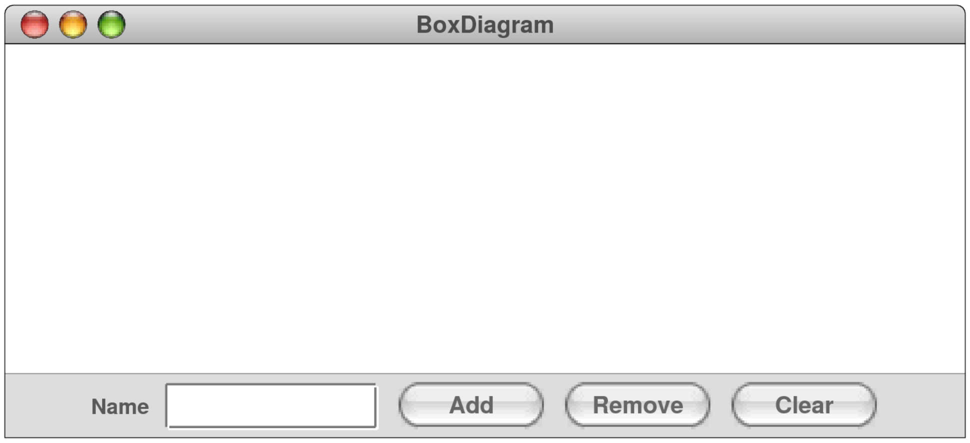
Handout #52
May 21, 2012

Section Handout #7
Interactors and Data Structures

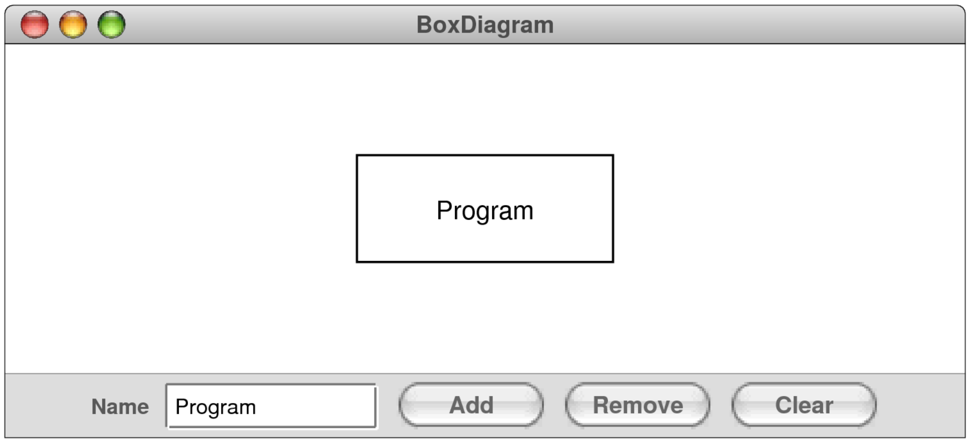
The purpose of this section is to practice the techniques interactors you need for the NameSurfer project in Assignment #6. The first problem gives you a chance to build an interactive application using a control strip; the second has you define a simple class of the same complexity order as **`NameSurferEntry`**.

1. Drawing box diagrams

In this problem, your job is to build a framework for an interactive design tool that allows the user to arrange boxes with labels on the window. The starting configuration for the program presents an empty graphics canvas and a control strip containing a **`JLabel`**, a **`JTextField`**, and three **`JButton`**s. The window as a whole then looks like this:



The most important operation in the program is to be able to add a new box to the screen, which you do by typing the name of the box in the **`JTextField`** and clicking the **Add** button. Doing so creates a new labeled box with that name in the center of the window. For example, if you entered the string **`Program`** in the **`JTextField`** and clicked **Add**, you would see the following result:



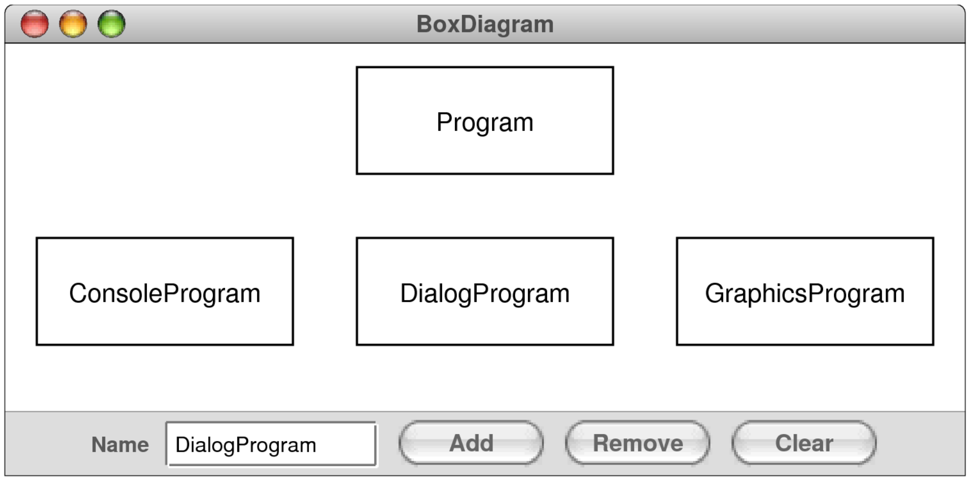
For this simple version of the application, you may assume that the box always has constant dimensions, as specified by the following constant definitions:

```
    private static final double BOX_WIDTH = 120;
    private static final double BOX_HEIGHT = 50;
```

As an extension, you can set the size by dragging a rectangle on the screen.

Once you have created a labeled box, your program should allow you to move the box as a unit by dragging it with the mouse. Because the outline and the label must move together, it makes sense to combine the **`GRect`** and **`GLabel`** into a **`GCompound`** and then use code similar to that in Figure 10-4 from the textbook to implement the dragging operation.

The ability to create new boxes and drag them to new positions makes it possible to draw box diagrams containing an arbitrary number of labeled boxes. For example, you could add three more boxes and drag them around to create the following diagram of the **`Program`** class hierarchy:



The other two buttons in the control strip are **Remove** and **Clear**. The **Remove** button should delete the box whose name appears in the **`JTextField`**; the **Clear** button should remove all of the boxes. While these operations are conceptually simple, they influence the design in the following ways:

- The fact that you may need to remove a box by name forces you to keep track of the objects that appear in the window in some way that allows you to look up a labeled box given the name that appears inside it. You therefore need some structure—and there is an obvious choice in the Java Collection Framework—that keeps track of all the boxes on the screen.
- If the only objects in the window were the labeled boxes, you could implement the **Clear** button by removing everything from the **`GCanvas`**. While that condition applies for this section assignment, you might want to extend this program so that there were other objects on the screen that were part of the application itself that should stay on the screen. In that case, you would want to implement **Clear** by going through the set of boxes on the screen and removing each one.

2. Implementing a class

Implement a class named `CalendarDate` that exports the following methods:

- A constructor that allows the client to create a new date by supplying the month, day, and year as integer values. Thus, it should be possible to write the following declarations that use the `CalendarDate` class:

  ```
  CalendarDate independenceDay = new CalendarDate(7, 4, 1776);
  CalendarDate bastilleDay = new CalendarDate(7, 14, 1789);
  CalendarDate today = new CalendarDate(5, 21, 2012);
  CalendarDate tomorrow = new CalendarDate(5, 22, 2012);
  ```

- Three getter methods—`getMonth`, `getDay`, `getYear`—that allow clients to retrieve these components of a `CalendarDate` object.

- A `compareTo` method that compares the current `CalendarDate` object with a second `CalendarDate` object supplied as an argument. The `compareTo` method should return—in much the same way as it does for the `String` class—an integer that is less than 0 if this date comes *before* the argument date, an integer greater than 0 if this date comes *after* the argument date, and 0 if the two dates are the same. Thus, given the earlier declarations, the expression

  ```
  independenceDay.compareTo(bastilleDay)
  ```

  should return a negative integer, because the original Independence Day occurred earlier than the original Bastille Day. Using the same logic

  ```
  tomorrow.compareTo(today)
  ```

  should return a positive integer, and

  ```
  today.compareTo(today)
  ```

  should return 0.

- A `toString` method that converts a `CalendarDate` object into a `String` in the following form:

  Month day`,` *year*

  where *Month* is the full name of the month, *day* is the day of the month, and *year* is the year. For example, the call

  ```
  independenceDay.toString()
  ```

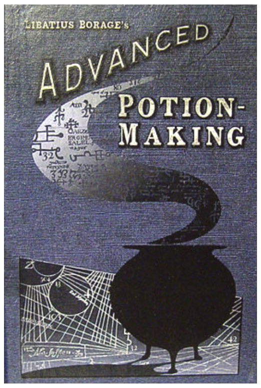
  should return the string `"July 4, 1776"`.

As you write this program, you should keep the following points in mind:

- You may assume that all arguments to the constructor are valid and need not check them to see if they represent a legal date.
- Your class definition should not include any instance variables that are visible outside the `CalendarDate` class itself.

3. Data structure design

In J. K. Rowling's *Harry Potter* series, the students at Hogwarts School of Witchcraft and Wizardry study many forms of magic. One of the most difficult fields of study is potions, which is taught by Harry's least favorite teacher, Professor Snape. Mastery of potions requires students to memorize complex lists of ingredients for creating the desired magical concoctions. Presumably to protect those of us in the Muggle world, Rowling does not give us a complete ingredient list for most of the potions used in the series, but we do learn about a few, including those shown in the following data file:



Potions.txt

```
Polyjuice Potion
shredded boomslang skin
lacewing flies
leeches
knotgrass
powdered bicorn horn
fluxweed
a bit of the person one wants to become

Wit-Sharpening Potion
ground scarab beetle
ginger root
armadillo bile

Shrinking Solution
chopped daisy roots
skinned shrivelfig
sliced caterpillar
rat spleen
leech juice

Draught of Living Death
asphodel in an infusion of wormwood
valerian roots
sopophorous bean
```

As the example illustrates, the format of the data file is a sequence of individual potions, each of which consists of the name of the potion, followed by a list of ingredients, one per line. Each of the potions is separated from the next by a blank line.

Implement a `PotionsManager` class that exports the following entries:

- A constructor that takes the name of a data file (in the format described earlier) and constructs whatever internal structures are necessary to implement the other methods.
- A method named `getPotionNames` that returns an array of the potion names in alphabetical order.
- A method named `getIngredients` that takes a potion name and returns an array of strings that represent the ingredients, or `null` if there is no such potion. The elements of this array should appear in the same order that they do in the data file.