11 User Interactions

Goals

In this lab you will learn a little about programming user interactions using the Model-View-Control pattern for organizing the responsibilities.

In the first part you will just see how to desing a simple test-based interactions with the user. You do not have to add anything, just run the program and use it as a guide in your own projects.

In the second part you will learn quickly how the world based games can be converted to applets. Again, all you need to do is *fill in the blanks* — there are no changes in your program, only two small additional files that allow us to construct the worlds and invoke the *bigBang* method.

The third part is where you actually learn a lot. You have a complete example of a program that builds a GUI based user interface and interacts with it. Use any part of it as a model for your work anytime anywhere. It is based on the JPT library that simplifies many tasks associated with designing GUIs in Java, but makes the essence of what has to be done clearly visible and accessible. Other languages have other libraries for building GUIs - you need to know what are the basic concepts without being bogged down in idiosyncratic details.

The JPT library allows you to concentrate on the key concepts and avoid the pitfalls of multitude of details, typically associated with GUI programming.

Text-Based IO

Download the file **Conversation.zip** and create a project *Conversation*. Add the **jpt.jar** library to your project and you are ready to run it - starting with the *Interactions* class.

It will open a GUI with a button for every *public* method defined in that file that consumes no arguments and has the return type *void*. You can then invoke any of these methods by clicking on the corresponding buttons. So, clicking on the *testSuite* button will run the tests in the *Examples* class.

Look at the implementation of the *conversation* method in the class *Interactions*. You see that it just asks the user for his name and prints a response, then asks the user for his age and prints another response.

Next, look at the class *Person*. We added two simple methods that allow us to save or print the data that an instance of this class represents, and in

reverse, use the input *String* to initialize the fields. the second part is a bit tricky. We first have to use a *default constructor* to create an instance with none of the fields initialized, and then invoke the *fromStringData* method to use the given *String* to initialize the fields. Because all fields have been translated into one *String*, we need to take the *String* apart and also convert the *String* representation of a number into the corresponding *int*.

Run the methods *conversation* and *testStringable* to see the actual interactions. Use this as a guide for designing text-based user interactions.

Making Applets

An Applet is a (small) Java application that runs in a browser, or as a standalone interactive window. The *World* based games you have designed earlier can be converted to applets very easily.

Download the files **BlobWorld.zip** and **draw.jar**. Create a project **BlobWorld** as you did before. Save the file **draw.jar** with your libraries, but make a new folder for it named **AppletDraw**. This is to make sure you do not overwrite your existing *draw.jar* library. Add this library to your project. Open the file BlobWorld.java and run it. Eclipse will ask you whether to run it as an Applet, or will just do it automatically.

As you can see, there is very little in that file:

```
import geometry.Posn;
import draw.*;

public class BlobApplet extends WorldApplet{

  // construct an instance of the BlobWorld
  public World getNewWorld(){
    return
      new BlobWorld(new Blob(new Posn(100, 200), 20));
  }
}
```

For any of your *World* projects you only have to create this file and have the method *getNewWorld* return a new instance of your *World*.

To run this from a web browser, yoou need just a little bit more. The *html* file youneed looks like this:

```
<html>
<head>
<title>My World Applet</title>
```

```
</head>

  <body>

  <h3> Blob World Applet </h3>
  <p>My world consists of a red blob ...</p>

  <applet code="BlobWorldApplet.class" width="200" height="400">
    <param name=cWIDTH value="200">
    <param name=cHEIGHT value="300">
  </applet>

  <hr>

</body>
</html>
```

The only missing part is what are the files you need to load onto the web page. The instructions for doing it will be posted shortly.

The Model and the View

The diagram below (on the next page) describes the classes already included in this application:

Here is a brief description of the role these files play in the application.

The model

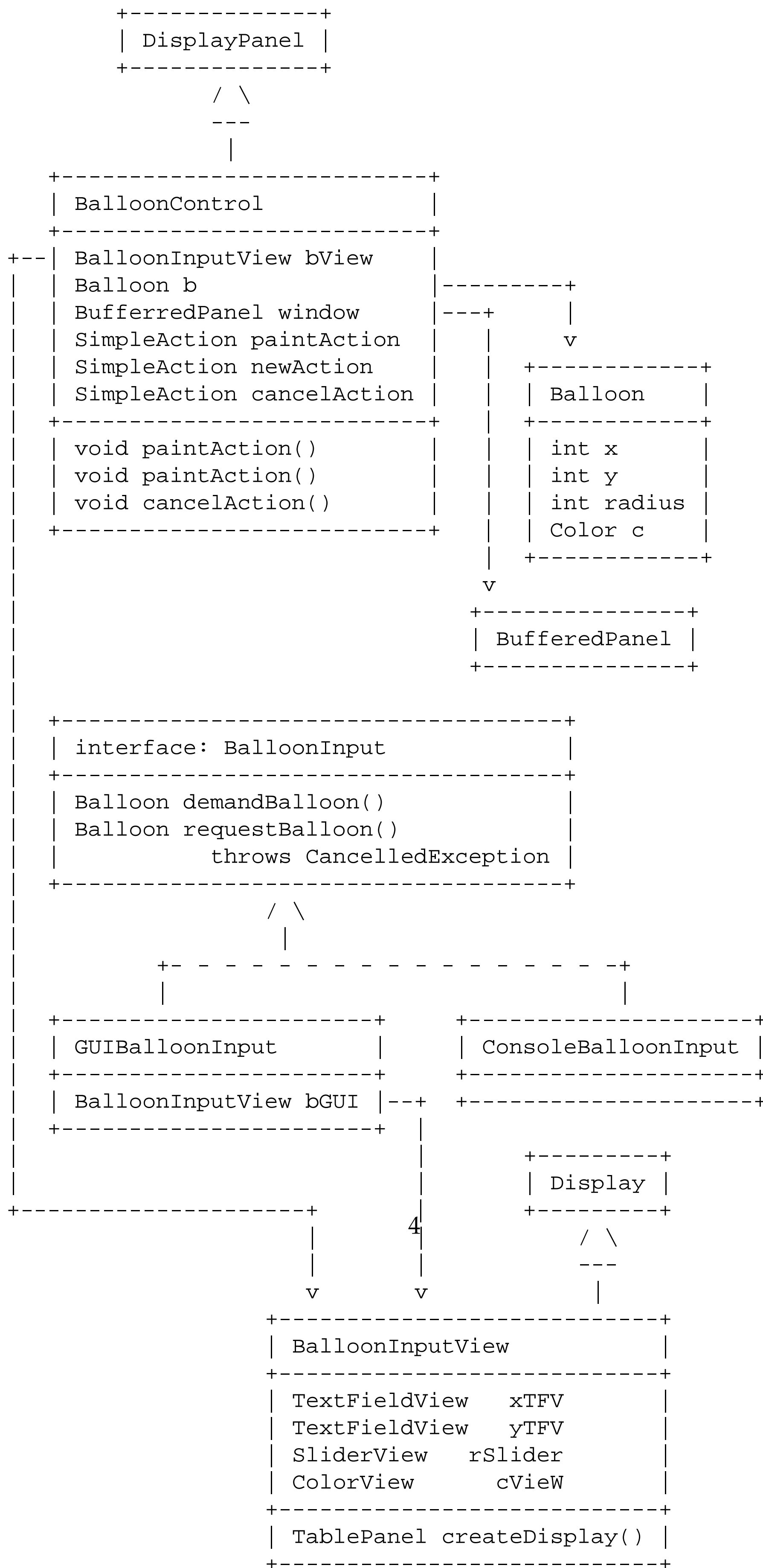
The program deals with balloons (for now just three of them).

- *class Balloon* This class represents one balloon object, allows the user to move it, paint it, and to compare two balloons for closeness to the top of the graphics window.

  We could have other classes here, such as a list of balloons, or a list of tied-up balloons and a list of floating balloons, etc.

The views

We can view the information about a particular *Balloon* object in several different ways. The *BalloonInput* interface provides two methods for getting the data needed to construct an instance of a new *Balloon*.


DisplayPanel
BalloonControl
BalloonInputView bView
Balloon b
BufferredPanel window
SimpleAction paintAction
SimpleAction newAction
SimpleAction cancelAction
void paintAction()
void paintAction()
void cancelAction()
Balloon
int x
int y
int radius
Color c
BufferedPanel
interface: BalloonInput
Balloon demandBalloon()
Balloon requestBalloon()
throws CancelledException
GUIBalloonInput
BalloonInputView bGUI
ConsoleBalloonInput
Display
BalloonInputView
TextFieldView xTFV
TextFieldView yTFV
SliderView rSlider
ColorView cVieW
TablePanel createDisplay()

To display the information about a *Balloon* object, we can print a *String* that represents the *Balloon* object in the console, or paint it in the given window, or display the values of its fields in a GUI.

To get the data from the user that is needed to instantiate a new *Balloon* we can read from the console, or from a GUI.

- *interface BalloonInput* contains two methods: *demandBalloon*() and *requestBalloon*() that allow us to instantiate a *Balloon* object from the source that implements the methods.

- *class ConsoleBalloonInput* implements the *BalloonInput* interface used for reading the input from the console.

- *class BalloonInputView* defines a GUI to request the user input for the data needed to initialize one *Balloon* instance. It contains two *TextFieldView*s, one *SliderView*, and one *ColorView*. It also allows us to display the data that represents an instance of a *Balloon*.

- *class GUIBalloonInput* implements the *BalloonInput* interface for extracting the user input from the *BalloonInputView* GUI.

The control

- *class BalloonControl* adds to the GUI *Action*s. These are buttons that allow the user to choose an action, such as *r*ead the *Balloon* data from a GUI and display the *Balloon* in the given canvas. (Our canvas is a window – a buffered panel.)

Run the code, and note the behavior in response to the various buttons.

Getting Familiar with the Environment

1. **The model**

   Read the code for the *class Balloon*. Add the method *eraseBalloon* which will paint the balloon in a white color (*Color.white*). Make sure you have the examples and tests for this method.

2. **The console input**

Read the code for the method *testConsoleInput* in the *class Interactions*. Describe to your partner what the method does. Look at the *ConsoleBalloonInput* class and see how the methods *demandBalloon* and *requestBalloon* are implemented. Run the code and see what happens if you type in a wrong data, or when you do not provide any input.

3. **The actions**

   Find the code for the action for the **New** button. Currently, it only sets the value of the *Balloon* instance variables. Add to this action a call to the method which paints the balloon, from the *class Balloon*. Make sure it works.

4. **Text input from a GUI**

   Find all places where the *xTFV* is defined or used. It is constructed in the class *BalloonInputView*. This class also defines the methods *demandBalloon* and *requestBalloon*, each of them produces a new instance of a Balloon from the user inputs.

   In the class *BalloonControl* user input to the *BalloonView* initializes the value of a *Balloon* object that represents our model. We could add to our model a list of tied balloons and a list of floating balloons, and more - for example a child holding the balloon.

   DO IT Using a similar technique, define a new *TextFieldView* named *rTFV*, to represent the numerical value of the *Balloon radius*.

5. **Connecting slider with a text field**

   Test the behavior of the slider. Does it have any effect on the balloon? Does it have any effect on the value displayed in the *rTFV* field? Change the value of the *rTFV* field. Does it affect the slider? Does it affect the balloon?

   The two views represent the same value and so should be designed to mimic each other. the slider has to act by changing its position whenever a new value is typed into the text field. The value in the text field has to change when the slider is moved, so it reflects its current position.

   Define two new *SimpleAction*s and the corresponding methods — an *rTFVaction* and a *SliderAction*. It does not matter what you choose for the label, because we are not going to use the actions with a button.

The first one *void rTFVaction* will be invoked when the value in the field *rTFV* changes. It should then set the value of the balloon *radius* and the value of the *rSlider* to the value displayed in the *rTFV*. To set the state of the *rSlider* use the method

rSlider.setViewState("" + *b.radius*);

The second method *void rSliderAction*() will be invoked every time the location of the slider (and the value it represents) changes. It must then change the *radius* of the balloon and set the view state of the *rTFV* calling the method *setViewState* in a manner similar to the above. If you run the program now, you may be surprised to see that these changes have no effect. Can you think of the way to test that the methods work correctly?

6. **Listening to changes in the values**

   Now you have to tell the *rSlider* and the *rTFV* to perform this action when their values change. The following two statements have to be added at the end of the method *void createViews*():

   rTFV.addActionListener(*rTFVaction*);
   rSlider.addSlidingAction(*sliderAction*);

   The first one tells the *rTFV* to perform the *rTFVaction* whenever its value changes. The second one tells the *rSlider* to perform the *sliderAction* whenever the position of the slider (and thus the value it represents) changes.

   Test that this works.

7. **Reporting changes in the model to the view**

   Now that you have seen the method *setViewState*, add such method to the *class BalloonInputView*. To see that is works, we need to modify some of the fields of a *Balloon* instance and invoke the method. Try it.

8. **Adding mouse actions**

   In the last part you will control the balloon with the mouse. You need to define what should happen when the mouse is clicked (or dragged, or released, etc.). You need to specify which GUI component should

*l*isten to the mouse and the user mouse actions. You then need to connect the *MouseListener* with the action it should trigger.

Build a separate frame

The first thing you need to do is to change the manner in which the GUI is displayed. Look at the code in the *class Interactions* for the method *testBalloonControl*(). Replace the line which calls the method *showOKDialog* with the following:

```
JPTFrame.createQuickJPTFrame("Balloon Control", bc);
```

This places the BalloonControl into a window that runs *in its own thread*, i.e. independently of the rest of the application. That allows the rest of the application to watch out for the mouse movement and clicks inside of the graphics window.

Define a mouse action The first mouse action you will build will increase the radius of the balloon by ten, every time you click the mouse. All of this is in the *class BalloonControl*. Start by defining the method
protected click(*MouseEvent mevt*) which does the following:

- Print into the console a message that the mouse was clicked.
- Erase the balloon
- Increase the balloon radius by 10
- Set the view state of the *BalloonInputView bView* to the current values of the balloon. (Only the radius has changed, but it is easier to let the *BalloonView* do the whole job by invoking the method *setViewState*.
- Finally, paint the changed balloon.

9. **Defining and installing Mouse action adapter**

   Install a *MouseActionAdapter* for the *BufferedPanel*as follows:

   - After the definition of the *BufferedPanel*, add the definition:

     ```
     public MouseActionAdapter mouseAdapter;
     ```

- Inside of the constructor for the *class BalloonControl* first initialize the *mouseAdapter* as follows:

  ```
  mouseAdapter = window.getMouseActionAdapter();
  ```

- Add the action to perform when the mouse is clicked as follows:

  ```
  // respond to mouse clicks
  mouseAdapter.addMouseClickedAction(
    new MouseAction() {
      public void mouseActionPerformed(MouseEvent mevt){
        click(mevt);
      }
    });
  ```

  At this point you should test that your program runs as you expected.

10. **Tracking the mouse movement**

    Finally, you will make the balloon move when the mouse moves. Do all the steps you have done for the clicked action, but do not get a new *mouseAdapter*. The following code will add the action:

    ```
    // track mouse motions
    mouseAdapter.addMouseMovedAction(
      new MouseAction() {
        public void mouseActionPerformed(MouseEvent mevt){
          track(mevt);
        }
      });
    ```

    Inside of the *track* method get the coordinates of the mouse as follows:

    ```
    b.x = mevt.getX();
    b.y = mevt.getY();
    ```

    and see what your program does. (Probably nothing - you still have to erase the old balloon, before you make the changes, paint the new balloon, and as a courtesy, set the view state for the view.) Now you should have fun.