Lab 2: User Interactions

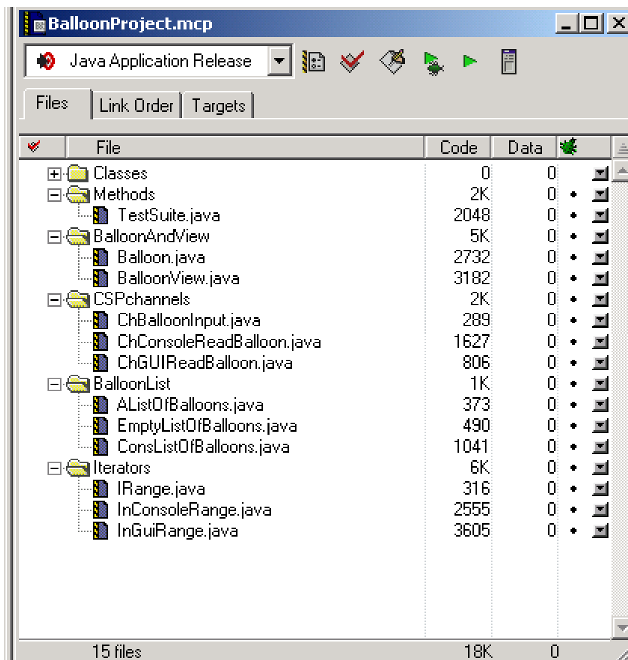
The goal of this lab is to get a first taste of designing programs with user interactions. In general, the persistent information about objects, which are manipulated by the program, is stored in the collection of objects, which represent the *model*. The information about the current state of the model is provided to the user in the form of one or more *views*. The views may also provide for initiating a user interaction with the program, such as requesting that model state be altered, or some model behavior (represented in its methods) be invoked. The communication between the model and the view is implemented in the *controller*.

We will focus on the most basic kind of user interaction, namely providing the input values to be used to define an object in the model.

The context for the lab is again a balloon and a list of balloons, and the view we will use is similar to the one used in earlier labs. In addition, we will also learn how to provide input from the console.

Part A: Getting Started

Download **Lab2.exe,** extract it, open the project. You will see the following files:



- The `Balloon` class is the familiar class and represents the model for this project.
- The `BalloonView` is a somewhat simpler version of the view that was used in earlier projects. It consists of three `TextFieldViews` and one `ColorView`, allowing the user to specify the coordinates of the center, the radius of the balloon and its color.

- There is no explicit definition of the `console`. Every class that wishes to use the console only specifies that it implements `ConsoleAware`. This gives the class access to the `console` object and its methods. However, there is only one console defined for each project and all classes, which implement `ConsoleAware`, share it.
- The three classes in the **CSPchannels** group provide the controller classes that facilitate reading of the `Balloon` data from either the `console` or the `BalloonView`. The `ChBalloonInput` interface specifies a single method
  ```
  public Balloon read();
  ```
  which determines the user's input values and constructs and returns a `Balloon` object.
- The **BalloonList** group contains our standard definition of a list of `Balloon` objects.
- The last group contains the `IRange` iterator interface we have used earlier, and two implementations - one that traverses over the console input and one that iterates over user's input via GUI.

!!!!! Sketch out the UML diagram for this class hierarchy.

Part B: Reading Data for One Balloon Object.

The interface `ChBalloonInput` represents a channel that provides for input of values for one `Balloon` object. The method

```
public Balloon read();
```

will extract the user supplied information from the appropriate view. There are two (incomplete) implementations of this interface. Your task is to complete the implementations.

The `console` input.

The following code processes user input to the console and produces an integer, if user input can be interpreted as such:

```
public int n = console.in.demandInt("Type in an integer");
```

In case of error in the input the user is issued an error message and a prompt to correct the error. The only way to terminate the user interaction is by supplying the input of desired type.

!!!!! Add code to the `ChConsoleReadBalloon` class so that it reads both x and y coordinate and also the radius.

!!!!! Design the method `demandShade`, which notifies the user of input error if the value is not within the range between 0 and 255 inclusive, and repeats the request indefinitely - until a valid input is supplied. This is best implemented by using recursion.

!!!!! Use the method `demandShade` to complete the demand for a `Balloon` object data with a demand for the three color shades - red, green, and blue.

The GUI input.

The following code processes user input from a text field view and produces an integer, if user input can be interpreted as such:

```
public int x = xTFV.demandInt();
```

In case of error in the input the user is issued an error message in a modal dialog, and a demand to correct the error. The only way to terminate the user interaction is by supplying the input of the desired type.

!!!!! Complete the code to in the `BalloonView` class demand the values for the y coordinate and for the radius.

!!!!! Complete the code in the `BalloonView` class to extract the color specified by the `ColorView`. Use the method

```
public Color c = cView.getColor();
```

Part B: Reading Data for Several Balloon Objects.

In many cases the user supplies input for a list of objects in the same class. As before, the input can come from a variety of sources - console and GUI being two examples we will focus on. Typically the user does not know ahead of the time how many objects will be given, but needs to signal in some fashion that all input values have been provided, and the program, which iterated over the input, can continue with other tasks.

On the program side the desired interface is the same as the `IRange` we have used earlier. The program instantiates an iterator for the desired input source, and loops extracting the user's input from `current()` until `hasMore()` indicates that no more input is available, calling `next()` to advance to the next input item.

The implementation of the iterator has the following behavior. The class member data are:

- an object in the target class (in our case `Balloon b`)
- a `boolean` state variable `submitted`, which is set to false while a new set of values is being read in and is `true` if `Balloon b` is properly defined
- a `boolean` state variable `closed`, which is set to `true` when the user indicates that there is no more input available.

The constructor for the GUI iterator constructs a non-modal GUI for user input. The GUI needs to include buttons for *submit* and *cancel* actions - and it needs some member data for this task.

The console object already exists, so there is no need to create one. The console input iterator has no additional member data.

The method `current()` returns the value of the `Balloon` object `b`.

The method `hasMore()` returns the `true` if a value for the balloon has been `submitted` and the input has not been `closed`.

The method `next()` first sets the state variable `submitted` to `false`, then calls the `read()` method, which processes the input from the console, or waits for notification of the input completion when reading input from GUI.

The GUI either processes the input through the *submit* action, which sets the state variable `submitted` to `true`, or it processes the *cancel* action, which sets the state variable `closed` to `true`, and closes the dialog.

In console, if the `read()` method succeeds in accepting user's input, it sets the value of `Balloon b` to this user's input and it sets the state variable `submitted` to `true`. Otherwise it sets the state variable `closed` to `true` and prints a message indicating the end of input.

!!!!! Write the code in the **TestSuite** to test both the console and the GUI input iterators.

!!!!! Modify the code for the console input, so that the balloon coordinates must be given in the 0-400 range, and the radius is given in the 1-200 range.
Hint: Generalize the `demandShade` and `requestShade` methods so they each take the two bounds as arguments.

!!!!! In order to implement the same restrictions on the GUI input, we can use JPT filters. Look at the documentation and find out how to define `RangeFilter.Long`. Define two such filters as additional member data in the `BalloonView` class - one for the `x` and `y` coordinates and one for the `radius`. Look for the documentation for `TextFieldView` and see how you can modify the `demandInt` method calls in the `getBalloon` method.