Exercise 2:
Feature Composition

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software Engineering Design & Construction
SS 2015 - Dr. Michael Eichberg

Your task in this exercise is to design and implement a very small part of a GUI class hierarchy similar to Java Swing. In contrast to Swing, however, your hierarchy will not be monolithic but allow fine-grained feature composition.
Your solution should be as simple as possible. As a first step, make yourself familiar with the code. Add your implementations in the corresponding places and test them thoroughly. We provide a little testbed to test your widgets. As a minimum requirement, your code must at least compile without errors.

Introduction

The base trait of the widget hierarchy is given as follows:

```
trait Widget {
  var bounds = new Rectangle(0,0,0,0)

  def render(g: Graphics2D) {}
  def handleMousePress(e: MouseEvent) {}
  def handleMouseRelease(e: MouseEvent) {}
}
```

There is a trait `TestBed` that you can use as follows to test concrete widget implementations:

```
object MyTest extends Testbed {
  def newWidget() = new MyWidget(...)
}
```

The testbed will call the widget's `render` method whenever it refreshes the screen. Method `handleMousePress` is called when the user presses a mouse button while the cursor is inside the widget bounds. Method `handleMouseRelease` is called whenever a mouse button has been released after a call to `handleMousePress`.
You will design and implement a small widget hierarchy for widgets that can have the following properties:

Labeled The widget has a given string as a label which is rendered in the center of the widget (see the `Utils` object in the code template for a helper method).

Bordered The widget has a 1-pixel-wide border of a given color.

Shaded The widget renders a background of a given background color.

Clickable The widget can be clicked and has two states: *pressed* and *unpressed*. The widget starts in the *unpressed* state. When the left mouse button is pressed while the mouse pointer is inside the bounds of the widget, the widget is in a *pressed* state until the mouse button is released.

Toggleable The widget can be toggled and has two states: *on* and *off*. The widget starts in one state and changes its state to the other (from *on* to *off* or vice versa) when clicked (after it received a release event).

Task 1 With Scala traits

Model each of the above properties as a single trait. Parameters of each property, such as border color or label string, should be modeled as **`var`** fields in the corresponding trait. Use these traits to compose the following widgets, each as a concrete class:

Label A label that has no border or means of interaction. Its constructor should take a single string as the initial label text.

Click Button A button that is shaded, has a border and a label and can be clicked. The widget highlights while pressed (see the `Utils` object for a helper method). Its constructor should take a single string as the initial label text.

Checkbox A checkbox is shaded, has a border and can be toggled. The widget should render a cross or checkmark if it is in the `on` state, none if it is in the `off` state. Its constructor should take a single boolean as the initial state (`true` for on, `false` for off).

Be careful about the order in which you compose traits.

Task 2 In Java

Since Java does not have traits, you cannot use mix-in composition to combine different widget properties. Instead, implement each of the widget properties (`Labeled`, `Shaded` etc) as a class in Java. Find a way to compose these classes into widgets (`Checkbox` etc) and implement the widgets from the previous task as Java classes (there are essentially two different ways to do this). Make sure that your implementation contains no duplication of functionality, i.e., the same functionality *must not* be implemented in multiple places.
Add a short comment to your Java `Widget` class that describes at least two drawbacks of the Java design compared to the Scala solution.[1]

Solve it on your own!

Although this exercise is not graded, it is highly recommended to do it by yourself. Just looking at a solution is much easier in comparison to actually coming up with it.

Requirements if you want to submit your solution

You can, once in the semester, submit your solution to get it corrected. Send your solution to `weiel@st.informatik.tu-darmstadt.de`. Make sure you zip your complete sbt project and make sure that is out of the box working by running `sbt run` and `sbt test`. If the project doesn't compile properly you will not receive any feedback!

[1] This analysis should give you an idea why Swing has a monolithic class hierarchy and how traits would have potentially allowed for a better design.