Exercise 6: Functional Reactive Programming with REScala

TECHNISCHE UNIVERSITÄT DARMSTADT

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

In this exercise, you will develop Pong and a stop watch using the REScala framework. 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. As a minimum requirement, your code must at least compile without errors.
Just as for the last exercise, you need to check out at least two projects: `REScala` and `ex06`, which depends on `REScala`.

Task 1 Pong

You will find a basic implementation of the classic computer game Pong in package `ex06.pong`. When you run application `Main`, you will notice that the ball penetrates the paddles and that the mechanism for counting points is not working. Your task is to

a) add collision detection between the ball and the paddles, so that the ball bounces off the paddles, and

b) implement the point counting mechanism, so that when a ball touches the walls on the left or right behind the paddles, the game correctly keeps track of the points and shows them on the screen.

For the second task, when the ball is about to go out of bounds, you can let it bounce off the wall and increase the corresponding point counter. You don't have to restart the game.
Hint: all you need to do is to add and modify a few signals and events.

Task 2 Stop Watch

You will find a code template for a stop watch in package `ex06.stopwatch`. You should implement the following behavior with REScala:

a) Initially, the window should display "Time: 0.0"

b) When the user clicks the mouse anywhere in the window, the time on the screen should start counting seconds beginning with 0.0.

c) When the user clicks again, the time should stop. Go to 2.

All you need to do in this exercise is to create events and signals that implement the above behavior. Here is one way to do it, but note that there are other, potentially simpler ways to do it. As inputs, we take the `time` signal from `ReactiveSwingApp` and the `clicks` events already present in `ex06.stopwatch.Main`. Then we create the following reactives:

- From `clicks`, we create two events, one that fires every odd occurrence and one that fires every even occurrence. That way, we partition click events into two, one that we will use to start counting time, and one that we will use to stop counting time.
- We create a signal that stores the time when a "stop" click occurred.
- We create a signal that stores the time passed since the last stop click.
- We assemble the final signal that stores the time to display.

The following are all combinators and operations needed to implement the stop watch in the aforementioned way: Signal constructors (`Signal { ... }`), `Signal.snapshot`, `Event.fold`, `Event.toggle`.
The code template contains a `lifeSign` signal that will remind you whether you are really redrawing the screen. A little red circle will be blinking in the upper right corner of the window.

Task 3 Dependency Graphs)

Draw the dependency graphs of the entire pong and stop watch implementations. Each node should have the same name as in the implementation. Edges should be directed and point from a dependency to a dependent, i.e., away from the sources and in the direction of propagation. Draw dynamic dependencies, i.e., edges that are not always present in a dashed style and leave a note when they are active.
You can use the `ticks` and `time` reactives as well as all reactives in the `ex06.Mouse` object as inputs.

Task 4 Stop Watch With Observers

If you would write the stop watch using observers instead of REScala, which *implementation* (not design) problems would you encounter, and how would you solve them? How does a framework such as REScala help you in that regard?
In particular, the code template contains a signal `trueOrFalse`. What do you think the value of `trueOrFalse` should be? What in REScala makes sure that it is indeed the case? How would you solve it with observers? How would you deal with dynamic dependencies (the dashed edges in the dependency graph) in the observer-based implementation, i.e., make sure that they are not always active?

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 sbt run. If the project doesn't compile properly, you will not receive any feedback! In addition to the sbt project make sure to submit a single pdf with the solution of task 3 and task 4.