Exercises, Observer SDJ2

## Traffic light

Copy the Traffic light code from the slides, the part with the subject/listeners working. Run the example and verify that the 3 cars react to the traffic light changing.

Implement the Taxi: It ignores yellow lights, stops when red, and drives when green.

Implement the SleepyDriver:

* If the light is red, and changes to yellow, it doesn’t do anything
* When the light changes to green, it start it’s engine and drives.
* If the light is green, and changes to yellow, it slows down

Implement the Pedestrian. When the cars are waiting for red light, he can cross the road:

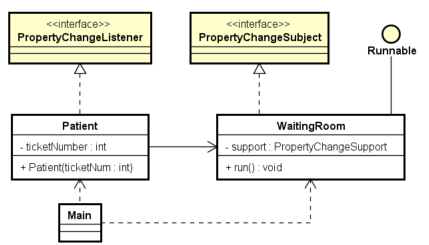
* When the light turns green he waits
* When the light turns from green to yellow, he runs faster across the road.
* When the light turns red he crosses the road.
* When the light turns from red to yellow, he gets ready to cross.

## Traffic light with PropertyChangeSupport

Modify your solution from 3.1 to use the PropertyChangeSupport, PropertyChangeListener and optionally a PropertyChangeSubject interface.

## Exercise – Doctor’s waiting room

You’re going to simulate a Doctor’s waiting room, initially it looks something like this:



The PropertyChangeListener interface is the one from Java. The PropertyChangeSubject is the one from the slides.

In the main class, create the WaitingRoom and run it in a separate thread. In the run method, it should increment a counter from 0, fire an event, and then sleep for e.g. 1 second. This is to simulate that each patient takes a ticket number when entering the waiting room. Occasionally the doctor is ready for the next patient, so the patients are informed by the ticket system in the waiting room, and the patient with the correct ticket number enters the doctor’s office.

In the main class, create a bunch of patients, give them each a ticket number (for-loop?), and add them as listeners to the waiting room.

Upon notification the patient must look up, if it’s their number go to the doctor’s office, otherwise go back to play with phone.

Simulate this with print outs. So, running the program could produce an output like this:

Patient 0 enters waiting room

Patient 1 enters waiting room

Diing!

Patient 0 looks up

Patient 0 goes to the doctor's room

Patient 1 looks up

Patient 1 goes back to looking at phone

Patient 2 enters waiting room

Diing!

Patient 1 looks up

Patient 1 goes to the doctor's room

Patient 2 looks up

Patient 2 goes back to looking at phone

Patient 3 enters waiting room

Patient 4 enters waiting room

Diing!

Patient 2 looks up

Patient 2 goes to the doctor's room

Patient 3 looks up

When a patient enters the doctor’s office, remove them as listeners. They are no longer interested in the ticket numbers.

## Exercise - The soccer match

The target of this exercise is to implement a soccer match, and a number of other classes observing this soccer match.

The SoccerMatch class is uploaded to itslearning. Download it, copy it to Eclipse/IntelliJ.

Create a different class, with a main method. Here an instance of SoccerMatch is created, and the startMatch method is called.

Verify that two things are printed out, with 9 seconds in between: the match has started and ended.

### 1

Change the SoccerMatch class so that it becomes a Subject:

* Implement an interface similar to the one shown in the Traffic Light example
* The SoccerMatch must have a private field of type PropertyChangeSupport, which is used similar to the Traffic Light example.
* Find the four TODOs, here you fire events, using the PropertyChangeSupport. Give meaningful names like “DreamTeamScore”, “OldBoysRoughTackle”. You can also here print out something like “Dream Team scored”.

### 2

Implement two fan classes: DreamTeamFan, OldBoysFan. These should be Listeners. When an event is fired, so that the team they’re rooting for has scored, print out something like “Dream team fans: YEEEAAH!”.

When the opposite team scores, print out some buuhing.

In the main method, create an instance of each, and add them as Listeners to the SoccerMatch. Run the example and verify it is working as expected.

### 3

Implement a Referee class a Listener. He should react to rough tackles, print out something like “Referee gives Old Boys a yellow card for a rough tackle”.

### 4

Create the class ‘AngryCoach’.

When his team scores, he should cheer. When the other team makes a rough tackle, he should yell at the judge.

### 5

Create a class ScoreBoard, it should print out the current score, every time it changes.

### 6

Create a Medic class. They should react to rough tackle updates.

## Exercise – Bird watching

Write a small program to represent the bird watcher example from the slides.

Create a bird class, have it flap its wings or sing a song. Just simulate it with print outs:

“Peacock flashes its wings”

“Peacock whistles”

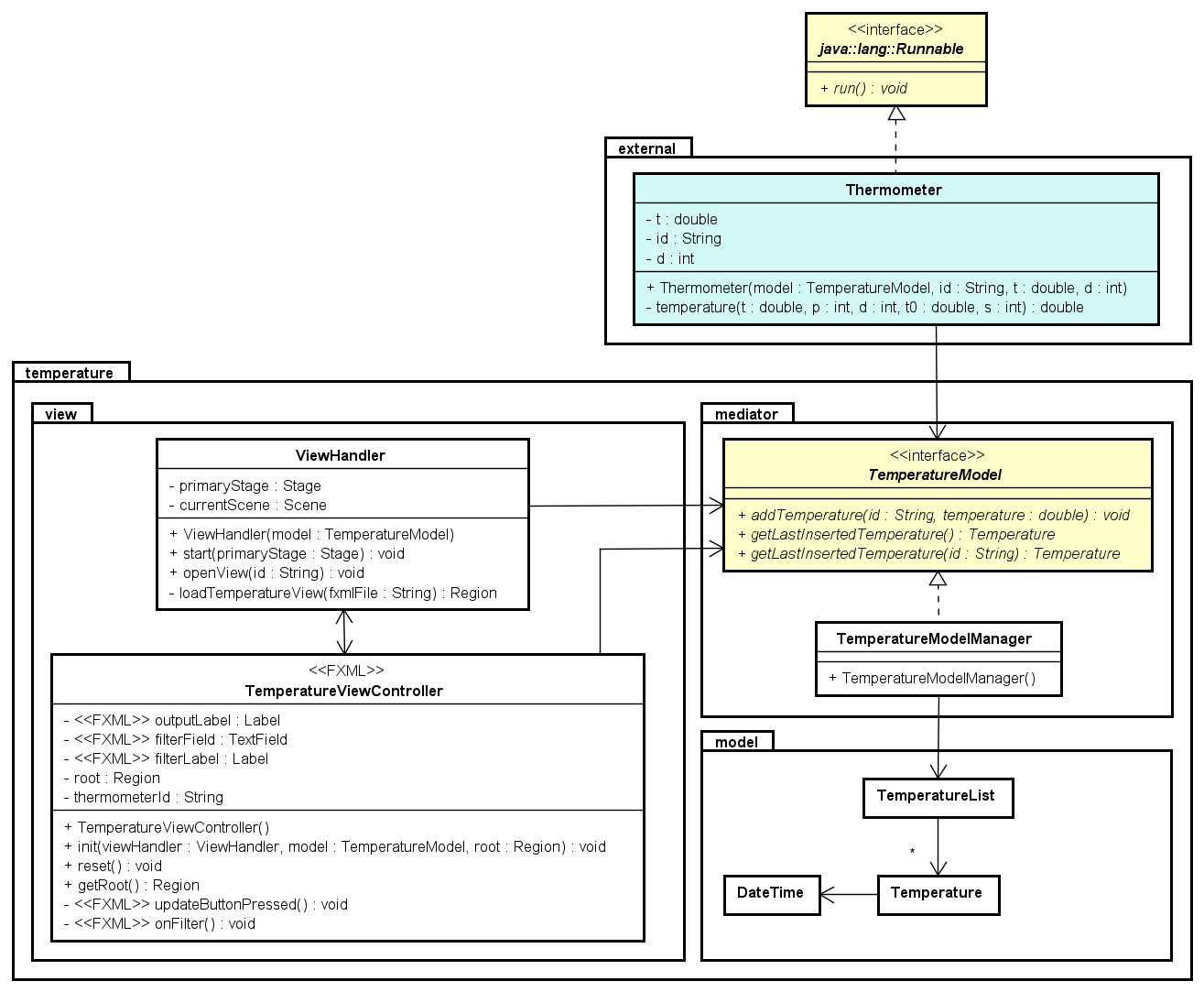
Create a BirdWatcher class, which will react to the birds behaviour with exclamations like: “Ooh”, “How nice”, “Would you look at that”.

Create a BlindBirdWatcher class, which can’t see anything, so he only reacts to the bird’s singing.

## Exercise: Temperature presenter with observer

The purpose for this exercise is to present temperature data generated by an external thread (from a previous exercise). A simple GUI (without observers) is given and you have to use your solution to the Thermometer-exercise.

Class diagram for the full system (Temperature presenter Model and GUI - and external system / Thermometer thread is shown below. *Note that class Thermometer is your solution to a previous exercise and may vary compared to the class diagram*. Model and GUI classes are all uploaded.



### Step 0: Model and GUI classes

Copy model and GUI classes (from the file Thermometer\_observer.zip) into an IntelliJ module. Run the application to see the GUI, and note that there are no data to present (the model is never updated)

### Step 1: Create the external system

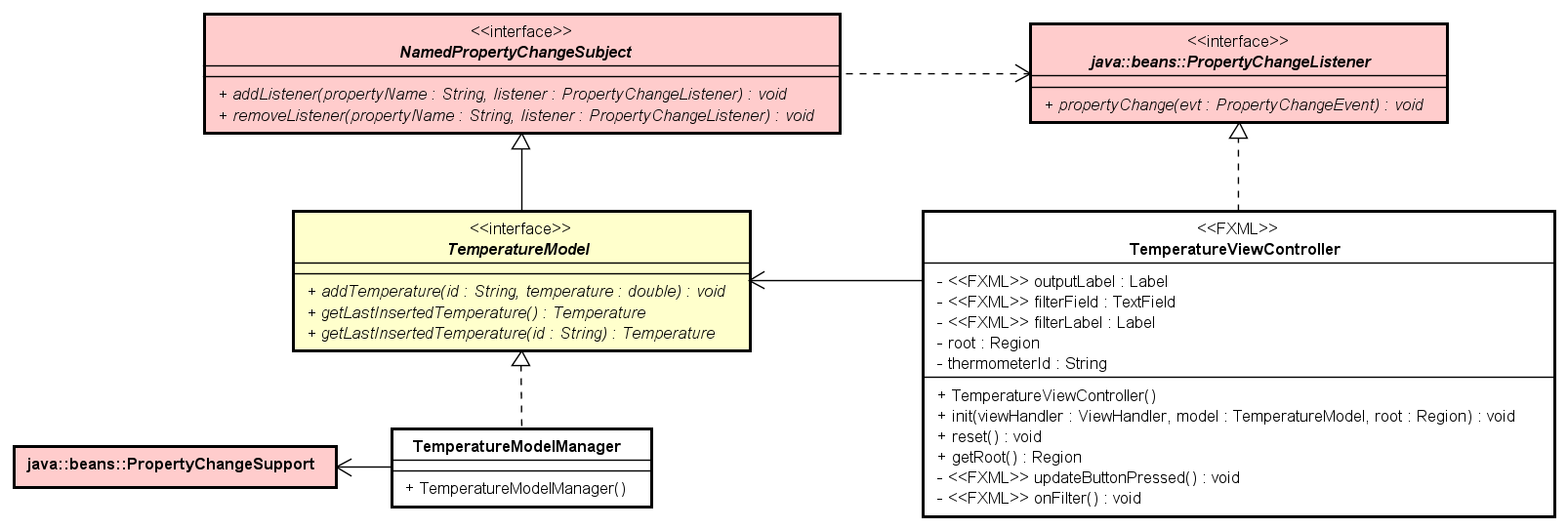
Copy your solution to the Thermometer exercises into the same module, and make the following changes (see also class diagram above)

* Add a model (TemperatureModel) instance variable to Thermometer and modify the constructor to take a model reference and initialize this extra instance variable.
* Modify the run method to call the addTemperature method when a new temperature has been calculated.
* Update class MyApplication to create and start two Thermometer threads.

Run the application again and click the Update button to verify that the data is shown in the GUI. Insert the id for one of the thermometers in the text field and click the filter button, and verify after clicking the Update button a couple of times that you only see temperatures for this thermometer. If you click the filter button with an empty field, temperatures for all (both) thermometers are shown.

### Step 2: Observer

The purpose is to implement the Observer pattern in order to make automatic updates in the GUI every time the model gets a new value. Implement it exactly as shown in the class diagram below:



Notes to the class diagram:

* TemperatureModel (interface) is extending an interface with methods to add and remove a listener for a specific propertyName (e.g. NamedPropertyChangeSubject)
* TemperatureModelManager now has the two extra methods from the interface. Define a PropertyChangeSupport instance variable, initialise it in the constructor and delegate to this 1) when implementing the two method and 2) when firing an event in method addTemperature
  + When implementing the add and remove listener methods you could define listening to all events as a null propertyName, i.e. like the following for add (and similar for remove)

@Override public void addListener(String propertyName,

PropertyChangeListener listener)

{

if (propertyName == null) // all events

{

property.addPropertyChangeListener(listener);

}

else // a specific event

{

property.addPropertyChangeListener(propertyName, listener);

}

}

* TemperatureViewController is implementing PropertyChangeListener with the method propertyChanged – in which you update the label with the value from the event parameter variable. *Note that this statement has to be wrapped into a Platform.runLater.*
* Figure out where in the TemperatureViewController you are going to add and remove it as listener to the model.

This is in the document ‘03cExercises-SDJ2-S20 (Observer- Temperature presenter).docx’.

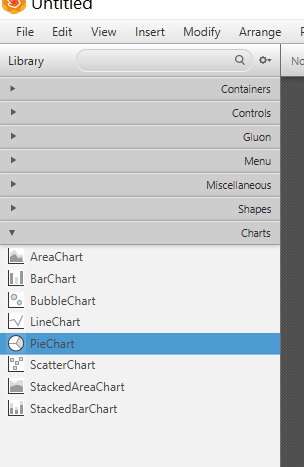
This exercise will help in solving assignment 1.

## Extra Exercise – Data representation

This is for the adventurous ones, it assumes some knowledge about JavaFX.

You’re going to implement the example from the slides about visualizing data in different ways.

In JavaFX there are several components to show data, you’ll have to investigate how to use them. In the SceneBuilder they’re found under the ‘charts’ section.



The uploaded class DataModel has a method, which will calculate three values: red, green, yellow. The sum will always be 100. The numbers will be recalculated whenever the recalculateData method is called.

Modify the DataModel class so that is becomes a Subject: Implement the interface, create a field variable of type PropertyChangeSupport, and fire events from the recalculateData method.

Instantiate the DataModel, e.g. in a main method. Create a while(true) loop, which calls the recalculateData method, then sleeps for a short while.

Run the example, and verify the printed output is as expected.

Create multiple Listener classes. Each class should listen for changes in the DataModel. Each class should present the data in a different way, play around with the chart components.