Lab 3: Interaction

Goals

- program interaction
- practice the mode-view-controller architecture
- practice collaboration in a team with the MVC paradigm

Assignment

Starting from the model-view assignement, where you implemented the models and the views, you will now add the final touch by implementing the interaction.

You will need to implement an overall state-controller that hides and shows views based on user interaction. Controllers of the individual views send messages (show dish, next step, previous step, etc.) to the overall controller.

You will need to implement the model->view notification mechanism (Observable and Observer pattern in the lecture) so that the relevant views will update when the dinner model changes. The dish model is not expected to change.

Finally, you will need to add a controller to each of the view that reacts to the expected user interaction (changing the number of people, adding a dish to the menu, etc.).

How (mobile)

Step 1. Add Observable to the model

To be able to achive updates of the view whenever the model changes, you need to implement the Observer pattern. The first part of the patern is extending Observable in DinnerModel.

When you do this, you should also make sure that in all the methods where you change the model (for example, in the implementation of the setNumberOfGuests method) you actually notify the observers. You do that by calling **setChanged** and then **notifyObservers**. NotifyObservers method can be called with no **arrgument** or you can pass any Object. Passing an argument can be useful to differentiate between different changes. Consider it as an information you send to the view to tell it what specifically changed, so that the view doesn't need to update everything, but just specific components.

Step 2. Implementing Observer in the view

In the previous lab you already passed the model to the view class and you populated the view components with model data. Now it is time to make the view change when model chages. Your task is to implement Observer interface in each of your views (or at least the ones that have changable components with changable data). When you add the interface to your view you will need to implement the **update** method. The update method has the observable and an object as the parameters. The observable is, in our case, the actual model (the one you extended in step 1). The object is the **argument** you optionally passed in your notifyObservers method (see step 1). Based on this your view now knows it needs to update, so you need to get the new values from the model and update the components that show the model data.

In the view constructor class you need to make sure to register the view as a observer of the model. You do that by calling the **model.addObserver(this)**.

Step 3. Creating the controllers

Each view should have one or more controller classes. The main function of the controllers is to respond to the actions performed on the view and, if necessary, communicate the changes to the model. For this reason, the controller classes should have a constructor that passes the model and the view to each controller.

To respond to the actions on the view the controllers need to respond to events. The best way of doing that is by handling input events.

To show the controllers on the example, lets expand on what we had in example of Lab 2. Let us add two buttons to our main layout and give them id: plusButton and minusButton. In the ExampleView we wil store them in the class variables **plusButton** and **minusButton**. We also need to get them from the actual layout in the ExampleView constructor:

```
plusButton = (Button) view.findViewById(R.id.plusButton);
minusButton = (Button) view.findViewById(R.id.minusButton);
```

Now we are ready to construct our controller. It is important to put the controller class in the same package (folder) as the view class. The full controller class would look like:

```
public class ExampleViewController implements OnClickListener {
 DinnerModel model;
 MainView view;

 public ExampleViewController(DinnerModel model, MainView view ) {
  this.model = model;
  this.view = view;

  // Here we setup the listeners
  view.plusButton.setOnClickListener(this);
  view.minusButton.setOnClickListener(this);
 }

 // This is the method of that we need to implement when implementing
 // the OnClickListener. Notice that the View here is an Android View
 // class (parent class of all the components), not the view we talk
 // about in the lab instructions.
 @Override
 public void onClick(View v) {
  if(v == view.plusButton) {
   // We update the model
   model.setNumberOfGuests(model.getNumberOfGuests() + 1);
  }
  if(v == view.minusButton) {
   // We update the model
   model.setNumberOfGuests(model.getNumberOfGuests() - 1);
  }
 }

}
```

In the controller constructor we add the constructor it self to the button **listeners**. In the **onClick** method we check which button was clicked and do the appropriate change to the model.

This is just one of many listeners you can implement, depending on what events you need to perform. Be sure to check the documentation of the components you are using to see which events it can respond to and what listeneres exists.

Apart from changing the model, some of the events you will respond to will need to launch different activities. Here is the simplest way to do from your main activity:

```
Intent intent = new Intent(this, OtherActivity.class);
startActivity(intent);
```

To make all of this work, we need to instantiate the controller somewhere. This is done in the activity that is using the controller. For our example this would be the in the MainActivity, right after instantiating the example view we create the controller:

```
// Creating the view class instance and passing the model
ExampleView exampleView = new ExampleView(findViewById(R.id.this_is_example_view_id), model);
ExampleViewController exampleCtrl = new ExampleViewController(model, exampleView);
```

Step 4. Swiping between the screens (optional)

Instead of using next and previous buttons for the navigation, a more interesting interaction would be to swipe between the screens. If you want to experiment with this, please check the documentation.

How (desktop)

Step 1. Add Observable to the model

To be able to achive updates of the view whenever the model changes, you need to implement the Observer pattern. The first part of the patern is extending Observable in DinnerModel.

When you do this, you should also make sure that in all the methods where you change the model (for example, in the implementation of the setNumberOfGuests method) you actually notify the observers. You do that by calling **setChanged** and then **notifyObservers**. NotifyObservers method can be called with no **arrgument** or you can pass any Object. Passing an argument can be useful to differentiate between different changes. Consider it as an information you send to the view to tell it what specifically changed, so that the view doesn't need to update everything, but just specific components.

Step 2. Implementing Observer in the view

In the previous lab you already passed the model to the view and you built the view components using the model data. Now it is time to make the view change when model chages. Your task is to implement Observer interface in each of your views (or at least the ones

that have changable components with changable data). When you add the interface to your view you will need to implement the **update** method. The update method has the observable and an object as the parameters. The observable is, in our case, the actual model (the one you extended in step 1). The object is the **argument** you optionally passed in your notifyObservers method (see step 1). Based on this your view now knows it needs to update, so you need to get the new values from the model and update the components that show the model data.

In the view constructor class you need to make sure to register the view as a observer of the model. You do that by calling the **model.addObserver(this)**.

Step 3. Creating the controllers

Each view should have one or more controllers. The main function of the controllers is to respond to the actions performed on the view and, if necessary, communicate the changes to the model. For this reason, the controller classes should have a constructor that passes the model and the view to each controller. More then one controller per view is usually needed when you have two different semantics for the same action on the view (remember the example from the lecture, drawing a circle and a square).

To respond to the actions on the view the controllers need to respond to events. The best way of doing that is by implementing event listeners.

To show the controllers on the example, lets expand on what we had in example in Lab 2. Let us add two buttons to our MainView and store them in the class variables **plusButton** and **minusButton**.

Now we are ready to construct our controller. It is important to put the controller class in the same package (folder) as the view class. The full controller class would look like:

```
public class MainViewController implements ActionListener {

 DinnerModel model;
 MainView view;

 public MainViewController(DinnerModel model, MainView view ) {
  this.model = model;
  this.view = view;

  // Here we setup the listeners
  view.plusButton.addActionListener(this);
  view.minusButton.addActionListener(this);
 }

 // This is the method of that we need to implement when implementing
 // the ActionListener.
 @Override
 public void actionPerformed(ActionEvent e) {
  if(e.getSource() == view.plusButton) {
   model.setNumberOfGuests(model.getNumberOfGuests() + 1);
  }
  if(e.getSource() == view.minusButton) {
   model.setNumberOfGuests(model.getNumberOfGuests() - 1);
  }
 }

}
```

In the controller constructor we add the constructor it self to the button **listeners**. In the **actionPerformed** method we check which button was clicked and do the appropriate change to the model.

This is just one of the many listeners you can implement, depending on what events you need to perform. Be sure to check the documentation of the components you are using to see which events it can respond to and what listeneres exists.

To make all of this work, we need to instantiate the controller somewhere. This is done at the same place where we previously instantiated our view. class. For our example this would be the in the DinnerPlanner class, right after instantiating the main view we create the controller:

```
//Creating the first view
MainView mainView = new MainView(model);

MainViewController mainCtrl = new MainViewController(model, mainView );
```

By implementing the controllers for all the views and listeners for the required events you should be done with the interactivity of your application. Some events will result in the model change (like in the example), some will just change the information in the view (like typing a text in the search field) while others will just launch other views. Make sure that whenever you create a view you also create the controller(s) for it.

Step 4. (Optional) DnD the dishes

The prototype defines that you should be able to add the dishes to the menu with Drag and Drop functionality. This is the desirable interaction, however if it requires a bit more of development effort. For this reason we leave this as an optional task, if you have time and motivation to explore. You can study the documentation on how to achieve this.

If you chose not to implement Drag and Drop support add a button next to each dish which should be used for adding dishes to the menu.

Uploading the results

When you have finished with developing either mobile or desktop application, just push your changes to GitHub and submit the repository link in the comment of **Bilda assignment** Lab 3: interaction.

If you didn't use GitHub, **zip** your whole project folder and upload it to the Bilda assignment.

Show earlier events (2) ›

Teacher Filip Kis changed the permissions | igår 19:59

Kan därmed läsas av studerande och lärare och ändras av lärare.

... or write a new post

Students, teachers and assistants of this course can read.

Last changed: 2014-02-10 20:00. Show versions

Tags: None so far. Add

Follow this page Report abuse