SOFT3202

Adam Ghanem

May 4, 2022

Contents

1 Tutorial 8 EXPORT

1.1 N-Tier Architectures

1.1.1 Description

This tutorial will be focusing purely on the n-tier architectures and how we can design effective GUI programs. The core idea is being **able to separate the view code from the application logic.** The most basic requirement of this in achieving model-view separation is extremely important across all these design systems.

But why N-tier what is the notion behind it? Well whenever we have this moderately scaled systems we can group together separate classes that have

a similar application or area they are dealing with. In this instance we group together classes responsible for dealing with application logic or classes dealing with the presentation of GUI elements. We currently have two tiers in the form of:

1. Model
2. View

In reality, we can have many more views on top of this including security layers etc. But in most cases we will be adding:

1. Controlller

   However, before continuing on with the three tiered architectures it is much more important to understand how to separate our model and view in the first place. This will cement our understanding and allow us to apply it to these three tiered patterns.

1.1.2 Model - View

As we said the most basic scenario we want to achieve in these GUI programs is model-view separation. Remember that the view is responsible for showcasing information to the user and taking inputs from the user.

The model on the other hand is where we store our POJO data and where we interact with the application data, be it database material etc. We should represents aspects such as Ships and Loans in this manner, an we represent changes that occur to them here as well, e.g. using fuel.

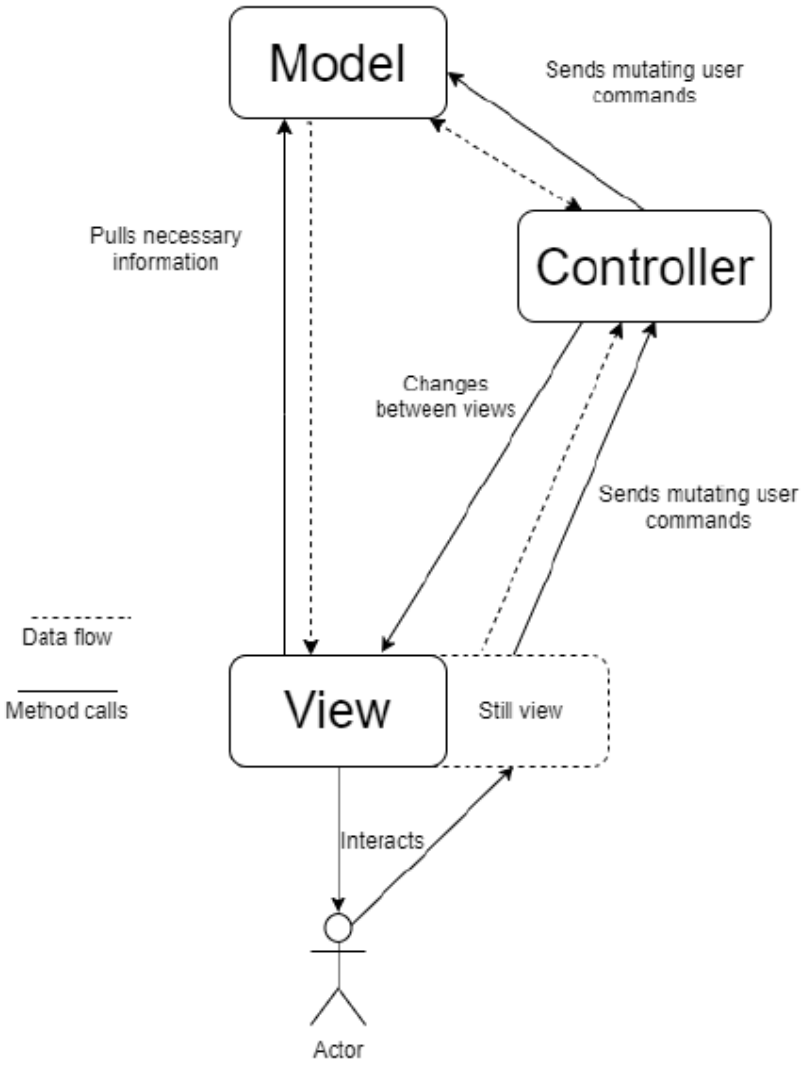
1.1.3 MVC

Think of our normal MV design but we slap on a Controller to interface with the user's interaction. This is the recommended n-tiered architecture as it is probably the most straightforward to get working.

Now the controller acts as a mechanism for interacting between the Model and the View. The core idea of this controller is to **deal with any action that causes state changes between the model and views**. So in lamens terms this basically means that the controller works in between the model and the view and manages everything that results in changes of state.

For a model this is quite straightforward as we will definitely let the controller handle instances when its state needs to be updated. For a view on the otherahnd the controller will manage switching between views not specifically what should appear internally in that view - this is the particular responsibility of the view class alone.

Sniped from the lectures:



And also sniped from Martin Fowler (18 July 2006) *GUI Architectures* blog post:

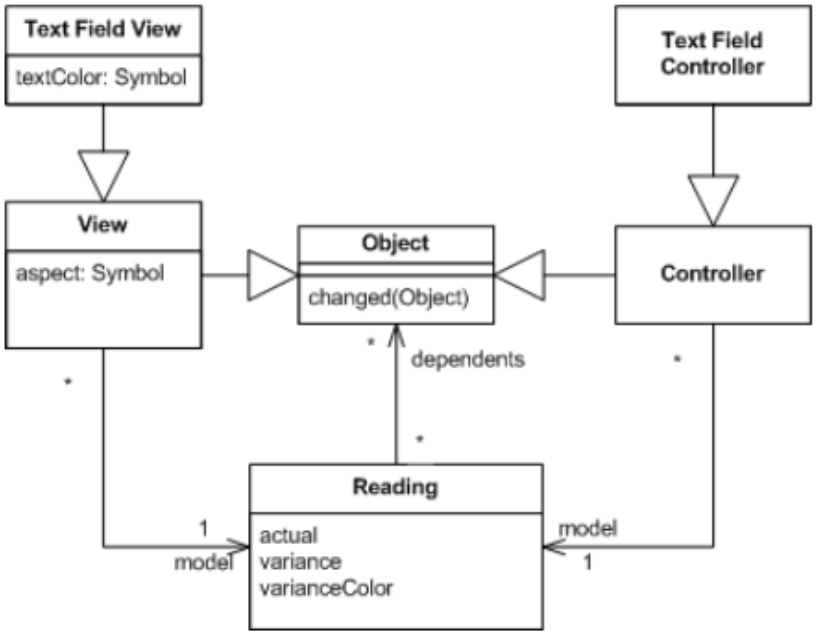


Figure 5: Classes for an MVC version of an ice-cream monitor display

And the sequence diagram as well:

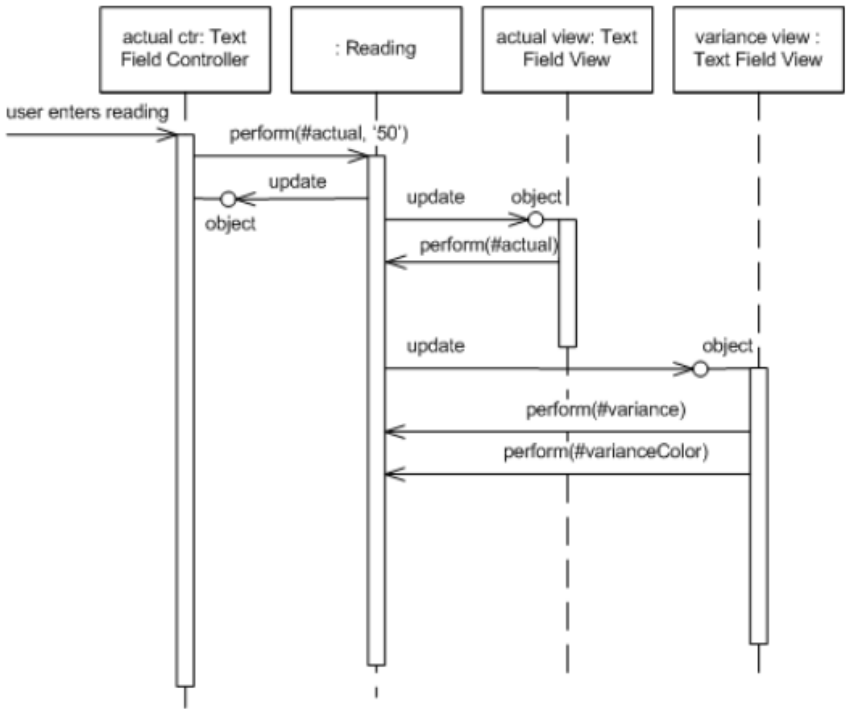


Figure 6: Changing the actual value for MVC.

What we notice from the above is the same ideas we had with MV separation but we have both the views and controllers essentially observing our model objects. Whenever there is a change to the state of our model then the view reacts accordingly. There are multiple ways we can do this in JavaFX but one commoon way is to bind using JavaFX properties. The example below should help here with a very simplistic view of a controller:

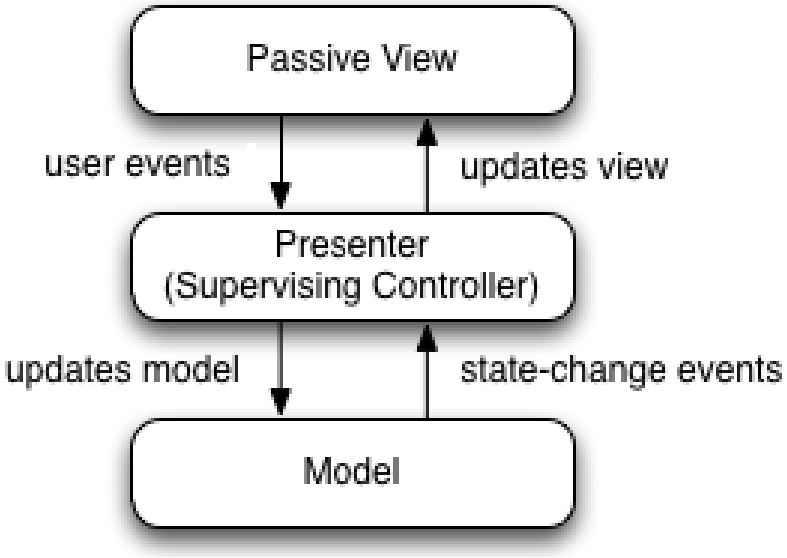
NOTE: This contains multiple approaches to completing your MVC, you can use `Observers`, `bind()`, `FXML` specific properties, `Integer/String/Property` etc. Please make use of the documentation and resources provided during the tutorials to find out about each respective method e.g. for Observables `https://docs.oracle.com/javase/8/javafx/api/javafx/collections/FXCollections.html`.

```
// CONTROLLER
public class ShipController {
```

```
  @FXML
  private TableView tableView;

  // Or if you are not using FXML
  private ShipView shipView;

  private final ShipManager ships;

  public ShipController(ShipManager ships) {
    this.ships = ships;
  }

  public void setUp() {
    // initialise ship etc and set up bindings
    // Instead of doing this we can also have separate control flow to check
    // when it is changed then update the view.
    shipView.getTextField().textProperty().bind(ships.getName());
  }

  public void addShips() {
    if (shipManager.checkValid(...)) {
      shipManager.addShip(tableView.getRow());
      shipManager.changeName(...);
    }
  }
}

// VIEW
class ShipView {

  private TextField name;
  private TableView table;
  private ShipManager ships;

  // structure, layout etc

  // Instead of having this logic in the controller it could be in the view
  // such that the view observes the model
```

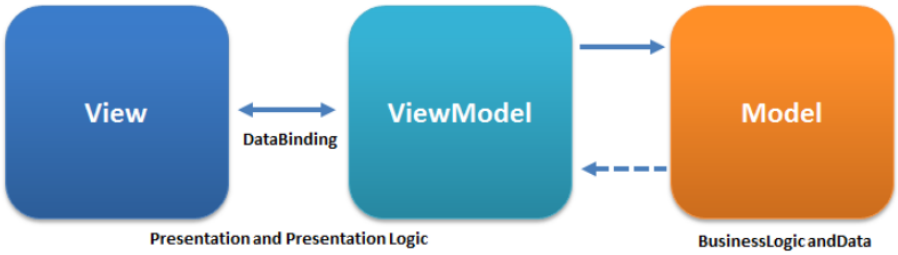
```
  private void updateShipsTableName() {
    // If we want to do something specific we do an addListener
    ships.getShips().addListener((...) -> updateRowsOfTable());

    // If we just want the text properties to be linked then we can do a binding
    name.textProperty().bind(ships.getName());
  }
}

// MODEL
class ShipManager {
  private List<Ship> ships;
  // We can also make use of:
  private final ObservableList<Ship> shipsList =
    FXCollections.observableArrayList();
  private String name;

  public ShipManager() {}

  // Some helper methods etc

}

// Main Class
class App extends Application {

  @Override
  public void start(Stage stage) {

    // Since this is the entry point of our program
    // we initialise our controller and continue from here:
    this.stage = ...
      Controller mainController = new ....;
    // Or we can use FXML
    Controller mainController = fxmlLoader.getController();
  }
}
```

1.1.4 Model View Presenter (MVP)



In this implementation the model and view act similarly as previously but the presenter is now responsible for retrieving data from the model and then passing this information onto the view. You can think of this as a man-in-the-middle, it interfaces all interactions that were previoulsy occurring between the model and the view.

1.1.5 Model View View-Model (MVVM)



In this scenario here, the model remains as is. It is still the representation of our application data. The view too remains the same and is the presentation of the these elements in the GUI, the structure and style.

The View-model is a bit different here than in the MVC and the MVP, In the presenter with the MVP it acts as an interface between the model and the view and contains references to the views. However in this design pattern the View Model has a binder which allows views to link to and receive their updates from the models. The binding technology used is different in each solutions such as XAML.

The binder will basically look out for changes in the View Model and then ping the View of the changes it finds and vice versa. Instead of directly observing the model the view will observe the ViewModel. The view model in this scenario then represents that state of the data in our application.

1.2 IMPORTANT Testing Update

So for those of you who attended my classes may have witnesssed me wrapping specific API implementations to provide the ability to mock our own classes for testing. In terms of the tasks for your assignment **THERE WILL BE NO REQUIREMENT TO TEST API CALLS**. Instead the way your program should be designed should facilate testing of individual modules up to the HTTP/API classes.

The recommended strategy is to use a **facade of the API** which implement from a parent interface. Now when we are testing we should be able to break down and mock this facade which will allow us to test individual components e.g. JSON Readers, Model POJOs, Separate classes for HTTP endpoints etc. Just to reiterate you are **NOT REQUIRED TO TEST HTTP CALLS** you are not required to wrap those Java objects on an individual level but you should have those API classes that are solely responsible for HTTP calls, this means you do not need to test it but you have abstracted all the information out that needs to be tested. For example, having a class containing HTTP endpoints, JSON parsing and creation of those objects would be extremely difficult to test, these should be separated through abstractions instead.

For those of you who are interested, testing these http calls could involve using external tools, creating HTTP servers within units tests or doing the wrapping method mentioned initially that would allow us to inject/return static values when we want to.

1.3 Questions

1. How to test when we have external dependencies?

   Addressed above.

2. Controller isn't really necessary? I have only one main page?

   Unless your code has no state changes then an argument can be made to house a controller. Even if there is a single page, no API provided is simple enough such that there is only one logical component. We can split controllers based on the concerns they are managing, just like models and views we should decouple them even if we have a single page solution. These can be grouped based on search functionality, another on managing logins, another for containg logic about the item we searched for and etc.

3. Concurrency, how crap can it be?

   This should be disclosed in the next milestone updates.