Question 2: MVVM + Observer pattern

IT-SDJ2-A21

Software Engineering

VIA University College

Jordi Lazo

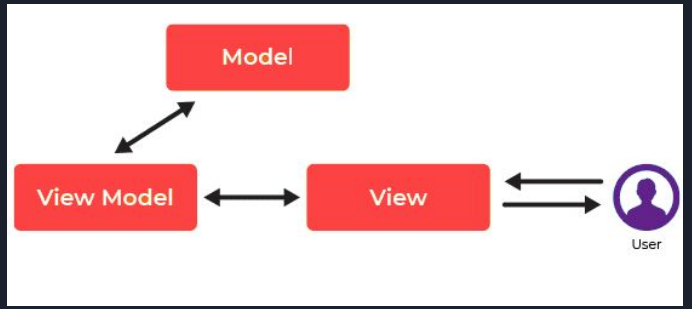
MVVM (Model–View–ViewModel)

What is the purpose of MVVM?

- MVVM is a software architectural pattern that facilitates the separation of the development of the graphical user interface (view) from the development of the business logic (model) so that the view is not dependent on any specific model and totally independent.
- MVVM helps to cleanly separate the business and presentation logic of an application from its user interface.
- No direct communication between View and ViewModel, only the ViewModel is aware of the needs of the View.
- It makes code more maintainable.

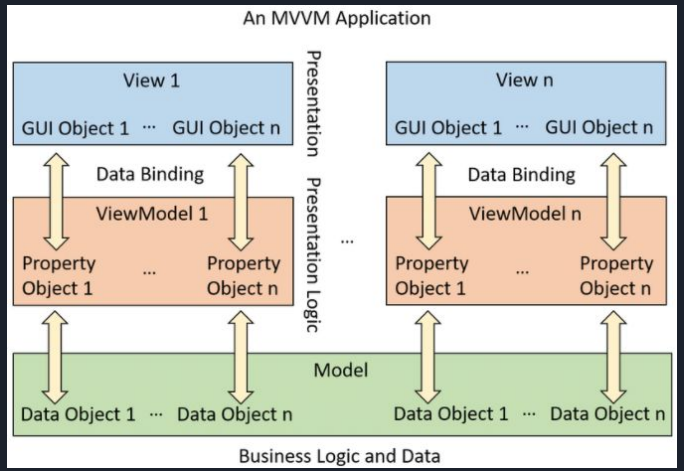
What are the different parts involved and their purpose?

- Parts involved:
  - Model → stores data and related logic. It represents data that is being transferred between controller components or any other related business logic.
  - View → is the structure, layout, and appearance of what a user sees on the screen (also receives user input).
  - ViewModel → this is where the controls for interacting with View are housed, while binding is used to connect the UI elements in View to the controls in ViewModel.

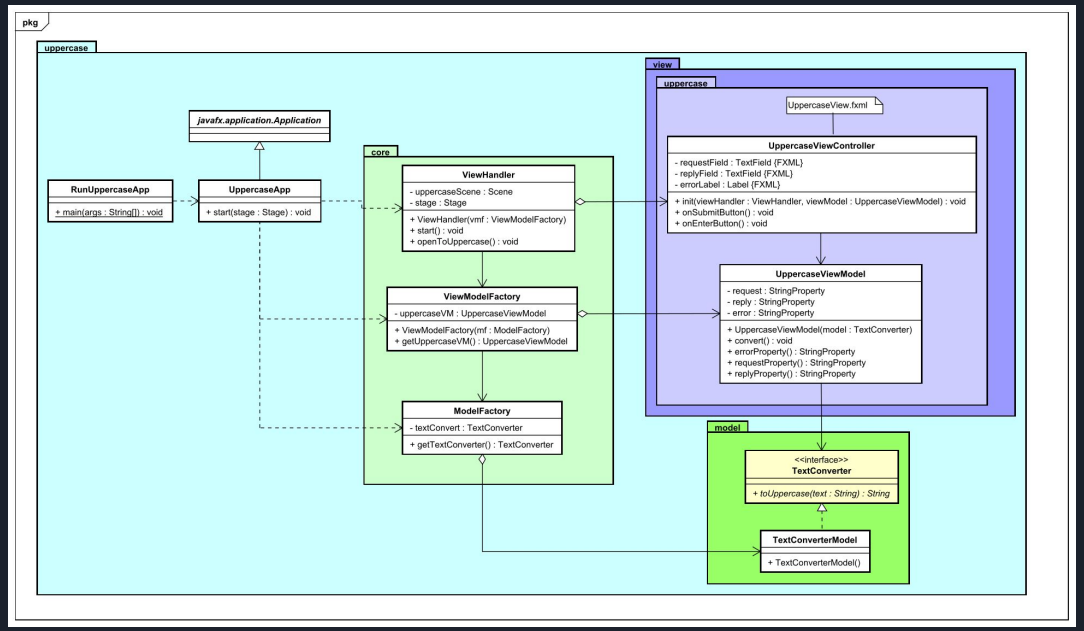


How do the different parts interact?

- Communication between the View and ViewModel is through some property and binding.
- Models are connected directly to the ViewModel and invoke a method by the model class.



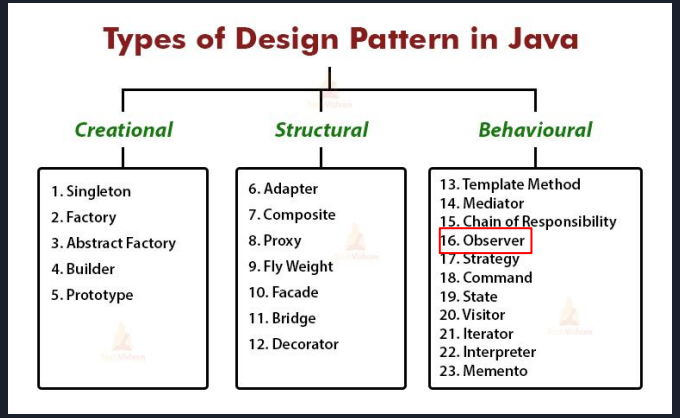
UML example


pkg
uppercase
javafx.application.Application
RunUppercaseApp
+ main(args : String[]) : void
UppercaseApp
+ start(stage : Stage) : void
core
ViewHandler
- uppercaseScene : Scene
- stage : Stage
+ ViewHandler(vmf : ViewModelFactory)
+ start() : void
+ openToUppercase() : void
ViewModelFactory
- uppercaseVM : UppercaseViewModel
+ ViewModelFactory(mf : ModelFactory)
+ getUppercaseVM() : UppercaseViewModel
ModelFactory
- textConvert : TextConverter
+ getTextConverter() : TextConverter
view
uppercase
UppercaseView.fxml
UppercaseViewController
- requestField : TextField {FXML}
- replyField : TextField {FXML}
- errorLabel : Label {FXML}
+ init(viewHandler : ViewHandler, viewModel : UppercaseViewModel) : void
+ onSubmitButton() : void
+ onEnterButton() : void
UppercaseViewModel
- request : StringProperty
- reply : StringProperty
- error : StringProperty
+ UppercaseViewModel(model : TextConverter)
+ convert() : void
+ errorProperty() : StringProperty
+ requestProperty() : StringProperty
+ replyProperty() : StringProperty
model
<<interface>>
TextConverter
+ toUppercase(text : String) : String
TextConverterModel
+ TextConverterModel()

Observer pattern

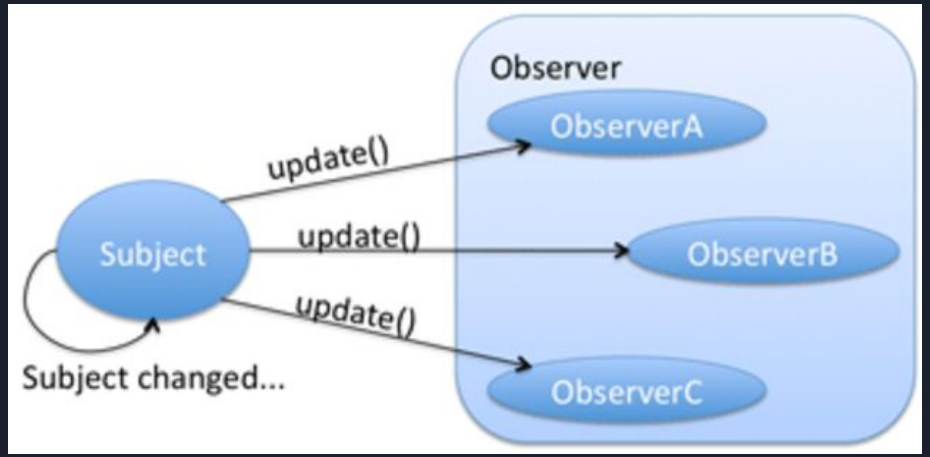
What is Design Pattern?

A software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. Design patterns are solutions to general problems that software developers faced during software development.



What is the Observer Pattern?

It specifies communication between objects: observable/subject and observers/listeners. An observable/subject is an object which notifies observers/listeners about the changes in its state.



When to use observer design pattern?

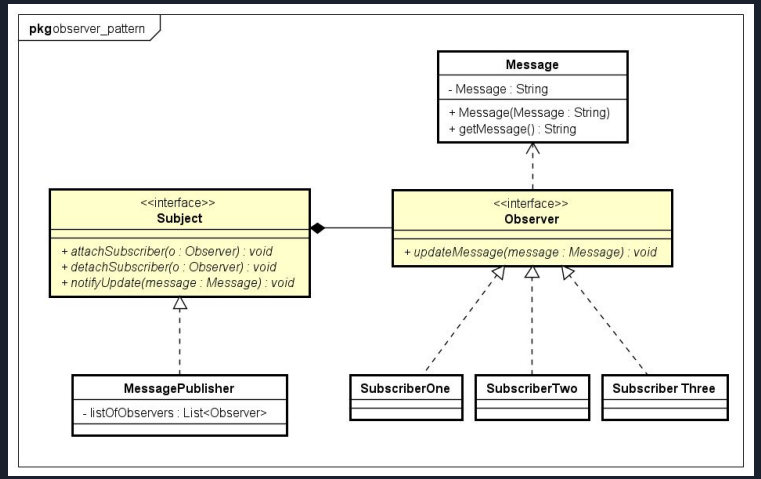
When you have a design a system where multiple entities are interested in any possible update to some particular second entity object, we can use the observer pattern.

Example:

- A real world example of observer pattern can be any social media platform such as Facebook or twitter. When a person updates his status – all his followers gets the notification. A follower can follow or unfollow another person at any point of time. Once unfollowed, person will not get the notifications from subject in future.

Observer design pattern example

In my example I will create a class called *MessagePublisher* of type *Subject* and three *Subscribers* of type *Observer*. *MessagePublisher* will publish the message periodically to all *Subscribers* and they will print the updated message to console.



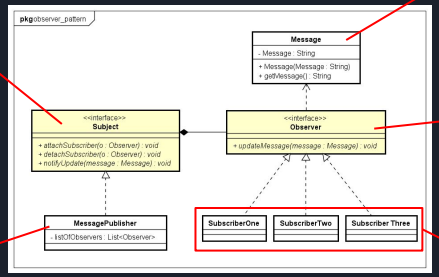
```java
public interface Subject {
    void attachSubscriber (Observer o);
    void detachSubscriber (Observer o);
    void notifyUpdate (Message message);
}
```

```java
public class Message {

    private final String message;

    public Message(String message) { this.message = message;

    public String getMessage() { return message; }
}
```




```java
public interface Observer {
    void updateMessage (Message m);
}
```

```java
public class MessagePublisher implements Subject{

    private final List<Observer> listOfObservers = new ArrayList<>();

    @Override
    public void attachSubscriber(Observer o) { listOfObservers.add(o); }

    @Override
    public void detachSubscriber(Observer o) { listOfObservers.remove(o);

    @Override
    public void notifyUpdate(Message message) {
        for(Observer allSubscribers: listOfObservers){
            allSubscribers.updateMessage(message);
        }
    }
}
```

```java
public class SubscriberOne implements Observer{
    @Override
    public void updateMessage(Message m) {
        System.out.println("Subscriber one [notification]: "+m.getMessage());
    }
}
```

Thanks for your time!