Architectural Patterns

An architectural pattern for a GUI application in Java

An architectural pattern, is a reusable and proven solution to a commonly occurring problem in software architecture.

Architectural patterns help solving concerns with scalability, maintainability, extensibility, and modifiability. They provide a blueprint for structuring and organizing the codebase, promoting separation of concerns, and improving overall system design.

Architectural patterns are often used to guide the development of large-scale software systems and help ensure that the resulting system is robust, efficient, and meets the desired quality attributes.

Architectural patterns help with testing and debugging, as they provide a clear separation of concerns and a well-defined structure for the codebase. They also help with code reuse, as they provide a reusable solution to a commonly occurring problem.

Separation of concerns is a design principle that states that a software system should be divided into distinct sections, each addressing a separate concern. This helps with code reuse, as it allows developers to reuse the same code in different parts of the system without having to rewrite it. It also helps with testing and debugging, as it allows developers to focus on one section at a time.

MVC - Model View Controller

Model-View-Controller (MVC) Pattern splits the program into specific sections that handle different aspects of the application. The MVC pattern consists of three components:

1. Model: This has the the data and the business logic of your application. It encapsulates the application's data and provides methods to manipulate and access that data. You can have multiple views per model.

2. View: This is responsible for rendering the user interface elements and presenting the data to the user. It receives input events from the user and forwards them to the controller for handling.

3. Controller: This component acts as an intermediary between the model and the view. It handles user input events from the view, updates the model accordingly, and updates the view to reflect any changes in the model. It encapsulates the application's business logic and is in charge with the interactions between the model and the view.

4. Application: This is the main entry point of the application. It initializes the model, view, and controller, and connects them together. It also handles the application's lifecycle events, such as starting up, shutting down, and restarting.

The controller calls both the methods of the model and the view. The model notifies through the controller about changes, and the view fetches the updated data from the model. When the user inputs something the view notifies the controller, which in turn updates the model and the view.

Example of MVC

```
// Model
public class Model {
    private String message;

    public void setMessage(String message) {
        this.message = message;
    }

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

```
// View
public class View extends Application {
    private Label messageLabel;
    private TextField inputField;
    private Button submitButton;

    public void start(Stage primaryStage) {
        primaryStage.setTitle("MVC Example");

        messageLabel = new Label();
        inputField = new TextField();
        submitButton = new Button("Submit");

        VBox vbox = new VBox(10);
        vbox.setPadding(new Insets(10));
        vbox.getChildren().addAll(messageLabel, inputField, submitButton);

        Scene scene = new Scene(vbox, 200, 150);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public void setMessage(String message) {
        messageLabel.setText(message);
    }

    public String getInputText() {
        return inputField.getText();
    }

    public void setSubmitButtonAction(EventHandler<ActionEvent>
eventHandler) {
        submitButton.setOnAction(eventHandler);
    }
}
```

```
// Controller
public class Controller {
    private Model model;
    private View view;

    public Controller(Model model, View view) {
        this.model = model;
        this.view = view;
        this.view.setSubmitButtonAction(this::handleSubmitButton);
    }

    private void handleSubmitButton(ActionEvent event) {
        String inputText = view.getInputText();
        model.setMessage(inputText);
        view.setMessage(model.getMessage());
    }
}

// Main
public class Main {
    public static void main(String[] args) {
        Model model = new Model();
        View view = new View();
        Controller controller = new Controller(model, view);
        view.start(new Stage());
    }
}
```

MVVM - Model View ViewModel

Model-View-ViewModel (MVVM) Pattern: MVVM is similar to MVC but introduces a ViewModel component, which acts as the middleman between the View and the Model. The ViewModel provides properties and methods that the View can access and interact with.

The main difference between MVC and MVVM is that the ViewModel is responsible for updating the View, instead of the Controller. This makes it easier to test the View, as it doesn't need to be mocked or stubbed out. It also makes it easier to reuse the View, as it doesn't need to be modified when the Model changes.

Example of MVVM

```
// Model
public class Model {
    private StringProperty message = new SimpleStringProperty();

    public StringProperty messageProperty() {
        return message;
    }
```

```
    public void setMessage(String message) {
        this.message.set(message);
    }

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

```
// ViewModel
public class ViewModel {
    private Model model;

    public ViewModel(Model model) {
        this.model = model;
    }

    public StringProperty messageProperty() {
        return model.messageProperty();
    }

    public void setMessage(String message) {
        model.setMessage(message);
    }
}
```

```
// View
public class View extends Application {
    private Label messageLabel;
    private TextField inputField;
    private Button submitButton;

    private ViewModel viewModel;

    public void start(Stage primaryStage) {
        primaryStage.setTitle("MVVM Example");

        messageLabel = new Label();
        inputField = new TextField();
        submitButton = new Button("Submit");

        VBox vbox = new VBox(10);
        vbox.setPadding(new Insets(10));
        vbox.getChildren().addAll(messageLabel, inputField, submitButton);

        Scene scene = new Scene(vbox, 200, 150);
        primaryStage.setScene(scene);
```

```
        primaryStage.show();

        // Bindings
        messageLabel.textProperty().bind(viewModel.messageProperty());
        inputField.textProperty().addListener((observable, oldValue,
newValue) -> viewModel.setMessage(newValue));
    }

    public void setViewModel(ViewModel viewModel) {
        this.viewModel = viewModel;
    }
}
```

```
// Main
public class Main {
    public static void main(String[] args) {
        Model model = new Model();
        ViewModel viewModel = new ViewModel(model);
        View view = new View();
        view.setViewModel(viewModel);
        view.start(new Stage());
    }
}
```