MODUL 15
MODEL VIEW PRESENTER (MVP)

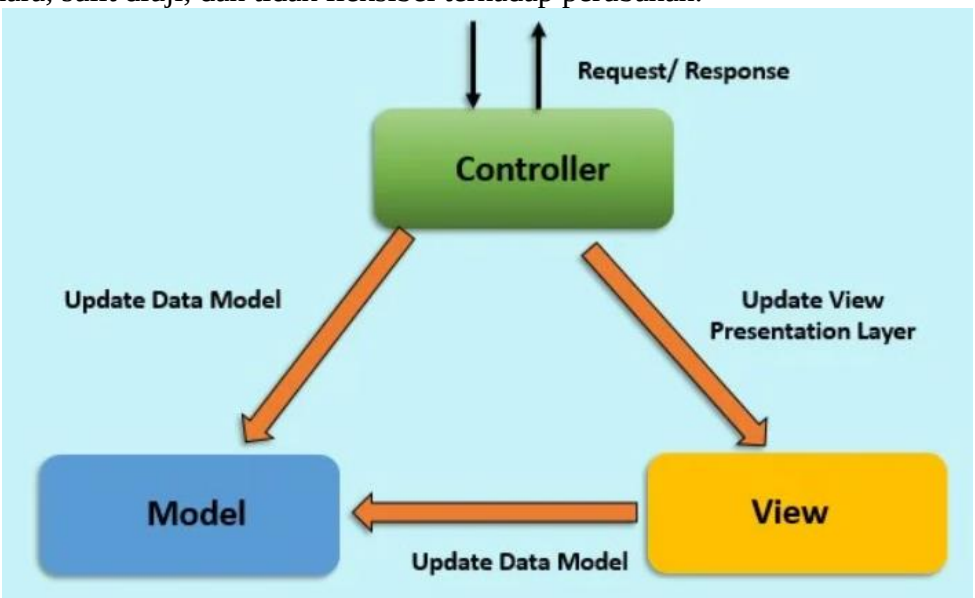
15.1 Tujuan Materi Pembelajaran

1. Menjelaskan konsep arsitektur Model-View-Presenter (MVP) beserta perbedaannya dengan arsitektur Model-View-Controller (MVC).
2. Mengidentifikasi komponen-komponen MVP dan peran masing-masing (Model, View, Presenter) dalam konteks aplikasi JavaFX.
3. Menganalisis permasalahan umum pada penggunaan MVC di JavaFX serta menjelaskan manfaat penerapan pola MVP sebagai solusinya.
4. Merancang struktur kontrak MVP (interface) antara Presenter dan View untuk memastikan keterpisahan tanggung jawab.
5. Mengimplementasikan pola MVP pada aplikasi sederhana berbasis JavaFX, termasuk pemisahan logika presentasi dari antarmuka pengguna.

15.2 Materi Pembelajaran

15.2.1. Masalah pada pola desain arsitektur Model View Controller (MVC)

Dalam pengembangan aplikasi desktop berbasis JavaFX, pengelolaan arsitektur perangkat lunak menjadi sangat krusial untuk memastikan keterpisahan tanggung jawab antar komponen, khususnya antara antarmuka pengguna dan logika bisnis. Salah satu tantangan umum yang sering ditemui pada aplikasi dengan fitur CRUD (Create, Read, Update, Delete), seperti pencatatan data mahasiswa, adalah pencampuran antara logika tampilan dan logika proses di dalam controller. Hal ini menyebabkan kode menjadi sulit dipelihara, sulit diuji, dan tidak fleksibel terhadap perubahan.



Gambar 15.2.1. Pola desain arsitektur MVC di mana kelas Controller bertanggung jawab langsung terhadap pengolahan event pengguna, manipulasi data, dan juga pengaturan tampilan

Pendekatan konvensional yang umum digunakan dalam pengembangan aplikasi JavaFX adalah Model-View-Controller (MVC), di mana kelas FXMLController bertanggung jawab langsung terhadap pengolahan event pengguna, manipulasi data, dan

juga pengaturan tampilan. Meskipun arsitektur ini terlihat terstruktur secara teoretis, dalam praktiknya sering kali controller menjadi sangat kompleks karena memuat semua tanggung jawab tersebut. Ketika aplikasi berkembang, perubahan kecil dalam tampilan atau data dapat memicu kebutuhan untuk mengubah banyak bagian kode, yang menghambat produktivitas dan meningkatkan risiko kesalahan.

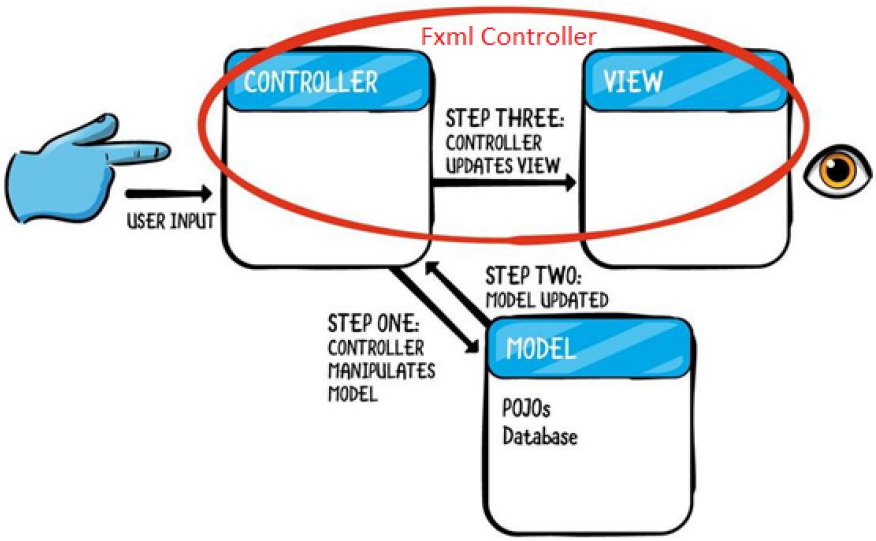
Dalam JavaFX, antarmuka pengguna (View) umumnya didefinisikan menggunakan file .fxml yang bersifat statis. Saat sebuah Controller mencoba memperbarui tampilan sebagai respons terhadap perubahan model atau aksi pengguna, fleksibilitas menjadi terbatas karena file .fxml tidak dirancang untuk berubah secara dinamis di waktu eksekusi. Dengan kata lain, jika elemen tampilan perlu dimodifikasi berdasarkan kondisi tertentu, pendekatan ini tidak memadai tanpa penambahan kode imperatif yang rumit dalam controller.

Secara arsitektural, dalam pola Model-View-Controller (MVC) yang ideal:

- Model bertanggung jawab atas data dan logika bisnis,
- View menampilkan data kepada pengguna,
- Controller bertindak sebagai penghubung, menangani input dan meneruskannya ke Model atau View.

Tidak semua antarmuka JavaFX dibangun dengan .fxml. Dalam beberapa kasus, layout perlu dibangun secara dinamis melalui kode Java (misalnya tergantung dari hak akses pengguna, data runtime, atau plugin sistem). Dalam situasi ini, controller bukan hanya mengatur alur logika, tetapi juga menyusun tampilan, yang semakin menegaskan bahwa controller sedang memainkan dua peran sekaligus: sebagai pengatur alur (Controller) dan sebagai pembangun tampilan (View).

Namun, pada praktik JavaFX, controller view hampir selalu berisi logika presentasi langsung, seperti manipulasi TableView, ListView, pemformatan data tampilan, atau pengaturan gaya elemen. Hal ini menyebabkan Controller mengambil alih sebagian atau seluruh peran View, yang merupakan pelanggaran prinsip separation of concerns. Akibatnya, kode dalam controller menjadi besar dan tidak terstruktur (fat controller), serta sulit untuk dipelihara.



Gambar 15.2.2. Pada desain pola arsitektur MVC, Controller view hampir selalu harus berisi logika untuk view sebagai respons terhadap inputan pengguna, yang melanggar prinsip controller.

Tujuan utama MVC adalah modularitas, keterpisahan tanggung jawab, dan kemudahan pengujian. Namun, ketika controller mengelola logika UI dan logika alur secara langsung:

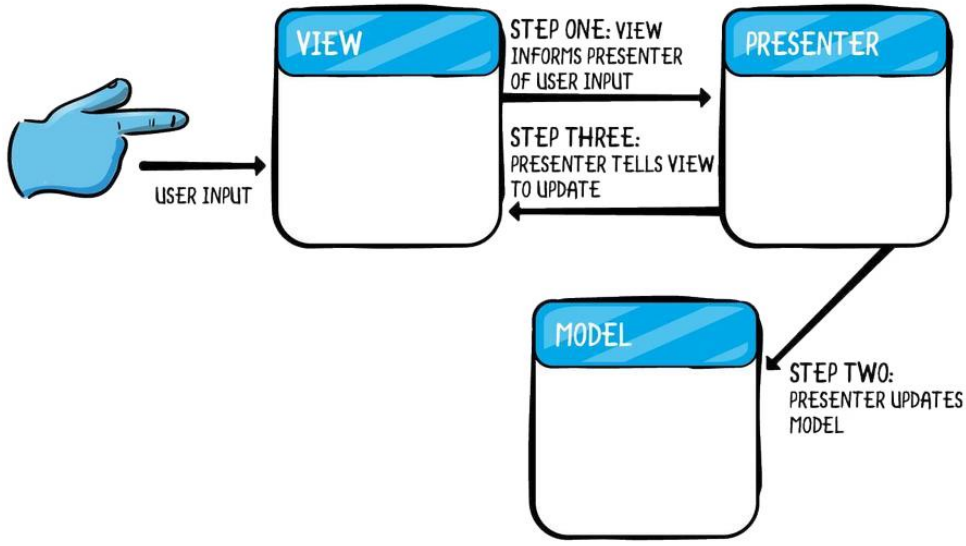
- MVC tidak lagi menawarkan manfaat pemisahan tanggung jawab yang dijanjikan.
- Modifikasi tampilan berdampak langsung terhadap controller, dan sebaliknya.
- Arsitektur menjadi rapuh dan sangat terikat (*tight coupling*).

Salah satu nilai jual utama dari MVC adalah kemudahan dalam melakukan unit testing terhadap logika aplikasi. Namun, karena controller berisi kode manipulasi UI dan logika aplikasi sekaligus, proses pengujian menjadi rumit:

- Tidak dapat menguji logika controller tanpa melibatkan JavaFX runtime.
- Tidak bisa mensimulasikan View secara terpisah, karena View terikat langsung dengan controller.
- Dependency yang sulit di-mock membuat pengujian unit tidak praktis tanpa framework khusus.

15.2.2. Pengenalan Pola Desain Arsitektur Model View Presenter (MVP)

Model-View-Presenter (MVP) hadir sebagai solusi arsitektural yang memisahkan lebih tegas antara logika presentasi dan antarmuka pengguna. Dalam pola MVP, View hanya bertanggung jawab menampilkan data dan menerima input pengguna, sementara Presenter bertanggung jawab sepenuhnya terhadap logika interaksi, dan Model menyimpan serta memproses data. Komunikasi antara View dan Presenter dilakukan melalui antarmuka kontrak (*contract interface*), sehingga tidak ada ketergantungan langsung yang kuat antar komponen.



Gambar 15.2.3. Pada pola MVP, View tidak diperbolehkan berkomunikasi langsung dengan Model.

Pada pola MVP, View tidak diperbolehkan berkomunikasi langsung dengan Model. Sebagai gantinya, semua interaksi harus melewati Presenter, yang berperan sebagai mediator atau perantara. Hal ini memutus ketergantungan langsung antara UI dan logika bisnis, dan menjadikan masing-masing komponen lebih mandiri dan dapat diuji secara terpisah. Adapun karakterisitik yang ada pada pola MVP adalah sebagai berikut :

1. Presenter bertindak sebagai pusat koordinasi logika interaksi, bukan hanya mengarahkan alur data, tetapi juga bertugas memproses respons terhadap input

pengguna, melakukan validasi, dan meneruskan permintaan ke Model. Presenter akan menerima notifikasi dari View, lalu memanggil Model sesuai kebutuhan, dan selanjutnya mengupdate View dengan hasil yang diperoleh. Dengan demikian, logika aplikasi tidak tersebar di View, tetapi terkonsentrasi di Presenter yang lebih mudah diuji.

2. Dalam pola MVP, View (biasanya controller dalam JavaFX/FXML) hanya bertugas:
   - Menangkap interaksi pengguna (seperti klik tombol).
   - Meneruskan peristiwa tersebut ke Presenter melalui kontrak yang telah disepakati.
   - Menampilkan data sesuai instruksi dari Presenter.
3. View menjadi "pasif" dan tidak mengetahui keberadaan Model secara langsung. Ini memungkinkan View diganti, diuji, atau dimodifikasi tanpa mengganggu logika aplikasi yang sebenarnya.
4. Model tetap mempertahankan fungsinya sebagai lapisan data dan logika bisnis. Dalam MVP, model tidak tahu-menahu soal View atau Presenter, sehingga benar-benar terpisah dari antarmuka pengguna. Ini menjamin modularitas dan meningkatkan reusabilitas logika bisnis.
5. Untuk memastikan struktur komunikasi tetap terorganisasi dan tidak saling bergantung langsung, MVP menggunakan interface (Contract) yang mendefinisikan metode yang disediakan dan dibutuhkan oleh View dan Presenter:
   - ViewContract berisi metode yang dapat dipanggil Presenter untuk menampilkan data.
   - PresenterContract berisi metode yang harus dipanggil View untuk mengirimkan event pengguna. Dengan cara ini, setiap pihak hanya mengetahui kontraknya masing-masing dan tidak perlu bergantung pada implementasi konkret.

Sebagai bentuk implementasi nyata, dalam aplikasi pencatatan data mahasiswa, penerapan MVP dapat dilakukan dengan memisahkan logika seperti penambahan mahasiswa, validasi input, dan pengambilan data dari database ke dalam Presenter, sementara View (controller) hanya bertugas menangani event tombol dan memperbarui tampilan. Dengan demikian, Presenter dapat diuji secara unit tanpa memerlukan tampilan grafis, dan aplikasi menjadi lebih modular dan mudah dikembangkan.

1. Definisikan Kontrak antara View dan Presenter

Mendeskripsikan antarmuka komunikasi satu arah antara tampilan dan logika presentasi.

```
public interface MahasiswaContract {
    interface View {
        void tampilkanDaftarMahasiswa(List<Mahasiswa> daftar);
        void tampilkanPesan(String pesan);
    }

    interface Presenter {
        void ambilSemuaMahasiswa();
        void simpanMahasiswa(Mahasiswa m);
    }
}
```

2. Implementasi Presenter

Mengatur alur aplikasi dan mengakses data dari Model.

```
public class MahasiswaPresenter implements
MahasiswaContract.Presenter {
    private final MahasiswaContract.View view;
    private final MahasiswaDAO dao;

    public MahasiswaPresenter(MahasiswaContract.View view) {
        this.view = view;
        this.dao = new MahasiswaDAO();
    }

    public void ambilSemuaMahasiswa() {
        List<Mahasiswa> daftar = dao.getAll();
        view.tampilkanDaftarMahasiswa(daftar);
    }

    public void simpanMahasiswa(Mahasiswa m) {
        dao.insert(m);
        view.tampilkanPesan("Mahasiswa berhasil disimpan.");
        ambilSemuaMahasiswa();
    }
}
```

3. View sebagai FXML Controller

Menghubungkan elemen tampilan dengan logika Presenter tanpa mengatur proses data secara langsung.

```
public class MahasiswaController implements Initializable,
MahasiswaContract.View {
    @FXML private TableView<Mahasiswa> tableView;
    private MahasiswaContract.Presenter presenter;

    public void initialize(URL location, ResourceBundle resources)
{
        presenter = new MahasiswaPresenter(this);
        presenter.ambilSemuaMahasiswa();
    }

    public void onTambahClicked() {
        Mahasiswa m = new Mahasiswa("Ani", "20231001", 3.7);
        presenter.simpanMahasiswa(m);
    }

    public void tampilkanDaftarMahasiswa(List<Mahasiswa> daftar) {
        tableView.setItems(FXCollections.observableList(daftar));
    }

    public void tampilkanPesan(String pesan) {
```

```
        new Alert(Alert.AlertType.INFORMATION, pesan).show();
    }
}
```

Dengan menggunakan pola MVP:

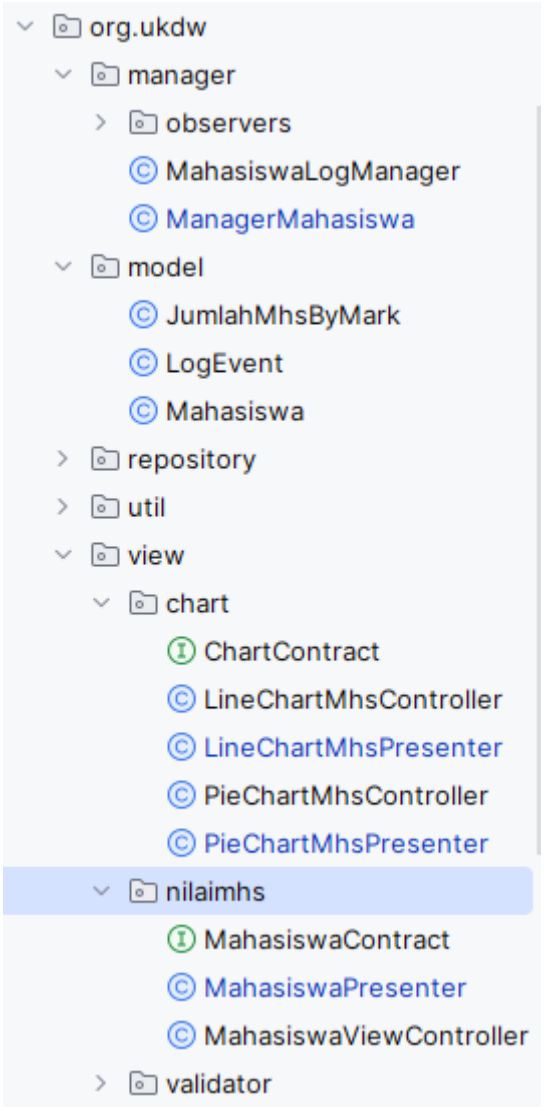
- View tidak tahu-menahu tentang bagaimana data diolah.
- Presenter tidak perlu tahu detail implementasi tampilan.
- Unit testing dapat difokuskan pada Presenter, tanpa perlu membuat tampilan JavaFX.

15.3 Rangkuman Materi

1. Model-View-Presenter (MVP) adalah pola arsitektur perangkat lunak yang dirancang untuk memisahkan logika antarmuka pengguna (UI) dari logika aplikasi (business logic). MVP terdiri dari tiga komponen utama:
   a. **Model:** Mengelola data dan logika bisnis.
   b. **View:** Menangani tampilan antarmuka dan menerima input pengguna.
   c. **Presenter:** Menjadi perantara yang menghubungkan View dan Model, bertanggung jawab atas alur logika interaksi.
2. Dalam JavaFX, pendekatan tradisional Model-View-Controller (MVC) sering kali menempatkan logika tampilan dan logika bisnis dalam satu entitas: controller. Hal ini menyebabkan:
   a. Kode menjadi tidak modular dan sulit diuji.
   b. Ketergantungan yang tinggi antar komponen.
   c. Pelanggaran prinsip pemisahan tanggung jawab (separation of concerns).
3. MVP menyelesaikan permasalahan tersebut dengan mengisolasi masing-masing komponen melalui kontrak (interface), sehingga:
   a. View tidak berkomunikasi langsung dengan Model.
   b. Presenter memegang seluruh logika interaksi dan dapat diuji tanpa melibatkan JavaFX.
   c. Model tetap bersih, tidak tergantung pada antarmuka pengguna.
4. MVP mendukung pengembangan aplikasi yang:
   a. Lebih modular, karena setiap komponen memiliki tanggung jawab spesifik.
   b. Lebih mudah diuji, karena Presenter dapat diuji tanpa UI.
   c. Lebih fleksibel, karena View dapat diganti tanpa memengaruhi logika bisnis.
   d. Lebih mudah dirawat dan dikembangkan, karena perubahan pada satu bagian tidak merusak bagian lain.
5. Dalam konteks aplikasi pencatatan data mahasiswa, penerapan MVP memungkinkan proses seperti penambahan, pembaruan, dan penghapusan data dilakukan melalui Presenter yang mengatur komunikasi antara tampilan (FXMLController) dan penyimpanan data (Model/DAO), tanpa saling bergantung secara langsung.

15.4 Latihan

Anda diminta untuk melakukan refactoring terhadap aplikasi JavaFX pencatatan data mahasiswa yang sebelumnya menggunakan pendekatan tradisional FXMLController yang langsung terhubung dengan model data. Refactoring dilakukan agar menerapkan pola desain Model-View-Presenter (MVP). Setelah refactoring, struktur package diharapkan sebagai berikut:



1. Buat dan implementasikan MahasiswaContract.java seperti berikut ini :

```
interface MahasiswaContract {
    interface View {
        void showLoading();
        void hideLoading();
        void displayAllNilaiMahasiswa(List<Mahasiswa> result);
        void showError(String errorMessage);
        void onSuccess(String message);
        ObservableList<Mahasiswa> getObservableList();
    }

    interface Presenter {
        void getAllNilaiMahasiswa();
        void updateNilaiMahasiswa(Mahasiswa mahasiswaOld,
Mahasiswa mahasiswaNew);
        void addNilaiMahasiswa(Mahasiswa mahasiswa);
        void deleteNilaiMahasiswa(Mahasiswa selectedItem);
    }
}
```

Penjelasan :

- **View** : Diimplementasikan oleh MahasiswaViewController. Menangani tampilan UI.
- **displayAllNilaiMahasiswa (...)** : Untuk me-refresh daftar data.
- Method **onSuccess()** atau **showError()** : Menampilkan pesan status berhasil atau gagal ke pengguna.
- **Presenter** : Diimplementasikan oleh MahasiswaPresenter. Menyimpan dan memuat data dari model, serta mengatur kapan View diubah.

2. Implementasikan interface MahasiswaContract.Presenter pada class MahasiswaPresenter.
    - Ambil data dari MahasiswaDAO.
    - Panggil metode di View jika perlu menampilkan daftar atau pesan.
    - Hindari penggunaan langsung ke komponen JavaFX.
3. Refactor MahasiswaViewController.java
    - Implementasikan MahasiswaContract.View.
    - Delegasikan seluruh logika akses ke data ke MahasiswaPresenter.
    - Jangan menyentuh objek MahasiswaDAO atau logika validasi di controller.
4. Refactor semua class view controller pada program Anda agar mengikuti pola desain MVP.

15.5 Kunci Jawaban

File MahasiswaContract.java berfungsi sebagai kontrak (interface) komunikasi antara komponen View dan Presenter. Kontrak ini mendefinisikan tanggung jawab yang harus diimplementasikan oleh kedua belah pihak agar tidak saling bergantung secara langsung. MahasiswaContract.View mendefinisikan fungsi-fungsi yang harus disediakan oleh kelas View. MahasiswaContract.Presenter mendefinisikan fungsi yang dijalankan presenter untuk menangani logika bisnis dan memproses data dari model. Implementasi kedua interface tersebut dilakukan pada class MahasiswaViewController dan MahasiswaPresenter.

MahasiswaVewController.java

```
public class MahasiswaViewController implements Initializable,
MahasiswaContract.View {

    @FXML
    public TableView<Mahasiswa> tblView;
    @FXML
    private TableColumn<Mahasiswa, String> colNIM;
    @FXML
    private TableColumn<Mahasiswa, String> colNama;
    @FXML
    private TableColumn<Mahasiswa, Double> colNilai;
    @FXML
    private TextField txtNim;
    @FXML
    private TextField txtNama;
    @FXML
    private TextField txtNilai;
    @FXML
    public TextField txtNamaCari;
    @FXML
```

```
    private Label lblInfo;
    @FXML
    private BarChart<String, Number> barChart;
    @FXML
    private ImageView imgFoto;
    private byte[] selectedMahasiswaFotoBlob;
    private FilteredList<Mahasiswa> mahasiswaFilteredList;
    private Mahasiswa selectedMahasiswa;
    private MahasiswaContract.Presenter presenter;
    LoadingComponent loadingComponent;

    @Override
    public void initialize(URL url, ResourceBundle resourceBundle) {
        presenter = new MahasiswaPresenter(this);

        this.loadingComponent = new
LoadingComponent(RegistrasiMahasiswa.getPrimaryStage());
        this.mahasiswaFilteredList = new
FilteredList<>(FXCollections.observableArrayList(), p -> true); // ←
filtered list dari master


tblView.getSelectionModel().selectedItemProperty().addListener((obs,
oldVal, newVal) -> {
            if (newVal != null) {
                selectedMahasiswa = newVal;
                txtNim.setText(newVal.getNim());
                txtNama.setText(newVal.getNama());
                txtNilai.setText(String.valueOf(newVal.getNilai()));
                txtNim.setDisable(true); // NIM tidak boleh diubah
saat edit
                selectedMahasiswaFotoBlob = newVal.getFoto();

imgFoto.setImage(convertBytesToImage(selectedMahasiswaFotoBlob));
            }
        });

        SortedList<Mahasiswa> sortedData = new
SortedList<>(mahasiswaFilteredList);

sortedData.comparatorProperty().bind(tblView.comparatorProperty());
        tblView.setItems(sortedData);

        txtNamaCari.textProperty().addListener((
                (obs, oldValue, newValue) ->
                {

mahasiswaFilteredList.setPredicate(createPredicate(newValue));
                }
        ));

        // Display the image in imageView
        Image image = new
Image(Objects.requireNonNull(RegistrasiMahasiswa.class.getResourceAsS
tream("default_img.png")));
        imgFoto.setImage(image);

        displayList();
    }

    private Predicate<? super Mahasiswa> createPredicate(String
```

```
searchText) {
        return order -> {
            if (searchText == null || searchText.isEmpty()) return
true;
            return searchFindsMahasiswa(order, searchText);
        };
    }

    private boolean searchFindsMahasiswa(Mahasiswa mahasiswa, String
searchText) {
        return
(mahasiswa.getNim().toLowerCase().contains(searchText.toLowerCase()))
||

(mahasiswa.getNama().toLowerCase().contains(searchText.toLowerCase())
);
    }

    private void displayList() {
        tblView.setEditable(false);
        colNIM.setCellValueFactory(new
PropertyValueFactory<>("nim"));
        colNama.setCellValueFactory(new
PropertyValueFactory<>("nama"));
        colNilai.setCellValueFactory(new
PropertyValueFactory<>("nilai"));
        presenter.getAllNilaiMahasiswa();
        updateInfo();
    }

    private void updateInfo() {
        double rata = 0;
        for (Mahasiswa m : getObservableList()) {
            rata += m.getNilai();
        }
        rata = ! getObservableList().isEmpty() ? rata /
getObservableList().size() : 0;
        lblInfo.setText("Jumlah data: " +  getObservableList().size()
+ ", Rata nilai: " + rata);
    }

    private void bersihkan() {
        txtNim.clear();
        txtNama.clear();
        txtNilai.clear();
        txtNim.setDisable(false);
        // Display the image in imageView
        Image image = new
Image(Objects.requireNonNull(RegistrasiMahasiswa.class.getResourceAsS
tream("default_img.png")));
        imgFoto.setImage(image);
        tblView.getSelectionModel().clearSelection();
        selectedMahasiswa = null;
        updateInfo();
    }

    @FXML
    public void onBtnAddClick(ActionEvent actionEvent) {
        Mahasiswa newMahasiswa = new Mahasiswa(txtNim.getText(),
txtNama.getText(),
                Double.parseDouble(txtNilai.getText()),
```

```
selectedMahasiswaFotoBlob);

        // Ambil daftar NIM dari masterData
        List<String> nimTerdaftar = getObservableList().stream()
                .map(Mahasiswa::getNim)
                .toList();
        InputMahasiswaValidator validator = new
EmptyFieldMahasiswaValidator();
        validator.setNext(new
NIMUniqueMahasiswaValidator(nimTerdaftar))
                .setNext(new IPKRangeMahasiswaValidator());
        if (!validator.validate(newMahasiswa)) {
            String pesanError = validator.getLastErrorMessage();
            Alert alert = new Alert(Alert.AlertType.ERROR);
            alert.setTitle("Validasi Gagal");
            alert.setHeaderText("Data mahasiswa tidak valid");
            alert.setContentText(pesanError != null ? pesanError :
"Periksa kembali data yang Anda masukkan.");
            alert.show();
            return;
        }

        presenter.addNilaiMahasiswa(newMahasiswa);
    }

    @FXML
    public void onBtnSimpanClick(ActionEvent actionEvent) {
        if (selectedMahasiswa == null) {
            new Alert(Alert.AlertType.WARNING, "Pilih data dari tabel
untuk diperbarui.").show();
            return;
        }

        Mahasiswa updatedMahasiswa = new Mahasiswa(txtNim.getText(),
txtNama.getText(),
                Double.parseDouble(txtNilai.getText()),
selectedMahasiswaFotoBlob);

        //proses validasi
        InputMahasiswaValidator validator = new
EmptyFieldMahasiswaValidator();
        validator.setNext(new IPKRangeMahasiswaValidator());
        if (!validator.validate(updatedMahasiswa)) {
            String pesanError = validator.getLastErrorMessage();

            Alert alert = new Alert(Alert.AlertType.ERROR);
            alert.setTitle("Validasi Gagal");
            alert.setHeaderText("Data mahasiswa tidak valid");
            alert.setContentText(pesanError != null ? pesanError :
"Periksa kembali data yang Anda masukkan.");
            alert.show();
            return;
        }
        presenter.updateNilaiMahasiswa(selectedMahasiswa,
updatedMahasiswa);
    }

    @FXML
    public void onBtnHapusClick(ActionEvent actionEvent) {
        if (selectedMahasiswa == null) {
            new Alert(Alert.AlertType.WARNING, "Pilih data dari tabel
```

```
untuk dihapus.").show();
                return;
            }
            presenter.deleteNilaiMahasiswa(selectedMahasiswa);
        }

        @FXML
        protected void onBtnPilihFotoClick() {
            FileChooser fileChooser = new FileChooser();
            fileChooser.setTitle("Pilih Foto Mahasiswa");
            fileChooser.getExtensionFilters().add(
                    new FileChooser.ExtensionFilter("Image Files",
"*.jpg", "*.jpeg", "*.png")
            );
            File file =
fileChooser.showOpenDialog(imgFoto.getScene().getWindow());
            if (file != null) {
                selectedMahasiswaFotoBlob = resizeImageToBytes(file, 100,
150);
                imgFoto.setImage(new Image(file.toURI().toString()));
            }
        }

        private byte[] resizeImageToBytes(File file, int width, int
height) {
            try {
                BufferedImage originalImage = ImageIO.read(file);
                BufferedImage resizedImage = new BufferedImage(width,
height, BufferedImage.TYPE_INT_ARGB);
                Graphics2D g2d = resizedImage.createGraphics();
                g2d.drawImage(originalImage.getScaledInstance(width,
height, java.awt.Image.SCALE_SMOOTH), 0, 0, null);
                g2d.dispose();
                ByteArrayOutputStream baos = new ByteArrayOutputStream();
                ImageIO.write(resizedImage, "png", baos);
                return baos.toByteArray();
            } catch (IOException e) {
                e.printStackTrace();
                return null;
            }
        }

        private Image convertBytesToImage(byte[] bytes) {
            if (bytes == null) return null;
            return new Image(new ByteArrayInputStream(bytes));
        }

        @FXML
        public void onLineChartCLicked() {
            RegistrasiMahasiswa.openViewWithModal("line-chart-view",
false);
        }

        @FXML
        public void onPieChartCLicked(ActionEvent actionEvent) {
            RegistrasiMahasiswa.openViewWithModal("pie-chart-view",
false);
        }

        public void onActionLogout(ActionEvent actionEvent) {
            // Create a new alert with type Confirmation
```

```
        Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
        alert.setTitle("Exit & Logout Confirmation");
        alert.setHeaderText("Are you sure you want to exit?");
        alert.setContentText("Press OK to exit the application.");

        // Add Yes and No buttons to the alert
        alert.getButtonTypes().setAll(ButtonType.YES, ButtonType.NO);
        // Show the alert and wait for user response
        alert.showAndWait().ifPresent(response -> {
            if (response == ButtonType.YES) {
                // User clicked Yes, exit the application
                SessionManager.getInstance().logout();
                RegistrasiMahasiswa.setRoot("login-view", false);
            }
        });
    }

    public void onActionAbout(ActionEvent actionEvent) {
        RegistrasiMahasiswa.openViewWithModal("about-view", false);
    }

    public void onActionShowLog(ActionEvent actionEvent) {
        RegistrasiMahasiswa.openViewWithModal("log-view", false);
    }

    @Override
    public void showLoading() {
        loadingComponent.show();
        System.out.println("Loading!!!!");
    }

    @Override
    public void hideLoading() {
        loadingComponent.hide();
        System.out.println("Loading Hide!!!!");
    }

    @Override
    public void displayAllNilaiMahasiswa(List<Mahasiswa> result) {

    }

    @Override
    public void showError(String errorMessage) {
        Alert alert = new Alert(Alert.AlertType.ERROR, errorMessage);
        alert.show();
        bersihkan();
    }

    @Override
    public void onSuccess(String message) {
        Alert alert = new Alert(Alert.AlertType.INFORMATION,
message);
        alert.show();
        bersihkan();
    }

    @Override
    public ObservableList<Mahasiswa> getObservableList() {
        return (ObservableList<Mahasiswa>)
mahasiswaFilteredList.getSource();
```

```
    }
}
```

MahasiswaPresenter.java

```
public class MahasiswaPresenter implements
MahasiswaContract.Presenter {
    private final MahasiswaContract.View view;
    private final ManagerMahasiswa manager;

    public MahasiswaPresenter(MahasiswaContract.View view) {
        this.view = view;
        this.manager = new ManagerMahasiswa();
        this.manager.addObserver(new MahasiswaLogManager());
    }

    @Override
    public void getAllNilaiMahasiswa() {
        view.showLoading();
        List<Mahasiswa> result = manager.getAllMahasiswa();
        if (!result.isEmpty()) {
            view.getObservableList().clear();
            view.getObservableList().addAll(result);
        } else {
            view.showError("Gagal memuat data Mahasiswa!");
        }
        view.hideLoading();
    }

    @Override
    public void updateNilaiMahasiswa(Mahasiswa mahasiswaOld,
Mahasiswa mahasiswaNew) {
        view.showLoading();
        if (manager.updateMahasiswa(mahasiswaNew)) {
            int iOldCatatan =
view.getObservableList().indexOf(mahasiswaOld);
            view.getObservableList().set(iOldCatatan, mahasiswaNew);
            view.onSuccess("Mahasiswa berhasil diupdate");
        } else {
            view.showError("Mahasiswa gagal diupdate");
        }
        view.hideLoading();
    }

    @Override
    public void addNilaiMahasiswa(Mahasiswa mahasiswa) {
        view.showLoading();
        if (manager.tambahMahasiswa(mahasiswa)) {
            view.getObservableList().add(mahasiswa);
            view.onSuccess("Data Mahasiswa Ditambahkan!");
        } else {
            view.showError("Data  Mahasiswa gagal Ditambahkan!");
        }
        view.hideLoading();
    }

    @Override
    public void deleteNilaiMahasiswa(Mahasiswa mahasiswa) {
        view.showLoading();
        // do your GUI stuff here
        if (manager.hapusMahasiswa(mahasiswa.getNim())) {
```

```
            view.onSuccess("Mahasiswa berhasil dihapus!");
            view.getObservableList().remove(mahasiswa);
        } else {
            view.showError("Mahasiswa gagal dihapus!");
        }
        view.hideLoading();
    }
}
```

Loading-view.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.control.ProgressIndicator?>
<?import javafx.scene.layout.VBox?>
<VBox xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
      alignment="CENTER" spacing="10" prefWidth="200"
prefHeight="100"
      fx:id="loadingRoot" style="-fx-background-color: white; -fx-
padding: 20;">

    <ProgressIndicator fx:id="progressIndicator" />
    <Label fx:id="loadingLabel" text="Loading..." />
</VBox>
```

LoadingComponent.java

```
public class LoadingController {
    private Stage loadingStage;

    public LoadingController(Stage primaryStage) {
        try {
            FXMLLoader fxmlLoader = new
FXMLLoader(RegistrasiMahasiswa.class
                    .getResource("loading-view.fxml"));
            Parent root = fxmlLoader.load();

            loadingStage = new Stage();
            loadingStage.setScene(new Scene(root));
            loadingStage.initModality(Modality.WINDOW_MODAL);
            loadingStage.setResizable(false);
            loadingStage.setTitle("Please Wait...");
            loadingStage.setOnCloseRequest(Event::consume); //
prevent manual close
            loadingStage.initOwner(primaryStage);
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

    public void show() {
        loadingStage.show();
    }

    public void hide() {
        loadingStage.close();
    }
}
```