MODUL 14
OBSERVER & CHAIN OF RESPONSIBILITY PATERN

14.1 Tujuan Materi Pembelejaran

1. Mengidentifikasi permasalahan yang sesuai untuk diselesaikan dengan Observer Pattern dan Chain of Responsibility Pattern.
2. Mendeskripsikan struktur dan komponen utama dari Observer Pattern dan Chain of Responsibility Pattern, termasuk hubungan antar objeknya.
3. Menganalisis kelebihan dan kekurangan dari masing-masing pattern dalam konteks desain perangkat lunak yang fleksibel dan modular.
4. Mengimplementasikan Observer Pattern untuk memisahkan logika pencatatan atau reaksi terhadap perubahan status objek.
5. Mengimplementasikan Chain of Responsibility Pattern untuk menyusun alur validasi atau proses berlapis secara terstruktur.
6. Menerapkan kedua pola desain tersebut dalam proyek aplikasi sederhana, dengan menunjukkan penerapan prinsip separation of concerns dan loose coupling.
7. Mengevaluasi hasil implementasi berdasarkan prinsip desain perangkat lunak yang baik: keterbacaan, keterpisahan tanggung jawab, dan kemudahan pemeliharaan.

14.2 Materi Pembelajaran

14.4.1. Observer Patern

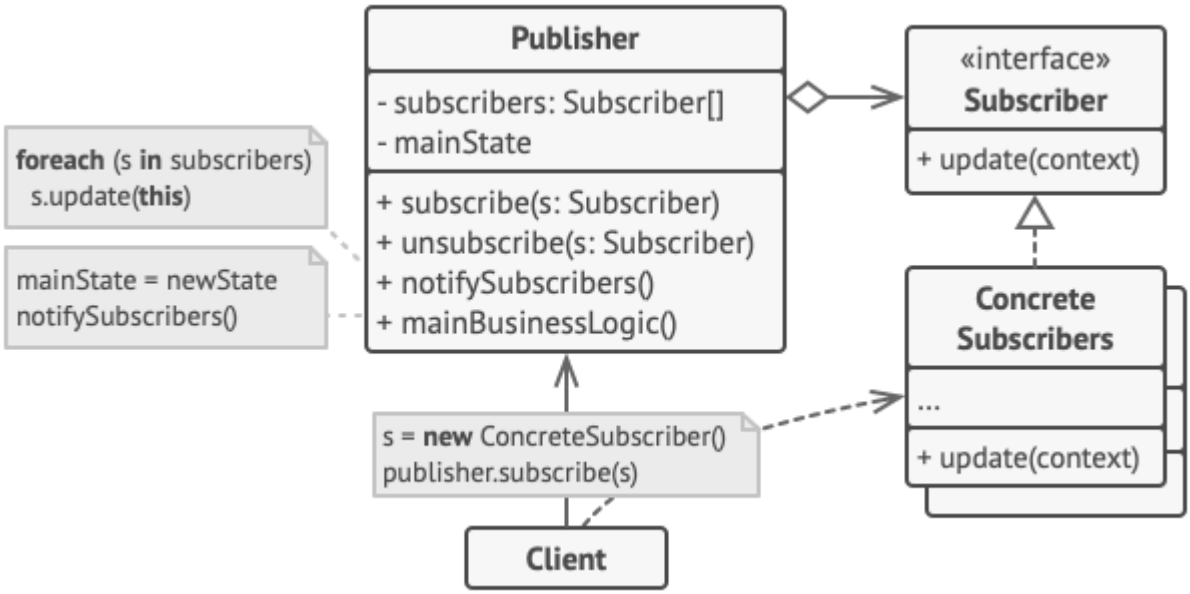
Dalam pengembangan aplikasi modern, aspek pemantauan dan pencatatan aktivitas sistem (sistem logging) merupakan elemen penting untuk mendukung keterlacakan (*traceability*), keamanan data, dan proses debugging. Salah satu kebutuhan umum dalam sistem yang melakukan perekaman data adalah kemampuan untuk mencatat setiap perubahan data yang terjadi, misalnya ketika pengguna menambahkan, mengubah, atau menghapus data. Di tengah tuntutan pengembangan perangkat lunak yang lebih modular dan terstruktur, muncul kebutuhan untuk menggunakan pendekatan desain yang memungkinkan logging dilakukan secara otomatis dan terpisah dari logika utama aplikasi.

Secara umum, pendekatan yang lazim digunakan untuk mencatat perubahan data mahasiswa adalah dengan menambahkan perintah `System.out.println()` atau pemanggilan eksplisit ke metode `log()` di setiap aksi pengguna seperti `tambahData()`, `hapusData()`, dan `updateData()`. Walaupun pendekatan ini mudah diimplementasikan, namun ia sangat terikat pada struktur program dan memerlukan pemanggilan manual di berbagai bagian kode. Hal ini menghasilkan *tight coupling* antara modul utama dan sistem logging, sehingga memperbesar risiko *code duplication*, kesalahan pencatatan, serta menyulitkan proses perawatan dan pengembangan lanjutan.

Evaluasi terhadap pendekatan konvensional tersebut menunjukkan adanya kelebihan dari segi kemudahan dan kecepatan implementasi awal, namun memiliki banyak kelemahan pada jangka panjang. Ketergantungan langsung antara logika utama dan pencatatan menyebabkan sulitnya melakukan refactoring dan pengujian unit secara terpisah. Selain itu, saat sistem berkembang dan semakin kompleks, developer cenderung lupa menyisipkan log

ke semua titik perubahan data, sehingga mengakibatkan logging yang tidak konsisten dan tidak komprehensif. Oleh karena itu, penting untuk mencari solusi yang memungkinkan sistem logging bekerja secara pasif dan *de-coupled* dari proses utama aplikasi.

Observer Pattern adalah salah satu pola desain perilaku (*behavioral design pattern*) yang memungkinkan sebuah objek (*subject*) secara otomatis memberi tahu objek-objek lainnya (*observers*) saat terjadi perubahan keadaan (*state*) (Sarcar, 2022). Dalam konteks aplikasi pencatatan data mahasiswa, objek yang bertindak sebagai *subject* adalah model data mahasiswa, sementara *observer* adalah komponen yang mencatat aktivitas (*logger*). Dengan demikian, ketika terjadi perubahan data, *observer* akan secara otomatis menerima notifikasi dan mencatat peristiwa tersebut tanpa perlu dipanggil secara eksplisit oleh bagian lain dalam sistem. Penggunaan Observer Pattern memberikan mekanisme yang rapi dan terstruktur di mana model mahasiswa hanya perlu memanggil satu fungsi `notifyObservers()`, dan semua komponen pencatat akan merespon secara otomatis. Dengan pendekatan ini, sistem dapat dengan mudah menambahkan logger tambahan (misalnya ke file, database, atau remote monitoring) tanpa perlu mengubah logika utama aplikasi.



Sebagai contoh penerapannya adalah penerapan Observer Pattern dalam konteks pencatatan data mahasiswa. Adapun tahapan-tahapannya adalah sebagai berikut :

1. **Definisikan Interface Observer**

   Membuat kontrak umum yang harus diikuti oleh semua kelas yang ingin menjadi observer. Antarmuka ini menyediakan method yang akan dipanggil saat perubahan terjadi di objek yang diamati.

```
public interface MahasiswaObserver {
    void onMahasiswaChanged(String message);
}
```

   Metode onMahasiswaChanged adalah callback yang akan dijalankan saat model mahasiswa mengalami perubahan. Parameter message digunakan untuk meneruskan informasi perubahan yang terjadi.

2. **Buat Kelas Observer Konkret**

Mengimplementasikan antarmuka MahasiswaObserver untuk mencatat perubahan ke konsol. Kelas ini akan menerima notifikasi dari MahasiswaModel.

```
public class ConsoleLogger implements MahasiswaObserver {
    @Override
    public void onMahasiswaChanged(String message) {
        System.out.println("[LOG] " + message);
    }
}
```

Kelas ini akan menerima informasi saat data mahasiswa berubah, dan langsung mencetak log-nya ke konsol.

3. **Implementasikan Kelas Subject MahasiswaModel**

Sebagai pusat perubahan data, MahasiswaModel bertindak sebagai Subject yang dapat didaftarkan Observer. Saat data mahasiswa berubah, MahasiswaModel akan memberi tahu semua observer yang telah terdaftar.

```
public class MahasiswaModel {
    private List<MahasiswaObserver> observers = new ArrayList<>();
    private List<Mahasiswa> dataMahasiswa = new ArrayList<>();

    public void addObserver(MahasiswaObserver observer) {
        observers.add(observer);
    }

    private void notifyObservers(String message) {
        for (MahasiswaObserver observer : observers) {
            observer.onMahasiswaChanged(message);
        }
    }

    public void addMahasiswa(Mahasiswa m) {
        dataMahasiswa.add(m);
        notifyObservers("Mahasiswa ditambahkan: " + m.getNama());
    }

    public void deleteMahasiswa(Mahasiswa m) {
        dataMahasiswa.remove(m);
        notifyObservers("Mahasiswa dihapus: " + m.getNama());
    }

    public void updateMahasiswa(Mahasiswa m) {
        notifyObservers("Mahasiswa diupdate: " + m.getNama());
    }
}
```

- addObserver() untuk mendaftarkan observer.

- notifyObservers() dipanggil setiap kali data berubah.
- addMahasiswa(), deleteMahasiswa(), dan updateMahasiswa() adalah method yang akan memicu notifikasi ke observer.

4. Registrasi Observer dan Penggunaan

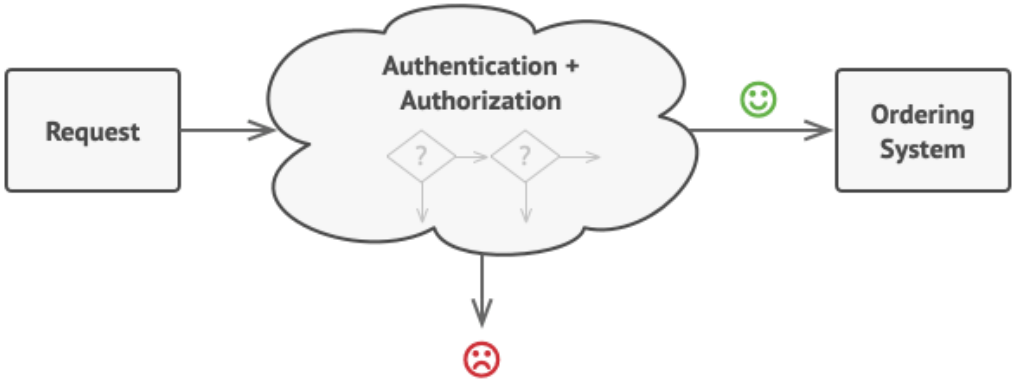
Menghubungkan MahasiswaModel dengan ConsoleLogger dan melakukan aksi pada model untuk memicu pencatatan.

```
public class MainApp {
    public static void main(String[] args) {
        MahasiswaModel model = new MahasiswaModel();
        ConsoleLogger logger = new ConsoleLogger();

        model.addObserver(logger);  // Logger mendaftar ke model

        Mahasiswa mhs = new Mahasiswa("Budi", "12345", 3.5);
        model.addMahasiswa(mhs);  // Akan memicu pencatatan otomatis
    }
}
```

Saat addMahasiswa() dipanggil, model memberi notifikasi ke semua observer yang telah terdaftar. Dalam contoh ini, ConsoleLogger akan mencetak informasi ke konsol.

14.4.2. Chain of Responsibility

Chain of Responsibility adalah pola desain perilaku yang memungkinkan sejumlah objek untuk menangani permintaan secara berantai (Sarcar, 2022). Pola ini menghindari pengikatan pengirim permintaan dengan penerima spesifiknya, dan memungkinkan lebih dari satu objek memiliki kesempatan untuk menangani permintaan tersebut. Tujuan utama pola ini adalah untuk melewatkan permintaan di sepanjang rantai objek sampai salah satu dari mereka menangani permintaan itu.



Gambar 14.2.1 Ilustrasi bagaimana setiap permintaan (*request*) harus melewati serangkaian pemeriksaan sebelum diproses.

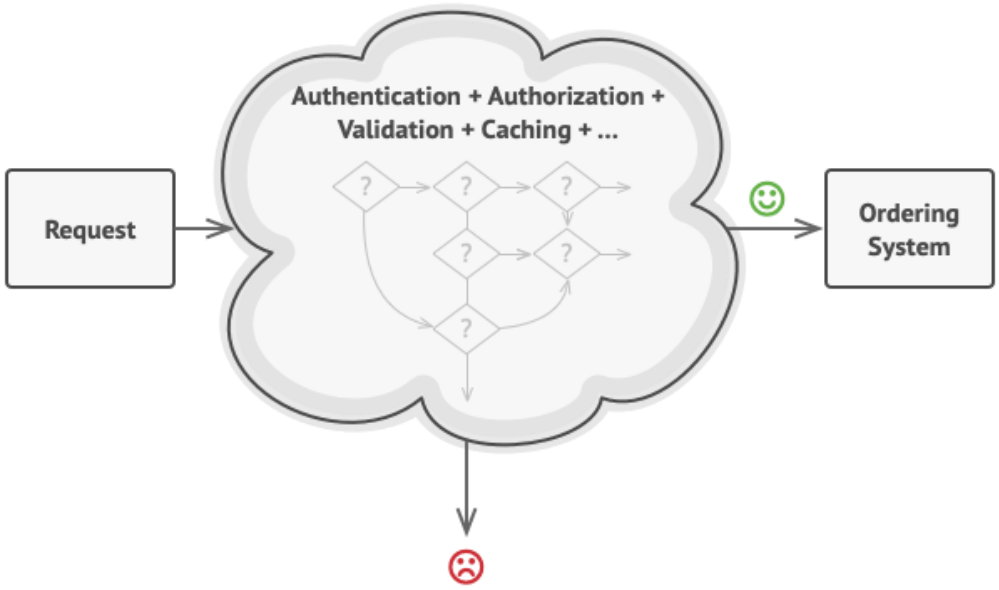
Sebagai suatu contoh yaitu aplikasi yang mencatat data mahasiswa, validasi input merupakan salah satu aspek penting yang harus dilakukan sebelum data dimasukkan ke dalam

database. Data seperti nama mahasiswa, NIM, dan IPK harus diperiksa terlebih dahulu agar sesuai dengan ketentuan yang berlaku. Dalam konteks ini, **Chain of Responsibility (CoR) Pattern** merupakan salah satu pola desain perilaku (*behavioral design pattern*) yang dapat digunakan untuk menyusun validasi secara modular dan fleksibel. Pola ini memungkinkan serangkaian objek (*handlers*) untuk memproses permintaan satu per satu dalam sebuah *rantai*, hingga permintaan tersebut diproses atau ditolak. Dalam ruang lingkup validasi data mahasiswa, setiap handler dalam rantai bertanggung jawab terhadap satu jenis validasi, seperti memeriksa apakah nama kosong, apakah NIM sudah digunakan, atau apakah IPK berada dalam batas yang diperbolehkan.

Selama ini, teknik validasi input pada aplikasi JavaFX umumnya dilakukan secara langsung di dalam controller atau class proses bisnis, menggunakan struktur bersarang if-else atau switch-case. Meskipun mudah dipahami, pendekatan ini membuat kode sulit dibaca dan dirawat, terutama jika jumlah aturan validasi bertambah. Validasi juga cenderung menjadi bagian dari antarmuka pengguna (UI), bukan bagian dari model, yang melanggar prinsip pemisahan tanggung jawab (*separation of concerns*).

```
if (nama.isEmpty()) {
    // tampilkan pesan
} else if (!isNimUnique(nim)) {
    // tampilkan pesan
} else if (ipk < 0 || ipk > 4) {
    // tampilkan pesan
}
```

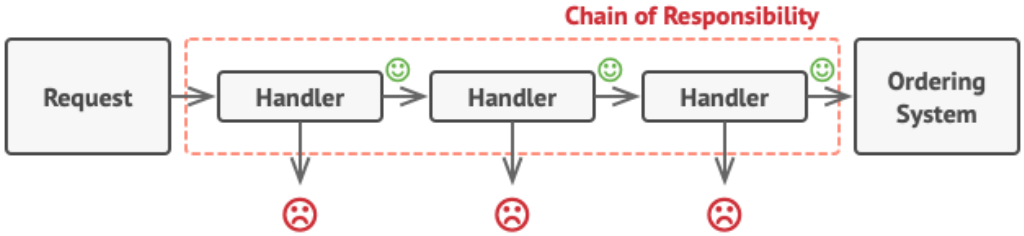
Evaluasi terhadap pendekatan konvensional ini menunjukkan beberapa kelebihan, seperti kemudahan implementasi awal dan keterpaduan langsung dalam logika antarmuka. Namun, kekurangannya tidak dapat diabaikan: pertambahan aturan validasi membuat logika menjadi rumit, sulit diuji secara unit, serta mengurangi fleksibilitas sistem saat ingin mengubah atau mengganti aturan validasi tertentu seperti yang dilihat pada Gambar 14.2.2. Selain itu, validasi yang tersebar di banyak tempat rentan terhadap inkonsistensi dan duplikasi.



Gambar 14.2.2 Mekanisme validasi bertambah seiring berkembangnya sistem, membuat kode program semakin kompleks.

Untuk mengatasi permasalahan tersebut, penggunaan pola *Chain of Responsibility* menawarkan pendekatan yang lebih bersih dan modular. Dengan memisahkan setiap aturan validasi ke dalam kelas handler tersendiri dan menghubungkannya dalam sebuah rantai, proses validasi dapat dilakukan secara berurutan, di mana setiap handler memutuskan apakah akan memproses atau meneruskan ke handler berikutnya. Hal ini memungkinkan sistem validasi yang mudah dikembangkan, diuji secara independen, dan diperluas tanpa mengubah struktur utama program. Tujuan utama dari penerapan Chain of Responsibility dalam validasi input data mahasiswa adalah:

- Memisahkan logika validasi dari antarmuka pengguna agar kode lebih terstruktur dan mudah diuji.
- Memungkinkan komposisi aturan validasi secara fleksibel.
- Mengurangi duplikasi kode dan meningkatkan maintainability sistem.



Gambar 14.2.3 Ilustrasi bagaimana Chain of Responsibility diterapkan pada Ordering System.

Pola Chain of Responsibility diimplementasikan dengan tahap-tahap berikut:

1. **Menyusun kelas abstrak InputValidator sebagai dasar semua handler.**

   Sebagai antarmuka atau kelas dasar yang mendefinisikan kontrak setNext() dan validate(). Semua validator konkret akan mewarisi ini.

```
public abstract class InputValidator {
```

```
    protected InputValidator next;

    public InputValidator setNext(InputValidator nextValidator) {
        this.next = nextValidator;
        return nextValidator;
    }

    public boolean validate(Mahasiswa mhs) {
        if (!doValidate(mhs)) return false;
        return next == null || next.validate(mhs);
    }

    protected abstract boolean doValidate(Mahasiswa mhs);
}
```

Method setNext() akan menyambungkan handler berikutnya, method validate() mengatur alur berantai antar handler sedangkan doValidate() adalah metode yang diimplementasikan oleh masing-masing validator konkret.

2. Implementasikan Validator Konkret

Mengimplementasikan validator khusus seperti EmptyFieldValidator, IPKRangeValidator, dan NIMUniqueValidator.

```
public class EmptyFieldValidator extends InputValidator {
    @Override
    protected boolean doValidate(Mahasiswa mhs) {
        if (mhs.getNama().isEmpty() || mhs.getNim().isEmpty()) {
            System.out.println("Validasi gagal: Nama atau NIM tidak
boleh kosong.");
            return false;
        }
        return true;
    }
}

public class IPKRangeValidator extends InputValidator {
    @Override
    protected boolean doValidate(Mahasiswa mhs) {
        double ipk = mhs.getIpk();
        if (ipk < 0.0 || ipk > 4.0) {
            System.out.println("Validasi gagal: IPK harus antara 0.0
dan 4.0");
            return false;
        }
        return true;
    }
}
```

```
public class NIMUniqueValidator extends InputValidator {
    private List<String> existingNIMs;

    public NIMUniqueValidator(List<String> existingNIMs) {
        this.existingNIMs = existingNIMs;
    }

    @Override
    protected boolean doValidate(Mahasiswa mhs) {
        if (existingNIMs.contains(mhs.getNim())) {
            System.out.println("Validasi gagal: NIM sudah terdaftar.");
            return false;
        }
        return true;
    }
}
```

3. Susun Rantai Validasi di Client (Controller/UI)

Menghubungkan handler tersebut dalam urutan tertentu dalam controller sebelum melakukan aksi penyimpanan.

```
public class MahasiswaController {
    private MahasiswaModel model;

    public MahasiswaController(MahasiswaModel model) {
        this.model = model;
    }

    public void prosesTambahMahasiswa(String nama, String nim, double
ipk) {
        Mahasiswa mhs = new Mahasiswa(nama, nim, ipk);

        List<String> nimSudahAda = model.getDaftarNIM();  // ambil dari
database

        InputValidator validator = new EmptyFieldValidator();
        validator.setNext(new NIMUniqueValidator(nimSudahAda))
                 .setNext(new IPKRangeValidator());

        if (validator.validate(mhs)) {
            model.addMahasiswa(mhs);
            System.out.println("Mahasiswa berhasil ditambahkan.");
        } else {
            System.out.println("Mahasiswa gagal ditambahkan.");
        }
    }
```

```
}
```

Pada contoh program diatas, Controller menyusun rantai validator: Kosong → NIM unik → IPK. Jika semua validasi berhasil, data diteruskan ke MahasiswaModel. Jika salah satu gagal, proses berhenti.

4. Integrasi ke Mekanisme Penyimpanan Data

Setelah itu, integrasikan ke mekanisme penyimpanan data aplikasi.

```
public class MahasiswaModel {
    private List<Mahasiswa> data = new ArrayList<>();

    public List<String> getDaftarNIM() {
        return data.stream().map(Mahasiswa::getNim).toList();
    }

    public void addMahasiswa(Mahasiswa mhs) {
        data.add(mhs);
        // Proses insert ke SQLite di sini
    }
}
```

14.3 Rangkuman Materi

1. Observer Pattern adalah Pola desain perilaku yang memungkinkan objek (*Observer*) menerima notifikasi otomatis dari objek lain (*Subject*) ketika terjadi perubahan. Karakteristik dari pola desain ini adalah :
   a. Memfasilitasi hubungan satu-ke-banyak antar objek.
   b. Subjek tidak mengetahui secara eksplisit siapa observer-nya → menghasilkan loose coupling.
   c. Umum digunakan dalam sistem event-driven seperti GUI, logging, atau notifikasi.
   d. Komponen utama:
      i. Subject: Menyimpan daftar observer dan memanggil notify().
      ii. Observer: Mendefinisikan antarmuka callback update() atau onChanged().
   e. Kelebihan:
      i. Memisahkan sumber perubahan dan reaksi terhadap perubahan.
      ii. Mudah ditambahkan observer baru tanpa mengubah Subject.
   f. Kekurangan:
      i. Potensi kebingungan jika terlalu banyak observer yang aktif.
      ii. Sulit di-debug saat banyak dependensi tidak eksplisit.
2. Chain of Responsibility Pattern ialah Pola desain perilaku yang menyusun sejumlah objek dalam rantai untuk menangani permintaan secara berurutan. Karakteristik dari pola desain ini adalah :
   a. Setiap objek (handler) memiliki kesempatan untuk menangani permintaan atau meneruskannya.
   b. Digunakan untuk menghindari struktur if-else bertingkat yang kompleks.
   c. Cocok untuk validasi berlapis, middleware, atau filter.
   d. Komponen utama:

i. Handler: Interface/kelas dasar yang mendefinisikan setNext() dan handle().
ii. ConcreteHandler: Implementasi logika pemeriksaan.

e. Kelebihan:
i. Memisahkan tanggung jawab ke dalam unit yang mandiri.
ii. Mudah menambah, mengubah, atau menyusun ulang handler tanpa mengubah client.

f. Kekurangan:
i. Tidak ada jaminan bahwa permintaan akan ditangani.
ii. Sulit dilacak saat rantai terlalu panjang atau dinamis.

14.4 Latihan

Anda diminta untuk melakukan refactor/modifikasi perbaikan pada aplikasi desktop sederhana menggunakan JavaFX dan SQLite untuk mencatat data mahasiswa yang telah dikembangkan pada modul 13. Refactor yang dilakukan bertujuan untuk meningkatkan modularitas, maintainability, dan separation of concerns menggunakan pola desain perilaku (behavioral design patterns). Lakukan modifikasi berikut:

1. Modifikasi class ManagerMahasiswa menggunakan DAO untuk mengakses data dari database.
   a. Buat class MahasiswaDAO dengan metode getAllMahasiswa(), insertMahasiswa(), updateMahasiswa(), dan deleteMahasiswa().
   b. Ganti semua akses SQL langsung di ManagerMahasiswa menjadi pemanggilan metode DAO.
2. Tambahkan Observer Pattern untuk mencatat log perubahan data mahasiswa ke tabel log di SQLite.
   a. Buat interface MahasiswaObserver dengan metode onMahasiswaChanged(String message).
   b. Implementasikan kelas DatabaseLogger sebagai observer yang menyimpan log ke tabel log.
   c. Registrasikan logger pada ManagerMahasiswa atau MahasiswaModel.
   d. Setiap kali mahasiswa ditambah, diubah, atau dihapus, sistem secara otomatis mencatat log ke tabel log.
3. Modifikasi validasi input data mahasiswa menggunakan Chain of Responsibility Pattern.
   a. Buat abstract class InputMahasiswaValidator dengan method validate(Mahasiswa) dan getLastErrorMessage().
   b. Implementasikan minimal 2 concrete validator:
      i. EmptyFieldMahasiswaValidator → memeriksa nama dan NIM tidak kosong.
      ii. NIMUniqueMahasiswaValidator → memeriksa NIM tidak duplikat.
      iii. IPKRangeMahasiswaValidator → memeriksa IPK berada pada rentang 0.0–4.0.
   c. Susun rantai validator di controller. Jika validasi gagal, tampilkan pesan kesalahan melalui komponen Alert.

14.5 Kunci Jawaban

Refactor dilakukan dengan tahpan-tahapan berikut ini :

1. Tambahkan Observer Pattern untuk mencatat log

Setiap kali mahasiswa ditambah/dihapus/diperbarui, sistem secara otomatis mencatat ke tabel log tanpa perlu pemanggilan eksplisit di semua tempat. Untuk itu buat interface MahasiswaObserver:

```
public interface MahasiswaObserver {
    void onMahasiswaChanged(String message);
}
```

Implementasikan penyimpanan log ke database.

LogEvent.Java

```
import java.time.LocalDateTime;

public class LogEvent {
    private int id;
    private String event;
    private LocalDateTime timestamp;

    // Konstruktor tanpa parameter (penting untuk ORM/framework
tertentu)
    public LogEvent() {}

    // Konstruktor lengkap
    public LogEvent(int id, String event, LocalDateTime timestamp)
{
        this.id = id;
        this.event = event;
        this.timestamp = timestamp;
    }

    // Konstruktor tanpa ID (untuk insert baru)
    public LogEvent(String event) {
        this.event = event;
        this.timestamp = LocalDateTime.now(); // default waktu
sekarang
    }

    // Getter dan Setter
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getEvent() {
        return event;
    }

    public void setEvent(String event) {
        this.event = event;
    }
```

```
    public LocalDateTime getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(LocalDateTime timestamp) {
        this.timestamp = timestamp;
    }

    @Override
    public String toString() {
        return "[" + timestamp + "] " + event;
    }
}
```

```
import java.time.LocalDateTime;

public class LogEvent {
    private int id;
    private String event;
    private LocalDateTime timestamp;

    // Konstruktor tanpa parameter
    public LogEvent() {}

    // Konstruktor lengkap
    public LogEvent(int id, String event, LocalDateTime timestamp)
    {
        this.id = id;
        this.event = event;
        this.timestamp = timestamp;
    }

    // Konstruktor tanpa ID (untuk insert baru)
    public LogEvent(String event) {
        this.event = event;
        this.timestamp = LocalDateTime.now(); // default waktu
sekarang
    }

    // Getter dan Setter
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getEvent() {
        return event;
    }

    public void setEvent(String event) {
        this.event = event;
    }
```

```
    public LocalDateTime getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(LocalDateTime timestamp) {
        this.timestamp = timestamp;
    }

    @Override
    public String toString() {
        return "[" + timestamp + "] " + event;
    }
}
```

MahasiswaLogManager,java

```
public class MahasiswaLogManager implements MahasiswaObserver {
    LogRepository logRepository;

    public MahasiswaLogManager() {
        this.logRepository = new
LogRepository(DBConnectionManager.getConnection());
    }

    @Override
    public void onMahasiswaChanged(String message) {
        this.logRepository.save(new LogEvent(message));
    }


    public List<LogEvent> getAllLogs() {
        return this.logRepository.findAll();
    }
}
```

Modifikasi ManagerMahasiswa.java agar dapat menambahkan observer.

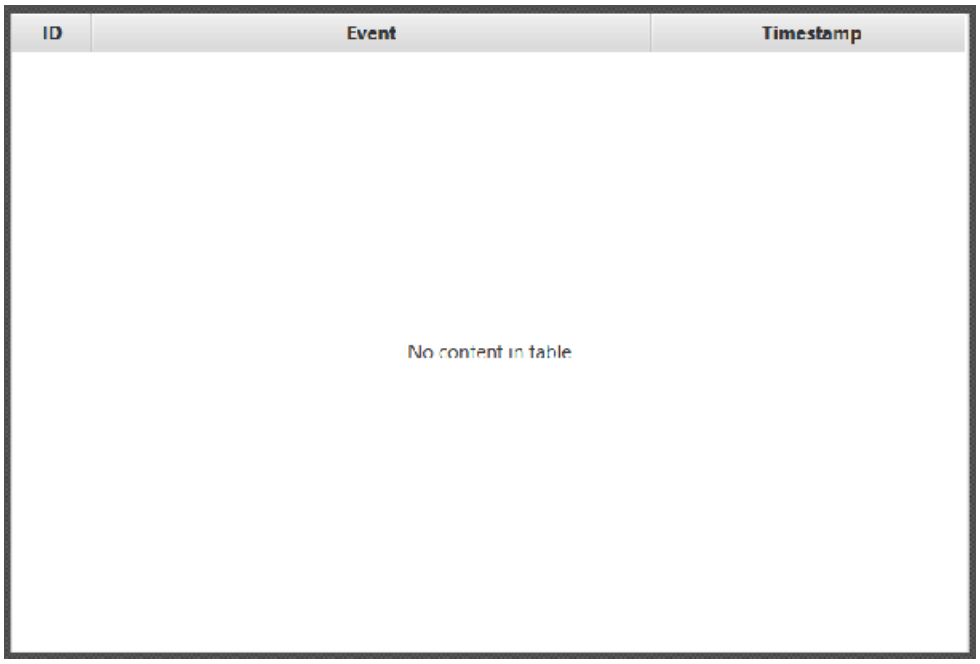
```
public class ManagerMahasiswa {
    . . .
    private final List<MahasiswaObserver> observers = new
ArrayList<>();

    public void addObserver(MahasiswaObserver observer) {
        observers.add(observer);
    }

    private void notifyObservers(String message) {
        for (MahasiswaObserver observer : observers) {
            observer.onMahasiswaChanged(message);
        }
    }
    . . .
}
```

Tambahkan MahasiswaLogManager sebagai observer pada MahasiswaViewController.

```
public class MahasiswaViewController implements Initializable {
. . .

    @Override
    public void initialize(URL url, ResourceBundle resourceBundle)
    {
        . . .
        this.manager = new ManagerMahasiswa();
        this.manager.addObserver(new MahasiswaLogManager());
        . . .
    }
```

Buat UI untuk melihat log perubahan data mahasiswa
log-view.fxml

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

<?import javafx.scene.control.TableColumn?>
<?import javafx.scene.control.TableView?>
<?import javafx.scene.layout.VBox?>

<VBox spacing="20.0" xmlns="http://javafx.com/javafx/21"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="org.ukdw.controller.LogViewController">
    <TableView fx:id="logTable" prefHeight="400" prefWidth="600">
        <columns>
            <TableColumn fx:id="idColumn" prefWidth="50" text="ID"
/>
            <TableColumn fx:id="eventColumn" prefWidth="350"
text="Event" />
            <TableColumn fx:id="timestampColumn" prefWidth="200"
text="Timestamp" />
        </columns>
    </TableView>
</VBox>
```



Implementasi LogViewController.java

```
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import org.ukdw.manager.MahasiswaLogManager;
import org.ukdw.model.LogEvent;

public class LogViewController {

    @FXML
    private TableView<LogEvent> logTable;
    @FXML
    private TableColumn<LogEvent, Integer> idColumn;
    @FXML
    private TableColumn<LogEvent, String> eventColumn;
    @FXML
    private TableColumn<LogEvent, String> timestampColumn;

    private final MahasiswaLogManager mahasiswaLogManager = new
MahasiswaLogManager();

    @FXML
    public void initialize() {

        idColumn.setCellValueFactory(new
PropertyValueFactory<>("id"));
        eventColumn.setCellValueFactory(new
PropertyValueFactory<>("event"));
        timestampColumn.setCellValueFactory(new
PropertyValueFactory<>("timestamp"));
```

```
        ObservableList<LogEvent> logList =
FXCollections.observableArrayList(mahasiswaLogManager.getAllLogs())
;
        logTable.setItems(logList);
    }
}
```

Modifikasi mahasiswa-view.fxml dan controllernya untuk dapat menampilkan tampilan UI Log

mahasiswa-view.fxml

```
. . .
<MenuBar>
  <menus>
    <Menu mnemonicParsing="false" text="File">
      <items>
        <MenuItem mnemonicParsing="false"
onAction="#onActionLogout" text="Logout" />
      </items>
    </Menu>
    <Menu mnemonicParsing="false" text="Settings">
      <items>
        <MenuItem mnemonicParsing="false"
onAction="#onActionShowLog" text="Log" />
      </items>
    </Menu>
    <Menu mnemonicParsing="false" text="Help">
      <items>
        <MenuItem mnemonicParsing="false" onAction="#onActionAbout"
text="About" />
      </items>
    </Menu>
  </menus>
   <VBox.margin>
      <Insets />
   </VBox.margin>
</MenuBar>
...
```

MahasiswaViewController.java

```
...

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

2. Modifikasi validasi input menggunakan Chain of Responsibility. Validasi dilakukan secara modular dan terpisah. Jika satu validasi gagal, proses berhenti, dan pesan kesalahan ditampilkan melalui Alert.
   Buat abstract InputMahasiswaValidator.

```
public abstract class InputMahasiswaValidator {
```

```
    protected InputMahasiswaValidator next;
    protected String errorMessage;

    public InputMahasiswaValidator setNext(InputMahasiswaValidator
nextValidator) {
        this.next = nextValidator;
        return nextValidator;
    }

    public boolean validate(Mahasiswa mhs) {
        if (!doValidate(mhs)) return false;
        return next == null || next.validate(mhs);
    }

    protected abstract boolean doValidate(Mahasiswa mhs);

    public String getLastErrorMessage() {
        if (this.errorMessage != null) {
            return this.errorMessage;
        } else if (this.next != null) {
            return this.next.getLastErrorMessage();
        }
        return null;
    }
}
```

Buat EmptyFieldMahasiswaValidator

```
public class EmptyFieldMahasiswaValidator extends
InputMahasiswaValidator {
    @Override
    protected boolean doValidate(Mahasiswa mhs) {
        if (mhs.getNama().isEmpty() || mhs.getNim().isEmpty()) {
            errorMessage = "Validasi gagal: Nama atau NIM tidak
boleh kosong.";
            return false;
        }
        return true;
    }
}
```

Buat IPKRangeMahasiswaValidator

```
public class IPKRangeMahasiswaValidator extends
InputMahasiswaValidator {
    @Override
    protected boolean doValidate(Mahasiswa mhs) {
        double nilai = mhs.getNilai();
        if (nilai < 0.0 || nilai > 4.0) {
            errorMessage = "Validasi gagal: IPK harus antara 0.0
hingga 4.0.";
            return false;
        }
        return true;
    }
}
```

Buat NIMUniqueMahasiswaValidator

```
public class NIMUniqueMahasiswaValidator extends
InputMahasiswaValidator {
    private final List<String> existingNims;

    public NIMUniqueMahasiswaValidator(List<String> existingNims)
{
        this.existingNims = existingNims;
    }

    @Override
    protected boolean doValidate(Mahasiswa mhs) {
        if (existingNims.contains(mhs.getNim())) {
            errorMessage = "Validasi gagal: NIM sudah digunakan.";
            return false;
        }
        return true;
    }
}
```

Implementasi validator saat proses tambah dan update data mahasiswa

```
@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 = masterData.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;
    }

    if (manager.tambahMahasiswa(newMahasiswa)) {
        masterData.add(newMahasiswa);
        Alert alert = new Alert(Alert.AlertType.INFORMATION, "Data
Mahasiswa Ditambahkan!");
        alert.show();
        bersihkan();
    } else {
```

```
        Alert alert = new Alert(Alert.AlertType.ERROR, "Data
Mahasiswa gagal Ditambahkan!");
        alert.show();
        bersihkan();
    }
}

@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;
    }

    if (manager.updateMahasiswa(updatedMahasiswa)) {
        selectedMahasiswa.setNama(updatedMahasiswa.getNama());
        selectedMahasiswa.setNilai(updatedMahasiswa.getNilai());
        selectedMahasiswa.setFoto(updatedMahasiswa.getFoto());
        tblView.refresh();
        Alert alert = new Alert(Alert.AlertType.INFORMATION, "Data
Mahasiswa Diperbarui!");
        alert.show();
        bersihkan();
    } else {
        Alert alert = new Alert(Alert.AlertType.ERROR, "Data
Mahasiswa gagal Diperbarui!");
        alert.show();
        bersihkan();
    }
}
```