MODUL 13
DATA ACCESS OBJECT (DAO) & DEPENDENCY INJECTION

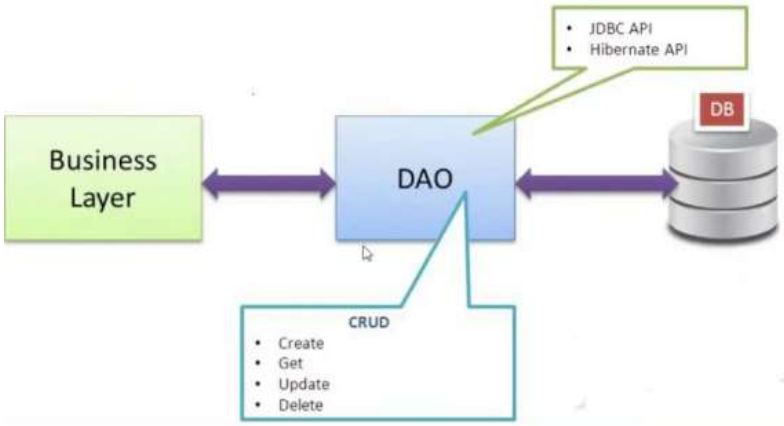
13.1 Tujuan Materi Pembelajaran

1. Menjelaskan peran dan manfaat penggunaan pola desain Data Access Object (DAO) dalam pemisahan logika bisnis dan akses data.
2. Mendesain antarmuka DAO dan mengimplementasikan operasi dasar CRUD menggunakan bahasa pemrograman Java dan IDE IntelliJ.
3. Menerapkan prinsip Dependency Injection (DI) untuk menghindari keterkaitan kuat (tight coupling) antara komponen dalam aplikasi.
4. Membedakan dan mengimplementasikan tiga metode Dependency Injection: melalui konstruktor, setter, dan field injection.

13.2 Materi

13.2.1. Data Access Object

Dalam pengembangan perangkat lunak modern, pengelolaan interaksi antara logika bisnis dan data yang disimpan secara persisten menjadi tantangan penting. Seiring meningkatnya kompleksitas aplikasi, pemisahan yang jelas antara aturan bisnis dan logika akses data menjadi krusial untuk menjaga kemudahan pemeliharaan, skalabilitas, dan pengujian sistem. Pada praktik sebelumnya, banyak pengembang menulis langsung perintah SQL dalam kode logika bisnis atau antarmuka pengguna, yang menyebabkan keterkaitan kode yang kuat dan sulit diuji serta dipelihara.



Gambar 13.2.1 Ilustrasi bagaimana DAO memisahkan *bussines layer* dengan implementasi detail dari penyimpanan data di *database*.

Data Access Object (DAO) adalah sebuah pola desain struktural yang bertugas mengabstraksi dan mengenkapsulasi seluruh akses ke sumber data. DAO menyediakan antarmuka standar untuk melakukan operasi seperti insert, update, delete, dan getAll, sambil menyembunyikan detail teknis dari proses penyimpanan data seperti koneksi database atau penggunaan SQL. Sebagai contoh, implementasi DAO mencakup penggunaan bahasa Java dengan JDBC di IntelliJ untuk mengelola entitas Mahasiswa dan menghubungkannya ke database.

Pada pendekatan sebelumnya (terutama pada sistem prosedural), kode untuk akses database sering ditempatkan dalam lapisan logika bisnis atau antarmuka pengguna secara langsung. Hal ini menyebabkan:

- Keterkaitan erat antar komponen (tightly coupled)
- Duplikasi dan redundansi kode
- Kesulitan dalam pengujian unit
- Ketergantungan tinggi pada jenis dan struktur database

Pendekatan langsung ini mungkin efisien dan lebih cepat diimplementasikan untuk prototipe aplikasi kecil, namun memiliki kekurangan besar dalam jangka panjang, yaitu:

- Sulit diuji dan dipelihara
- Perubahan di level database berdampak besar ke seluruh aplikasi
- Tidak modular
- Tidak adanya lapisan abstraksi yang memisahkan logika bisnis dari detail penyimpanan data.

Pola desain DAO menjawab masalah tersebut dengan cara menyediakan pemisahan *responsibility* yang jelas. DAO mendukung arsitektur yang bersih, modular, dan fleksibel terhadap perubahan teknologi penyimpanan data. Hal ini juga mendukung prinsip Single Responsibility dan Dependency Inversion dalam desain perangkat lunak berorientasi objek.

Berikut ini adalah tahapan untuk menerapkan pola desain DAO.

1. Membuat Entity Class
   Misalnya, `Mahasiswa` dengan properti seperti `nim`, `nama`, `prodi`.

```
public class Mahasiswa {
    private String nim;
    private String nama;
    private String prodi;
    // Constructor, getter, setter
}
```

2. Membuat Interface DAO
   Misalnya, `MahasiswaDAO` yang mendefinisikan operasi CRUD.

```
public interface MahasiswaDAO {
       void insert(Mahasiswa m);
       void update(Mahasiswa m);
       void delete(String nim);
       List<Mahasiswa> getAll();
   }
```

3. Membuat Implementasi DAO
   Misalnya, `MahasiswaDAOImpl` yang menggunakan JDBC untuk melakukan operasi database.

```
public class MahasiswaDAOImpl implements MahasiswaDAO {
    private Connection conn;
    public MahasiswaDAOImpl(Connection conn) {
        this.conn = conn;
    }
```

```
    @Override
    public void insert(Mahasiswa m) {
        try (PreparedStatement stmt = conn.prepareStatement(
            "INSERT INTO mahasiswa (nim, nama, prodi) VALUES (?,
?, ?)")) {
            stmt.setString(1, m.getNim());
            stmt.setString(2, m.getNama());
            stmt.setString(3, m.getProdi());
            stmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    // Implementasi update, delete, getAll...
}
```

4. Menyediakan Koneksi Database
   Biasanya dibuat dalam class `DBConnectionManager`.
5. Menggunakan DAO dalam Business Layer
   Seperti pada class `ManagerMahasiswa`, DAO di-*inject* dan digunakan tanpa tergantung pada detail database-nya.

```
public class ManagerMahasiswa {
    private MahasiswaDAO dao;

    public ManagerMahasiswa(MahasiswaDAO dao) {
        this.dao = dao;
    }

    public void tambahMahasiswa(Mahasiswa m) {
        dao.insert(m);
    }
    // Metode lain untuk update, delete, getAll...
}
```

Dengan demikian penerapan pola desain DAO dalam aplikasi maka kelebihan yang akan mendapatkan ialah sebagai berikut:

- Maka struktur kode akan lebih terorganisir dan modular.
- Kode akan lebih mudah diuji, dirawat, dan dikembangkan.
- Komponen logika bisnis tidak akan terpengaruh oleh perubahan pada lapisan penyimpanan data.

13.2.2. Dependency Injection

Dalam pengembangan perangkat lunak berorientasi objek, salah satu tantangan utama adalah bagaimana mengelola dependency—yaitu objek-objek yang saling bergantung satu sama lain. Pada praktik tradisional, objek sering kali membuat instance dari objek lain

secara langsung (misalnya new ClassB() di dalam ClassA), yang menyebabkan keterkaitan kuat (tight coupling) dan menyulitkan pengujian serta pemeliharaan.

Dependency Injection (DI) adalah sebuah teknik dalam desain perangkat lunak di mana sebuah objek menerima dependensinya dari luar, bukan membuatnya sendiri. Dengan kata lain, objek tidak lagi bertanggung jawab untuk menciptakan objek lain yang ia butuhkan, melainkan dependensi tersebut disuntikkan (*injected*) ke dalamnya. DI dapat diterapkan melalui tiga pendekatan utama: *constructor injection*, *setter injection*, dan *field injection*. Dalam konteks praktikum ini, DI akan digunakan untuk menyuntikkan objek DAO ke dalam class logika bisnis (ManagerMahasiswa).

Pada pendekatan konvensional, setiap class yang membutuhkan dependensi akan membuat sendiri objek tersebut. Contoh:

```
public class MahasiswaViewController implements Initializable {
    private MahasiswaDAO dao = new MahasiswaDAOImpl();

    @Override
    public void initialize(URL url, ResourceBundle resourceBundle) {
        dao = new MahasiswaDAOImpl();
    }
}
```

Walaupun pendekatan diatas lebih sederhana dan mudah diterapkan dalam skala kecil, namun memiliki kekurangan:

- Menyebabkan *tight coupling* antar komponen
- Sulit mengganti dependensi dengan implementasi lain atau mock

Kondisi ini menunjukkan adanya kebutuhan akan strategi yang lebih fleksibel dalam pengelolaan dependensi antar objek.

DI menyediakan solusi untuk mengurangi keterkaitan antar objek. Dengan DI, kita dapat mengganti komponen dengan mudah (misalnya DAO yang berbeda), melakukan pengujian dengan mock object, dan membuat sistem lebih fleksibel terhadap perubahan. DI juga memfasilitasi desain arsitektur berbasis kontrak (*interface-driven*) yang lebih *robust* dan *scalable*. Berikut ini adalah contoh penggunaan 3 jenis DI :

1. Construction Injection

Dependency disuntikkan melalui constructor. Ini adalah metode DI yang paling umum dan aman, karena membuat dependency tersedia secara immutable sejak objek dibuat.

```
public class ManagerMahasiswa {
    private MahasiswaDAO dao;

    // Constructor Injection
    public ManagerMahasiswa(MahasiswaDAO dao) {
        this.dao = dao;
    }

    public void tambahMahasiswa(Mahasiswa m) {
        dao.insert(m);
```

```
    }
}
```

Cara penggunaan

```
Connection conn = DBConnectionManager.getConnection();
MahasiswaDAO dao = new MahasiswaDAOImpl(conn);
ManagerMahasiswa manager = new ManagerMahasiswa(dao);
```

2. Setter Injection

Dependency disuntikkan menggunakan method setter. Cocok jika dependency bersifat opsional atau ingin dapat diubah setelah objek dibuat.

```
public class ManagerMahasiswa {
    private MahasiswaDAO dao;

    // Setter Injection
    public void setMahasiswaDAO(MahasiswaDAO dao) {
        this.dao = dao;
    }

    public void hapusMahasiswa(String nim) {
        dao.delete(nim);
    }
}
```

Cara penggunaan

```
ManagerMahasiswa manager = new ManagerMahasiswa();
manager.setMahasiswaDAO(new
MahasiswaDAOImpl(DBConnectionManager.getConnection()));
```

3. Field Injection

Dependency langsung disuntikkan ke atribut (field). Biasanya dilakukan melalui framework (seperti Spring) dengan anotasi @Inject atau @Autowired. Namun untuk contoh sederhana, kita simulasikan secara manual.

```
public class ManagerMahasiswa {
    // Field Injection
    public MahasiswaDAO dao;

    public List<Mahasiswa> tampilkanSemua() {
        return dao.getAll();
    }
}
```

Contoh Penggunaan

```
ManagerMahasiswa manager = new ManagerMahasiswa();
```

```
manager.dao = new
MahasiswaDAOImpl(DBConnectionManager.getConnection());
```

Catatan: Field Injection mudah diterapkan tapi kurang disarankan dalam praktik OOP murni karena melanggar prinsip enkapsulasi dan menyulitkan pengujian.

Dengan menggunakan salah satu jenis DI diatas maka manfaat yang akan didapat ialah sebagai berikut:

- Class akan lebih fleksibel dan tidak tergantung pada implementasi spesifik
- Dependensi dapat diganti, diuji, dan dipelihara dengan lebih mudah
- Desain aplikasi akan mendukung prinsip-prinsip arsitektur modern seperti SOLID.

13.3 Rangkuman Materi

1. Data Access Object (DAO) adalah pola desain yang bertujuan memisahkan logika bisnis dari logika akses data. DAO bertindak sebagai perantara antara objek aplikasi dan database, sehingga sistem menjadi lebih modular, mudah diuji, dan fleksibel terhadap perubahan teknologi penyimpanan data.
2. DAO melibatkan tiga komponen utama:
   a. Entity class (misal: Mahasiswa)
   b. Interface DAO (misal: MahasiswaDAO) yang mendefinisikan operasi seperti insert(), update(), delete(), dan getAll()
   c. Implementasi DAO (misal: MahasiswaDAOImpl) yang mengakses database menggunakan JDBC
3. Dependency Injection (DI) adalah teknik untuk menghindari keterkaitan kuat antar objek dengan menyuntikkan dependensi dari luar objek, bukan membuatnya sendiri. Ini mendukung desain yang fleksibel dan mudah diuji.
4. Tiga pendekatan umum dalam DI:
   a. Constructor Injection: dependensi diberikan saat objek dibuat
   b. Setter Injection: dependensi diberikan melalui method setter
   c. Field Injection: dependensi langsung diset ke atribut (kurang disarankan tanpa framework)
5. Penggunaan DAO dan DI secara bersamaan memungkinkan kita membangun aplikasi berarsitektur bersih (clean architecture) dengan pemisahan tanggung jawab yang jelas antara lapisan data, logika bisnis, dan antarmuka pengguna.

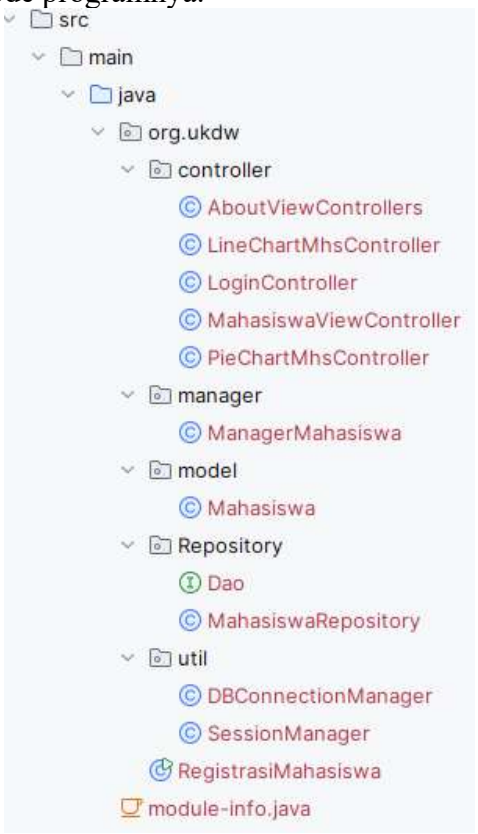
13.4 Latihan

Anda diminta untuk melakukan refactor/modifikasi perbaikan pada aplikasi desktop sederhana menggunakan JavaFX dan SQLite untuk mencatat data mahasiswa yang sudah dibuat pada modul 12. Refactor yang dilakukan adalah sebagai berikut :

1. Modifikasi class ManagerMahasiswa menggunakan DAO untuk mengakses data-data dari database.
2. Modifikasi class yang mengimplementasi DAO agar object connection menerapkan Dependency Injection sehingga connection dapat diberikan melalui constructor Injection.

13.5 Kunci Jawaban

Untuk menerapkan pola desain DAO, maka harus membuat package Repository terlebih dahulu untuk meletakan kode-kode class yang terkait akses data ke database. Setelah itu membuat class interface Dao yang bersisi method findById, findAll, save, update, dan delete. Class interface Dao akan di-implement oleh MahasiswaRepository, dan kedua class tersebut diletakan di-package Repository. Impelementasi construcotr Dependency injection dilakukan dengan menambahkan parameter input bertipe Connection pada constructor class yang mengimplementasi interface Dao. Berikut adalah gambaran struktur project dan contoh kode programnya.



Dao.java

```
import java.util.List;

public interface Dao<ObjectType, IdType> {

    ObjectType findById(IdType id);

    List<ObjectType> findAll();

    boolean save(ObjectType objectType);

    boolean update(ObjectType newObjectType);

    boolean delete(ObjectType objectType);
}
```

MahasiswaRepository.java

```
import org.ukdw.model.Mahasiswa;
import org.ukdw.util.DBConnectionManager;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class MahasiswaRepository implements Dao<Mahasiswa, String> {

    Connection connection;

    public MahasiswaRepository(Connection connection) {
        this.connection = connection;
        buatTabelJikaBelumAda();
    }

    // Membuat tabel Mahasiswa jika belum ada
    private void buatTabelJikaBelumAda() {
        String sql = "CREATE TABLE IF NOT EXISTS mahasiswa (" +
                "nim TEXT PRIMARY KEY, " +
                "nama TEXT NOT NULL, " +
                "nilai REAL NOT NULL, " +
                "foto BLOB) ";
        try {
            Statement stmt = this.connection.createStatement();
            stmt.execute(sql);
        } catch (SQLException e) {
            System.err.println("Gagal membuat tabel: " + e.getMessage());
        } finally {
            DBConnectionManager.closeConnection();
        }
    }

    @Override
    public Mahasiswa findById(String id) {
        return null;
    }

    @Override
    public List<Mahasiswa> findAll() {
        ArrayList<Mahasiswa> mhslist = new ArrayList<>();
        String sql = "SELECT * FROM mahasiswa";
        try {
            Statement stmt = this.connection.createStatement();
            ResultSet rs = stmt.executeQuery(sql);
            while (rs.next()) {
                mhslist.add(new Mahasiswa(
                        rs.getString("nim"),
                        rs.getString("nama"),
                        rs.getDouble("nilai"),
                        rs.getBytes("foto")
                ));
            }
        } catch (SQLException e) {
            System.err.println("Gagal membaca data mahasiswa: " +
e.getMessage());
        } finally {
            DBConnectionManager.closeConnection();
```

```
        }
        return mhslist;
    }

    // CREATE: Menambahkan Mahasiswa
    @Override
    public boolean save(Mahasiswa mahasiswa) {
        String sql = "INSERT INTO mahasiswa (nim, nama, nilai, foto)
VALUES (?, ?, ?, ?)";
        try (PreparedStatement pstmt =
this.connection.prepareStatement(sql)) {
            pstmt.setString(1, mahasiswa.getNim());
            pstmt.setString(2, mahasiswa.getNama());
            pstmt.setDouble(3, mahasiswa.getNilai());
            pstmt.setBytes(4, mahasiswa.getFoto());
            pstmt.executeUpdate();
            System.out.println("Mahasiswa berhasil ditambahkan.");
            return true;
        } catch (SQLException e) {
            System.err.println("Gagal menambahkan mahasiswa: " +
e.getMessage());
            return false;
        } finally {
            DBConnectionManager.closeConnection();
        }
    }

    // UPDATE: Memperbarui Data Mahasiswa
    @Override
    public boolean update(Mahasiswa newMahasiswa) {
        String sql = "UPDATE mahasiswa SET nama = ?, nilai = ? , foto = ?
WHERE nim = ?";
        try (PreparedStatement pstmt =
this.connection.prepareStatement(sql)) {
            pstmt.setString(1, newMahasiswa.getNama());
            pstmt.setDouble(2, newMahasiswa.getNilai());
            pstmt.setBytes(3, newMahasiswa.getFoto());
            pstmt.setString(4, newMahasiswa.getNim());
            int rowsAffected = pstmt.executeUpdate();
            if (rowsAffected > 0) {
                System.out.println("Data mahasiswa berhasil diperbarui.");
                return true;
            }
        } catch (SQLException e) {
            System.err.println("Gagal memperbarui data mahasiswa: " +
e.getMessage());
        } finally {
            DBConnectionManager.closeConnection();
        }
        return false;
    }

    // DELETE: Menghapus Mahasiswa
    @Override
    public boolean delete(Mahasiswa mahasiswa) {
        String sql = "DELETE FROM mahasiswa WHERE nim = ?";
        try (PreparedStatement pstmt =
```

```
this.connection.prepareStatement(sql)) {
            pstmt.setString(1, mahasiswa.getNim());
            int rowsAffected = pstmt.executeUpdate();
            if (rowsAffected > 0) {
                System.out.println("Mahasiswa dengan NIM " +
mahasiswa.getNim() + " telah dihapus.");
                return true;
            }
        } catch (SQLException e) {
            System.err.println("Gagal menghapus mahasiswa: " +
e.getMessage());
        } finally {
            DBConnectionManager.closeConnection();
        }
        return false;
    }
}
```

ManagerMahasiswa.java

```
public class ManagerMahasiswa {

    Dao<Mahasiswa, String> mahasiswaRepository;

    public ManagerMahasiswa() {
        mahasiswaRepository = new
MahasiswaRepository(DBConnectionManager.getConnection());
    }

    // CREATE: Menambahkan Mahasiswa
    public boolean tambahMahasiswa(Mahasiswa mahasiswa) {
        return mahasiswaRepository.save(mahasiswa);
    }

    public ArrayList<Mahasiswa> getAllMahasiswa() {
        return (ArrayList<Mahasiswa>) mahasiswaRepository.findAll();
    }

    // UPDATE: Memperbarui Data Mahasiswa
    public boolean updateMahasiswa(Mahasiswa mahasiswa) {
        return mahasiswaRepository.update(mahasiswa);
    }

    // DELETE: Menghapus Mahasiswa
    public boolean hapusMahasiswa(String nim) {
        return mahasiswaRepository.delete(new Mahasiswa(nim, "", 0,
"".getBytes()));
    }
}
```

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

13.4.1. Observer Patern

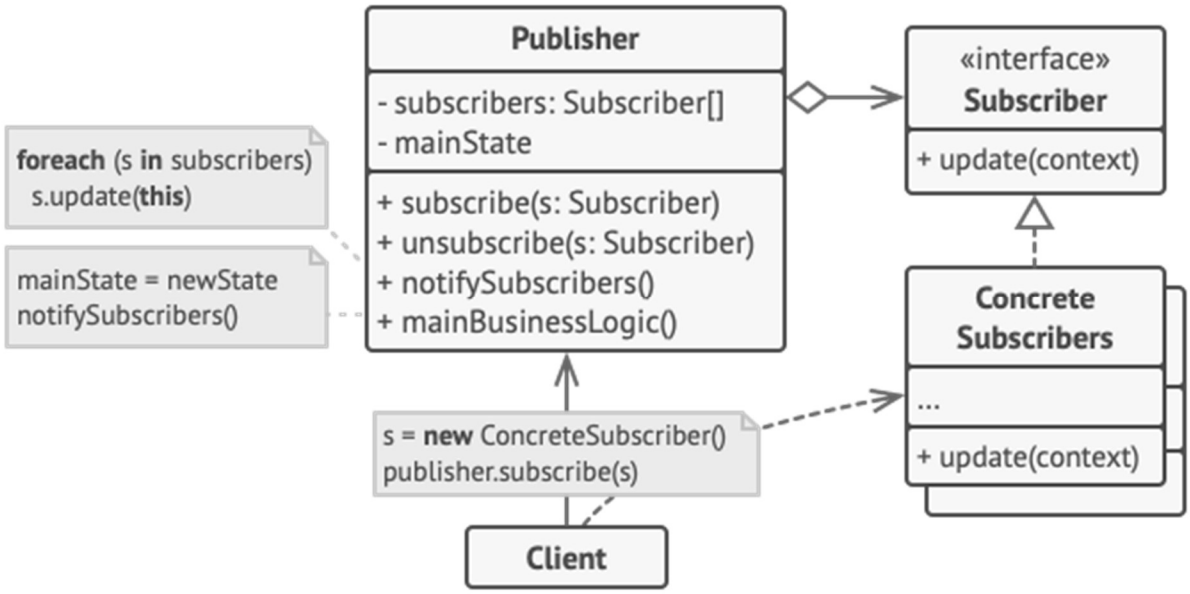
Dalam pengembangan aplikasi modern, aspek pemantauan dan pencatatan aktivitas sistem (sistem logging) merupakan elemen penting untuk mendukung keterlacakan (*traceability*), keamanan data, dan proses debugging. Salah satu kebutuhan umum dalam sistem yang melakukan perekaman data adalah kemampuan untuk 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

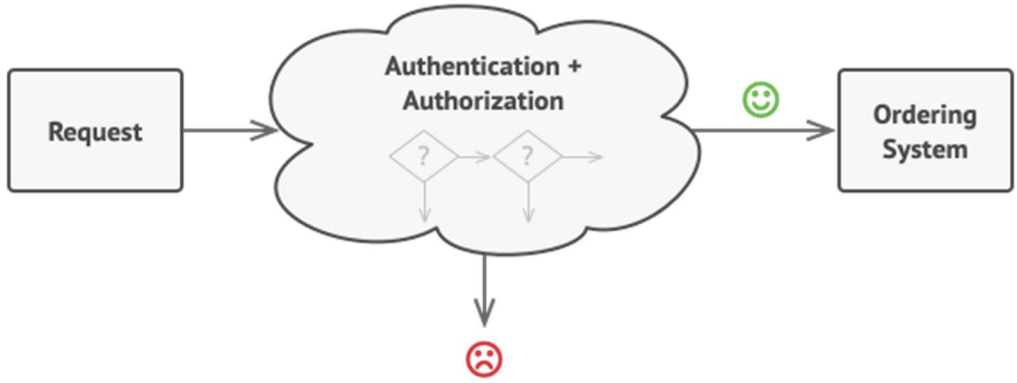
Menghubungkan MahasiswaModel dengan ConsoleLogger dan melakukan aksi pada model untuk memicu pencatatan.

```
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.

13.4.2. 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.

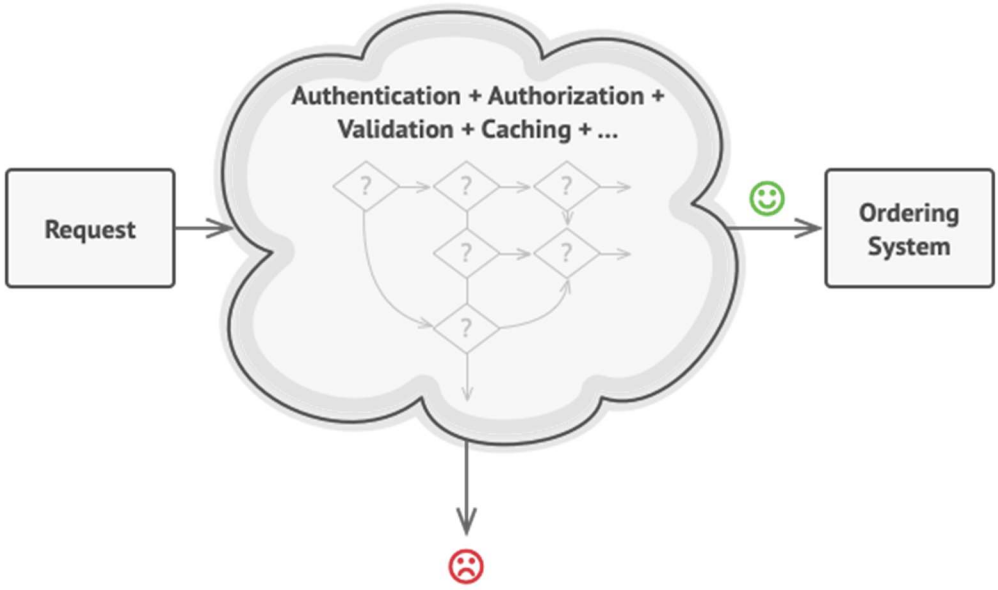
Sebagai suatu contoh yaitu aplikasi yang mencatat data mahasiswa, validasi input merupakan salah satu aspek penting yang harus dilakukan sebelum data dimasukkan ke dalam

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
}
```

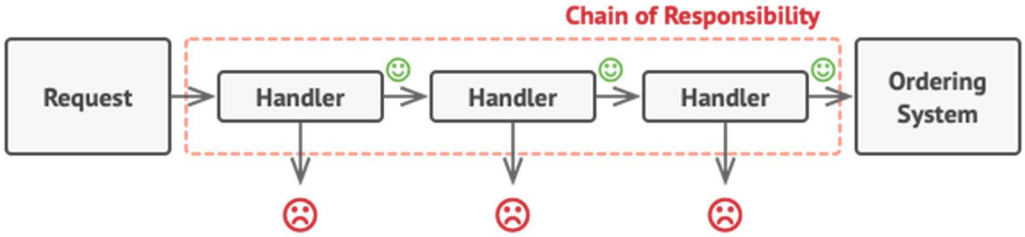
Evaluasi terhadap pendekatan konvensional ini menunjukkan beberapa kelebihan, seperti kemudahan implementasi awal dan keterpaduan langsung dalam logika antarmuka. Namun, kekurangannya tidak dapat diabaikan: pertambahan aturan validasi membuat logika menjadi rumit, sulit diuji secara unit, serta mengurangi 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

MODUL 15
MODEL VIEW CONTROLLER (MVC)

15.1 Tujuan Materi Pembelajaran

1. Mahasiswa dapat mengenal tentang python turtle graphic
2. Mahasiswa mampu berpikir logis dalam hal pembangunan program
3. Mahasiswa mampu membuat bentuk menggunakan turtle graphic.

15.2 Materi Pembelajaran

14.4.1. ss

15.3 Rangkuman Materi

15.4 Latihan

DAFTAR PUSTAKA

Semua rujukan yang diacu di dalam artikel harus dicantumkan dalam daftar pustaka. Jumlah referensi yang direkomendasikan adalah 10 (sepuluh) atau lebih dalam 5 tahun terakhir. Penulisan rujukan di dalam teks dan daftar pustaka dianjurkan menggunakan program aplikasi manajemen referensi misalnya Mendeley, Zotero, EndNote, atau lainnya.

Gunakan gaya ***APA* (*American Psychological Association*)** atau ***APA Reference Style*** untuk penulisan rujukan dan daftar pustaka. Rujukan ditulis di dalam teks dengan mencantumkan nama penulis dan tahun (Vogelgesang dan Astin, 2000). Jika penulis lebih dari dua, maka hanya dituliskan nama penulis pertama diikuti "dkk" atau "et al." (Bezuidenhout et al., 2009). Contoh penulisan daftar Pustaka: https://www.mendeley.com/guides/apa-citation-guide

Rhoads, R. A. (1997). *Community service and higher learning: Explorations of the caring self*. New York: State University of New York Press.

Vogelgesang, L. J., & Astin, A. W. (2000). Comparing the effects of community service and service-learning. *Michigan Journal of Community Service Learning*, *7*(1), 25-34.

(*Times New Roman 12, Spacing: before 12 pt; after 12 pt; Line spacing: Multiple at 1 pt*)

GLOSARIUM

Glosarium adalah daftar istilah atau kata-kata khusus yang digunakan dalam suatu bidang ilmu, buku, atau dokumen tertentu, lengkap dengan definisinya. Glosarium biasanya disusun secara alfabetis untuk memudahkan pencarian dan membantu pembaca memahami istilah yang mungkin kurang familiar.

Contoh:

Naskah/manuskrip adalah kumpulan tulisan yang telah memenuhi kriteria tertentu sesuai dengan persyaratan penerbitan buku yang telah ditetapkan.

Penerbitan adalah suatu rangkaian kegiatan penyuntingan, produksi, promosi dan distribusi, yang bertujuan untuk menambah nilai suatu naskah/artikel sehingga menjadi terbitan (tercetak maupun elektronik) yang dapat tersampaikan dengan baik kepada pembaca sasaran.

Plagiat adalah perbuatan secara sengaja atau tidak sengaja dalam memperoleh atau mencoba memperoleh kredit atau nilai untuk suatu karya ilmiah, dengan mengutip sebagian atau seluruh karya dan/atau karya ilmiah pihak lain yang diakui sebagai karya ilmiahnya, tanpa menyatakan sumber secara tepat dan memadai.

Terbitan adalah *output* final dari proses penerbitan yang sudah layak dipublikasikan kepada masyarakat, baik secara elektronik maupun tercetak. Jenis terbitan dapat berupa buku, jurnal, bunga rampai, serta terbitan nonilmiah dan populer lainnya.

INDEKS

Indeks buku adalah daftar kata kunci, istilah, atau topik penting yang terdapat dalam sebuah buku, disusun secara alfabetis, dan biasanya diletakkan di bagian akhir buku. Indeks membantu pembaca menemukan informasi dengan cepat dengan menunjukkan nomor halaman tempat istilah tersebut dibahas.

Indeks berbeda dengan daftar isi. Daftar isi menunjukkan bab dan subbab dalam urutan halaman, sementara indeks mengelompokkan kata kunci secara alfabetis beserta lokasi (halaman) tempat kata tersebut dibahas.

Contoh:

A
Adaptasi, 45, 67, 120
Alga, 23, 78, 112
Anatomi tumbuhan, 56, 89

B
Bioteknologi, 34, 76, 150
Bioma, 90, 135
Biosfer, 12, 98

C
Clorofil, 44, 88
Ciri makhluk hidup, 5, 22

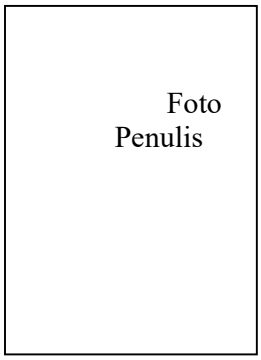
D
DNA, 40, 79, 143
Daur hidup, 60, 125

Pada contoh di atas, jika pembaca ingin mencari informasi tentang Bioteknologi, pembaca bisa langsung menuju halaman 34, 76, atau 150.

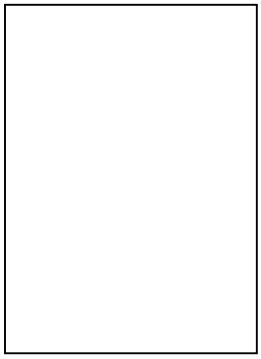
LAMPIRAN

Lampiran adalah bagian tambahan yang berisi informasi-informasi pendukung untuk melengkapi isi buku. Lampiran tidak wajib ada, tetapi lampiran digunakan untuk memberikan data tambahan. Contoh lampiran adalah grafik, tabel, atau materi pelengkap untuk membantu pemahaman materi.

BIOGRAFI PENULIS



(Nama Lengkap) adalah dosen Program Studi _____, Fakultas _____, Universitas Kristen Duta Wacana. Perempuan/Pria kelahiran _____ ini menyelesaikan Program Sarjana _____ di Universitas _____ tahun _____, lalu melanjutkan Program Pascasarjana _____ di Universitas _____ tahun _____, dan Program Doktor _____ di Universitas _____ tahun _____. (Dapat menuliskan karya buku yang pernah ditulis dalam 5 tahun terakhir). Penulis dapat dihubungi pada alamat email _____.



Antonius Rachmat Chrismanto adalah dosen Program Studi Informatika, Fakultas Teknologi Informasi, Universitas Kristen Duta Wacana. Pria kelahiran Purwokerto ini menyelesaikan Program Sarjana Teknik Informatika di Universitas Kristen Duta Wacana tahun 2024 lalu melanjutkan Program Pascasarjana Magister Ilmu Komputer di Universitas Gadjah Mada, lulus tahun 2008 dan Program Doktor Ilmu Komputer di Universitas Gajdah Mada tahun 2024. Penulis telah menulis 2 buku pemrograman dan berbagai publikasi ilmiah lain di jurnal internasional/nasional. Penulis dapat dihubungi pada alamat email anton@ti.ukdw.ac.id.

SINOPSIS

Sinopsis adalah ringkasan singkat yang memberikan gambaran tentang isi buku tanpa mengungkap keseluruhan cerita atau materi secara detail. Sinopsis biasanya terletak di bagian belakang sampul buku dan bertujuan untuk menarik minat pembaca agar ingin membaca lebih lanjut. Sinopsis terdiri dari 150-200 kata.

(JUDUL BUKU AJAR/REFERENSI)

Sinopsis Buku

Sinopsis adalah ringkasan singkat yang memberikan gambaran tentang isi buku tanpa mengungkap keseluruhan cerita atau materi secara detail. Sinopsis biasanya terletak di bagian belakang sampul buku dan bertujuan untuk menarik minat pembaca agar ingin membaca lebih lanjut. Sinopsis terdiri dari 150-200 kata.

DWUP
Duta Wacana University Press

Duta Wacana University Press
Jl. Dr. Wahidin Sudirohusodo No. 5-25, Yogyakarta
Telp: +62 274 563 929
Email: dwup@staff.ukdw.ac.id
Website: www. dwup.ukdw.ac.id

ISBN