CLEAN CODE DAN DESIGN PATTERN

TUGAS ANALISIS DESIGN PATTERN KASUS E-COMMERCE

Dosen Pengampu: Alif Finandhita, S.Kom, M.T

Disusun oleh:

10122119 Raditya Nalendra Utomo

10122145 Salman Faris Alhaitami

10122152 Muhammad Febriansyah

10122153 Salem Abdulah Al Oraifi

CCDP-1

PROGRAM STUDI TEKNIK INFORMATIKA

FAKULTAS TEKNIK DAN ILMU KOMPUTER

UNIVERSITAS KOMPUTER INDONESIA

2026

BAB I

PENJELASAN DOMAIN KASUS

1.1 Deskripsi Domain

Domain yang dipilih untuk tugas besar ini adalah Sistem Marketplace E-Commerce Terintegrasi. Sistem ini merupakan platform digital yang memfasilitasi transaksi jual-beli antara penyedia produk dan konsumen. Marketplace dipilih karena memiliki alur kerja yang dinamis dan melibatkan banyak entitas yang saling berinteraksi, mulai dari pengelolaan katalog produk, manajemen pengguna, hingga sistem transaksi yang melibatkan pihak ketiga.

1.2 Alur Bisnis Utama

Sistem ini mencakup beberapa proses bisnis inti, antara lain:

a. **Manajemen Katalog:** Pengelolaan data produk yang memiliki berbagai varian (warna, ukuran) dan kategori yang bertingkat.
b. **Proses Transaksi (Checkout):** Alur pemesanan yang kompleks, mulai dari validasi stok, pemilihan jasa pengiriman, hingga perhitungan biaya tambahan (asuransi/packing).
c. **Sistem Promosi:** Penerapan berbagai strategi diskon yang dapat berubah sewaktu-waktu sesuai kebijakan marketplace.
d. **Integrasi Pembayaran:** Penghubung antara sistem marketplace dengan berbagai penyedia layanan keuangan (Payment Gateway) melalui interface yang seragam.
e. **Pelacakan Pesanan:** Pemantauan perubahan status pesanan secara *real-time* hingga barang sampai ke tangan konsumen.

1.3 Alasan Pemilihan Domain

Pemilihan domain E-Commerce didasarkan pada tingkat kompleksitasnya yang sangat sesuai untuk implementasi berbagai jenis *Design Pattern*:

a. **Creational:** Diperlukan untuk menangani pembuatan objek pesanan yang rumit dan manajemen koneksi database yang efisien.
b. **Structural:** Diperlukan untuk menyatukan komponen sistem yang berbeda, seperti integrasi API eksternal dan struktur kategori produk yang bercabang.

c. **Behavioral:** Diperlukan untuk mengelola komunikasi antar objek, seperti perubahan status pesanan dan logika perhitungan diskon yang beragam.

Dengan menerapkan 12 *design pattern* pada domain ini, sistem yang dibangun diharapkan menjadi lebih modular, mudah dikelola (*maintainable*), dan fleksibel terhadap pengembangan fitur di masa depan.

BAB II

MAPPING DESIGN PATTERN

2.1 Tabel Mapping Keseluruhan

Berikut adalah ringkasan pemetaan *design pattern* berdasarkan kategori dan kasus penggunaannya:

Kategori	Design Pattern	Kasus Penggunaan (Mapping)
Creational	Singleton	Manajemen koneksi database pusat.
	Builder	Penyusunan objek pesanan (*Order*) yang kompleks.
	Prototype	Kloning varian produk (warna/ukuran) yang serupa.
	Abstract Factory	Pembuatan elemen UI untuk platform berbeda (Web/Mobile).
Structural	Adapter	Integrasi dengan API berbagai *Payment Gateway*.
	Decorator	Penambahan biaya layanan opsional (asuransi/packing).
	Facade	Penyederhanaan proses *checkout* multi-tahap.
	Composite	Struktur hierarki kategori dan sub-kategori produk.
Behavioral	Strategy	Logika perhitungan berbagai jenis diskon promo.
	Observer	Sistem notifikasi otomatis saat status pesanan berubah.
	State	Manajemen transisi status hidup pesanan (*Order Life Cycle*).
	Chain of Responsibility Pattern	Urutan validasi keamanan dan stok sebelum transaksi.

2.2 Penjelasan Singkat Mapping

Setelah mengidentifikasi daftar *design pattern* yang akan digunakan, bagian ini akan menjelaskan secara lebih mendalam mengenai rasio logis di balik pemilihan setiap pattern tersebut. Penjelasan dibagi ke dalam tiga kategori utama, yaitu *Creational*, *Structural*, dan *Behavioral*, untuk menunjukkan bagaimana masing-masing pola memberikan solusi spesifik terhadap permasalahan desain pada sistem *E-Commerce*.

A. Creational Patterns

Kategori *Creational Patterns* diterapkan untuk mengoptimalkan proses instansiasi objek di dalam sistem. Fokus utamanya adalah memastikan bahwa pembuatan objek seperti koneksi database, pesanan pelanggan, hingga elemen antarmuka dilakukan dengan cara yang paling efisien, terkontrol, dan mudah untuk dikembangkan di kemudian hari.

B. Structural Patterns

Kategori *Structural Patterns* difokuskan pada pengorganisasian hubungan antara kelas dan objek agar membentuk struktur aplikasi yang kokoh namun tetap fleksibel. Implementasi pada kategori ini bertujuan untuk menyatukan berbagai komponen yang berbeda, seperti integrasi layanan pihak ketiga dan pengelolaan struktur produk yang kompleks, menjadi satu kesatuan sistem yang harmonis.

C. Behavioral Patterns

Kategori *Behavioral Patterns* digunakan untuk mengatur pola komunikasi dan pembagian tanggung jawab antar objek di dalam sistem. Dengan pola ini, alur logika bisnis yang dinamis seperti perubahan status pesanan, sistem notifikasi, hingga perhitungan berbagai jenis diskon dapat dikelola secara terpisah sehingga meningkatkan modularitas dan keterbacaan kode.

BAB III

IMPLEMENTASI DESIGN PATTERN

3.1 Kelompok Creational Patterns

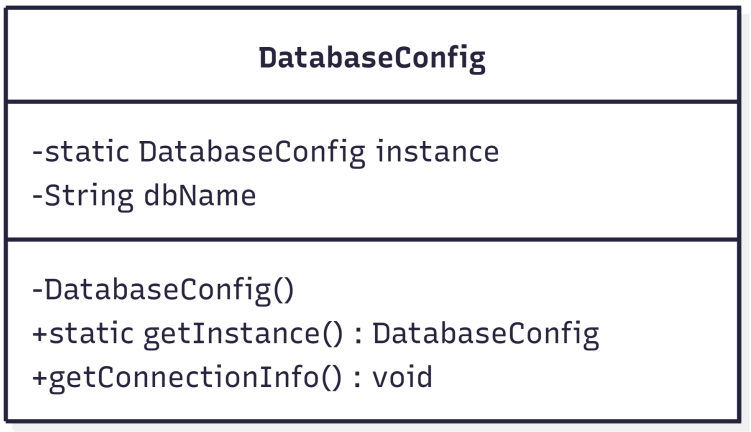
Kelompok *Creational Patterns* berfokus pada mekanisme pembuatan objek yang sesuai dengan situasi tertentu. Dalam sistem ini, kami mengimplementasikan empat pattern untuk memastikan manajemen memori yang efisien dan kemudahan dalam membangun objek yang kompleks.

3.1.1 Singleton Pattern (Database Connection)

A. Alasan Pemilihan

Dalam sistem Marketplace, akses ke database terjadi setiap detik. Jika setiap permintaan membuat koneksi baru, sumber daya server akan habis. Singleton digunakan untuk memastikan hanya ada satu instansi *Database Configuration* yang melayani seluruh aplikasi.

B. Class Diagram



C. Penjelasan Implementasi

Berikut adalah penjelasan mengenai implementasi singleton pattern:

1. Membuat kelas DatabaseConfig dengan atribut instance bertipe statis.
2. Mengatur *constructor* menjadi private agar tidak bisa diakses dengan kata kunci new.

3. Membuat method getInstance() yang memiliki pengecekan: jika instansi belum ada, maka buat baru; jika sudah ada, kembalikan yang lama.

D. Penjelasan Kode Program

Berikut adalah penjelasan mengenai kode programnya :

1. DatabaseConfig.java

```java
public class DatabaseConfig {
    // 1. Variabel static untuk menyimpan satu-satunya instansi
    private static DatabaseConfig instance;
    private String dbName;

    // 2. Constructor PRIVATE! Ini kunci agar tidak bisa di-"new" dari luar
    private DatabaseConfig() {
        this.dbName = "E-Commerce_Production_DB";
        System.out.println("--- Menginisialisasi Koneksi Database ---");
    }

    // 3. Method static untuk mendapatkan satu-satunya instansi
    public static synchronized DatabaseConfig getInstance() {
        if (instance == null) {
            instance = new DatabaseConfig();
        }
        return instance;
    }

    public void getConnectionInfo() {
        System.out.println("Terhubung ke: " + dbName);
    }
}
```

Kelas ini mengimplementasikan mekanisme *Lazy Initialization* untuk mengoptimalkan penggunaan sumber daya server. Inti dari implementasi ini adalah penggunaan akses kontrol private pada *constructor*, yang secara efektif menutup celah bagi kelas lain untuk membuat instansi baru secara ilegal. Aliran data dikontrol melalui metode statis getInstance() yang bertindak sebagai gerbang tunggal (*Single Point of Access*). Dengan pendekatan ini, sistem menjamin konsistensi data konfigurasi database di seluruh aplikasi dan mencegah terjadinya *memory leak* akibat penumpukan objek koneksi yang identik.

2. Main.java

```java
public class Main {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        // Mencoba mendapatkan instance pertama
        DatabaseConfig connection1 = DatabaseConfig.getInstance();
        connection1.getConnectionInfo();

        // Mencoba mendapatkan instance kedua
        DatabaseConfig connection2 = DatabaseConfig.getInstance();
        connection2.getConnectionInfo();

        // PEMBUKTIAN: Cek apakah alamat memori/objeknya sama
        if (connection1 == connection2) {
            System.out.println(x: "\nHasil: Objek connection1 dan connection2 IDENTIK.");
            System.out.println(x: "Singleton Pattern: BERHASIL.");
        } else {
            System.out.println(x: "\nHasil: Objek Berbeda.");
            System.out.println(x: "Singleton Pattern: GAGAL.");
        }
    }
}
```

File ini berperan sebagai *test bench* untuk memvalidasi integritas pola Singleton. Di dalamnya, dilakukan eksperimen dengan memanggil referensi objek beberapa kali dalam *thread* yang sama. Penjelasan dalam laporan harus menekankan pada perbandingan identitas objek (menggunakan operator `==` atau pengecekan *hashcode*). Hasil pengujian yang identik membuktikan bahwa pola ini berhasil mempertahankan satu *state* global bagi seluruh modul yang memerlukan akses database.

Hasil tes:

```
--- Menginisialisasi Koneksi Database ---
Terhubung ke: E-Commerce_Production_DB
Terhubung ke: E-Commerce_Production_DB

Hasil: Objek connection1 dan connection2 IDENTIK.
Singleton Pattern: BERHASIL.
```

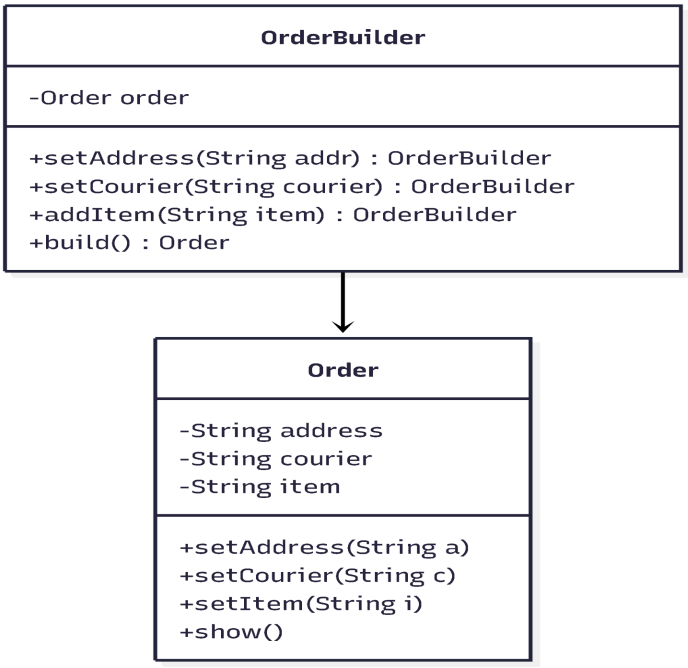
3.1.2 Builder Pattern (Order System)

A. Alasan Pemilihan

Pemilihan Builder Pattern didasari oleh kebutuhan untuk menyusun objek pesanan (Order) yang memiliki banyak atribut opsional dan kompleks seperti alamat lengkap, jenis kurir, serta daftar item yang beragam. Tanpa pola ini, kita akan terjebak dalam masalah *telescoping constructor* (constructor dengan terlalu banyak parameter) yang rawan kesalahan urutan input data. Dengan Builder, proses pembuatan pesanan

menjadi lebih terbaca karena dilakukan langkah demi langkah melalui *method chaining*, sehingga kode tetap bersih dan mudah dipelihara meskipun detail pesanan di masa depan bertambah kompleks.

B. Class Diagram



C. Penjelasan Implementasi

Berikut adalah penjelasan mengenai implementasi Builder Pattern::

1. Mendefinisikan Product: Membuat kelas Order sebagai objek akhir yang menyimpan informasi alamat, kurir, dan item.
2. Membuat Builder: Kelas OrderBuilder berfungsi untuk mengonstruksi objek Order langkah demi langkah.
3. Method Chaining: Setiap metode seperti setAddress() dan setCourier() mengembalikan instance OrderBuilder agar pemanggilan metode dapat dilakukan berantai.
4. Finalisasi: Metode build() dipanggil di akhir untuk menghasilkan objek Order yang sudah lengkap.

D. Penjelasan Kode Program

Berikut adalah penjelasan mengenai kode programnya :

1. Order.java

```java
package creational.builder;

public class Order {

    private String address;
    private String courier;
    private String item;

    public void setAddress(String a) { this.address = a; }
    public void setCourier(String c) { this.courier = c; }
    public void setItem(String i) { this.item = i; }

    public void show() {
        System.out.println(x: "--- Ringkasan Pesanan ---");
        System.out.println("Item    : " + item);
        System.out.println("Alamat  : " + address);
        System.out.println("Kurir   : " + courier);
        System.out.println(x: "-------------------------");
    }
}
```

File ini merupakan representasi dari objek akhir atau produk yang ingin diciptakan, yaitu sebuah invoice pesanan yang lengkap. Kodingan di dalam file ini fokus pada penyimpanan data (POJO) seperti alamat, kurir, dan item yang dipesan dengan akses privat untuk menjaga keamanan data (enkapsulasi), serta menyediakan metode show() yang berfungsi untuk merangkum dan menampilkan seluruh informasi pesanan tersebut ke layar dalam format yang rapi.

2. OrderBuilder.java

```java
package creational.builder;

public class OrderBuilder {

    private Order order = new Order();

    public OrderBuilder setAddress(String addr) {
        order.setAddress(addr);
        return this;
    }

    public OrderBuilder setCourier(String courier) {
        order.setCourier(courier);
        return this;
    }

    public OrderBuilder addItem(String item) {
        order.setItem(item);
        return this;
    }

    // Sesuai diagram: +build() : Order
    public Order build() {
        return this.order;
    }
}
```

File ini adalah mesin penggerak utama yang mengatur alur pembuatan objek pesanan secara bertahap dan fleksibel. Kodingan di dalamnya menggunakan teknik *method chaining*, di mana setiap metode seperti setAddress atau addItem akan mengembalikan instance dari dirinya sendiri, yang memungkinkan pengembang untuk menyusun pesanan dengan sintaks yang elegan dan mudah dibaca, serta diakhiri dengan metode build() yang merakit semua potongan informasi tersebut menjadi satu objek Order yang utuh.

3. Main.java

```java
package creational.builder;

import structural.decorator.*;
import behavioral.observer.*;

public class Main {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        // Dekorator Produk
        Product myProduct = new SimpleProduct();
        myProduct = new GiftWrap(myProduct);
        myProduct = new Insurance(myProduct);

        System.out.println("Produk Terpilih: " + myProduct.getDescription());
        System.out.println("Total Harga: Rp" + myProduct.getCost());

        // Builder Pesanan
        Order finalOrder = new OrderBuilder()
                .addItem(myProduct.getDescription())
                .setAddress(addr: "Jl. Sudirman No. 10, Jakarta")
                .setCourier(courier: "JNE Express")
                .build();

        finalOrder.show();

        // Observer Notifikasi
        StockManager stock = new StockManager();
        stock.addObserver(new NotificationSystem(name: "Admin Gudang"));
        stock.addObserver(new NotificationSystem(name: "Pembeli"));

        stock.setStockStatus(status: "Barang Sedang Dikirim");
    }
}
```

File ini bertindak sebagai titik masuk utama (*entry point*) program yang mengoordinasikan interaksi antara semua pola desain. Di dalam file ini, logika bisnis dijalankan dengan cara memanggil OrderBuilder untuk menciptakan pesanan, menggunakan dekorator untuk memodifikasi produk, dan menghubungkan StockManager dengan NotificationSystem, sehingga file ini berfungsi sebagai dirigen yang memastikan seluruh komponen kodingan bekerja sama secara harmonis.

Hasil tes:

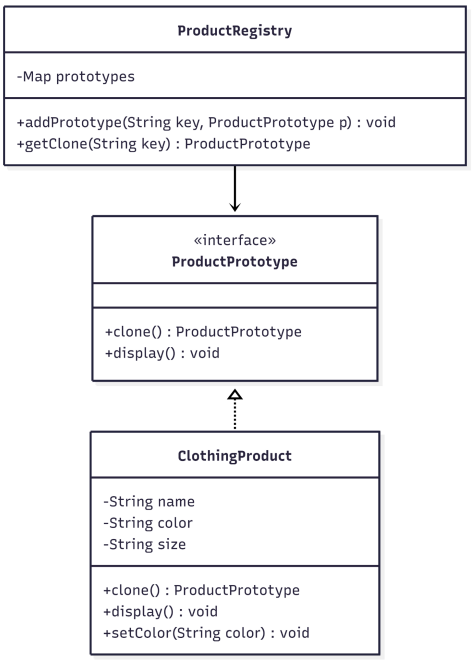
```
Produk Terpilih: Smartphone + Bungkus Kado + Asuransi
Total Harga: Rp5125000.0
--- Ringkasan Pesanan ---
Item    : Smartphone + Bungkus Kado + Asuransi
Alamat  : Jl. Sudirman No. 10, Jakarta
Kurir   : JNE Express
-----------------------

[StockManager] Perubahan status: Barang Sedang Dikirim
Notifikasi untuk Admin Gudang: Pesanan Anda saat ini Barang Sedang Dikirim
Notifikasi untuk Pembeli: Pesanan Anda saat ini Barang Sedang Dikirim
```

3.1.3 Prototype Pattern (Product Variation)

A. Alasan Pemilihan

Dalam domain e-commerce, satu model produk (misal: "Kaos Polos") seringkali memiliki puluhan varian warna dan ukuran. Menginisialisasi setiap varian menggunakan perintah new secara manual tidak efisien jika data produk tersebut harus diambil kembali dari database yang berat. Prototype dipilih agar sistem cukup mengambil satu "*objek master*" dan melakukan kloning memori untuk membuat varian lainnya, sehingga menghemat resource dan waktu eksekusi.

B. Class Diagram

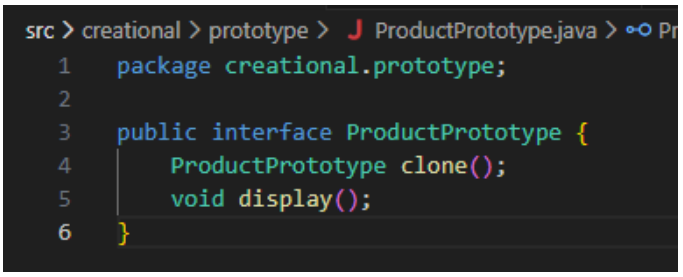


C. Penjelasan Implementasi

1. **Interface** ProductPrototype: Mendefinisikan kontrak metode clone().
2. **Concrete Class** ClothingProduct: Mengimplementasikan proses penyalinan objek itu sendiri.
3. **Registry** ProductRegistry: Bertindak sebagai gudang penyimpanan objek-objek master. Saat sistem membutuhkan varian baru, ia hanya perlu memanggil getClone() dari registry tanpa perlu tahu cara pembuatan objek dari nol.

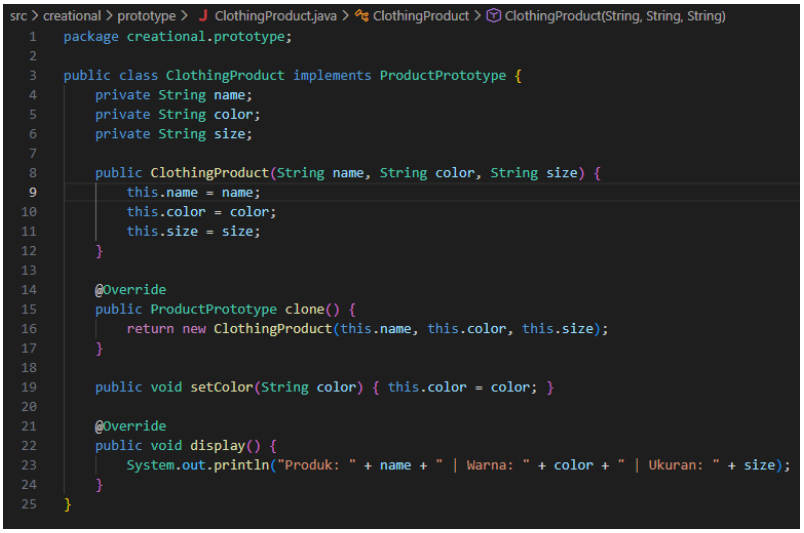
D. Penjelasan Kode Program

1. ProductPrototype.java



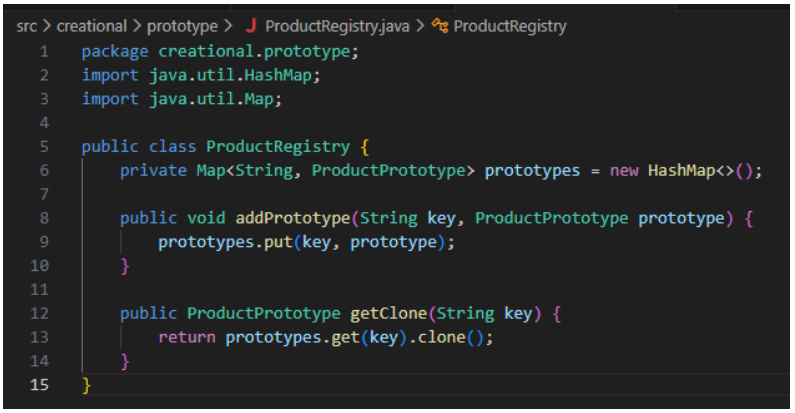
Kelas ini bertindak sebagai kontrak fundamental yang mendefinisikan kemampuan objek untuk mereplikasi dirinya sendiri. Melalui metode clone(), interface ini memastikan bahwa setiap entitas produk dalam sistem *e-commerce* memiliki standarisasi mekanisme penyalinan objek tanpa harus mengekspos detail implementasi kelas konkretnya.

2. ClothingProduct.java



Kelas ini merepresentasikan objek produk fisik (pakaian) yang menyimpan data spesifik seperti nama, warna, dan ukuran. Implementasi metode clone() di kelas ini memungkinkan pembuatan entitas baru berdasarkan status objek saat ini (*current state*). Hal ini sangat efektif untuk skenario pembuatan varian produk, di mana sistem hanya perlu mengubah atribut kecil (seperti warna) tanpa melakukan proses instansiasi ulang yang memakan beban komputasi tinggi.

3. ProductRegistry.java



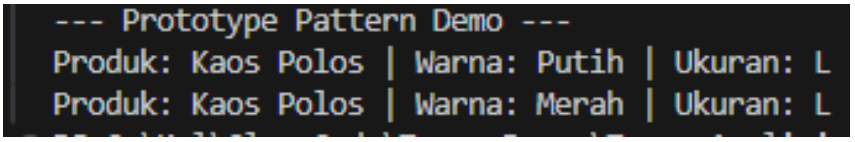
Berperan sebagai gudang penyimpanan (*cache*) bagi objek-objek template atau master. Kelas ini meminimalkan ketergantungan sistem terhadap database dengan menyediakan mekanisme pencarian objek master yang siap di kloning kapan saja. Dengan menggunakan ProductRegistry, aplikasi dapat menghindari

redudansi data di memori dan mempercepat proses penyediaan objek varian produk kepada pengguna.

4. PrototypeDemo.java

Kelas ini berfungsi sebagai pengatur alur simulasi pembuatan varian produk tanpa proses instansiasi manual yang repetitif. Di dalam kelas ini, sistem mendemonstrasikan efisiensi penggunaan memori dengan mendaftarkan satu objek *master* ke dalam ProductRegistry dan memproduksi berbagai varian produk melalui mekanisme kloning. Melalui output yang dihasilkan, kelas ini membuktikan bahwa perubahan pada atribut objek kloning (seperti warna atau ukuran) tidak akan mengganggu integritas data pada objek master, yang merupakan inti dari keunggulan pola *Prototype*.

Hasil Tes :

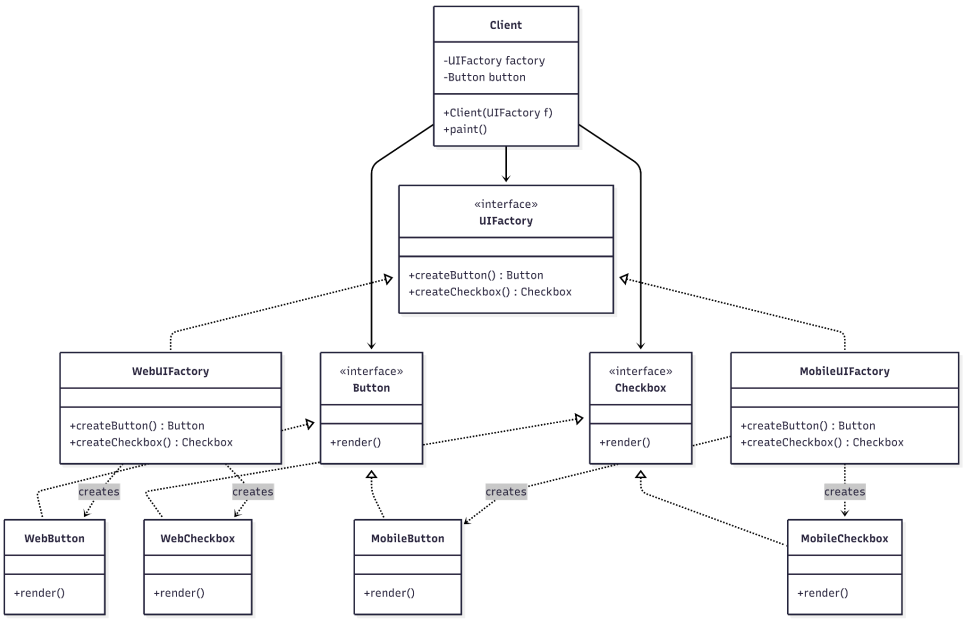


3.1.4 Abstract Factory Pattern (UI Theme)

A. Alasan Pemilihan

Dalam pengembangan aplikasi multi-platform, seringkali kita perlu menampilkan elemen antarmuka (seperti Button dan Checkbox) yang memiliki fungsi sama tetapi tampilan berbeda sesuai OS. Jika menggunakan instansiasi langsung (new), kode akan penuh dengan percabangan if-else platform. Abstract Factory dipilih untuk menciptakan keluarga objek yang konsisten (Web atau Mobile) tanpa membiarkan Client terikat pada kelas konkret, sehingga menambah platform baru di masa depan menjadi jauh lebih mudah.

B. Class Diagram



C. Penjelasan Implementasi

Berikut adalah penjelasan mengenai implementasi Abstract Factory pattern:

1. Mendefinisikan Abstract Product: Membuat interface Button dan Checkbox sebagai kontrak dasar untuk semua jenis komponen UI.
2. Implementasi Concrete Product: Membuat kelas spesifik seperti WebButton dan MobileButton yang mengimplementasikan logika render sesuai platformnya.
3. Membuat Abstract Factory: Membuat interface UIFactory yang mendeklarasikan metode untuk menciptakan keluarga produk (button dan checkbox).
4. Implementasi Concrete Factory: Membuat WebUIFactory dan MobileUIFactory untuk memproduksi objek yang konsisten dengan tema platform tertentu.
5. Interaksi Client: Kelas Client hanya bergantung pada interface factory, sehingga penambahan platform baru tidak akan mengubah kode pada sisi client.

D. Penjelasan Kode Program

1. Button

```
public interface Button {
    void render();
}
```

File ini merupakan interface yang berperan sebagai kontrak dasar untuk setiap objek tombol (*Button*) dalam sistem. Kode di dalamnya hanya mendefinisikan method render() tanpa implementasi, sehingga setiap kelas yang mengimplementasikan interface ini wajib menyediakan cara masing-masing untuk menampilkan tombol sesuai kebutuhan. Dengan adanya interface ini, sistem menjadi lebih fleksibel dan mudah dikembangkan karena berbagai jenis tombol dapat dibuat dengan perilaku render yang berbeda tanpa mengubah struktur utama program.

2. Checkbox

```
public interface Checkbox {
    void render();
}
```

File ini merupakan interface yang berfungsi sebagai kontrak dasar untuk komponen *Checkbox* dalam sistem. Kode di dalamnya mendefinisikan method render() tanpa implementasi, sehingga setiap kelas yang mengimplementasikan interface ini wajib menyediakan cara masing-masing untuk menampilkan checkbox sesuai platform atau kebutuhan desain. Struktur ini membantu menjaga konsistensi antar komponen sekaligus membuat kode lebih fleksibel, karena berbagai variasi checkbox dapat dibuat tanpa mengubah bagian program yang lain.

3. MobileButton

```
public class MobileButton implements Button {
    @Override
    public void render() {
        System.out.println(x: "Rendering Button gaya Mobile.");
    }
}
```

File ini merupakan kelas konkret yang mengimplementasikan interface Button dan merepresentasikan tombol dengan tampilan khusus untuk platform mobile. Kode di

dalamnya mengimplementasikan method render() dengan cara menampilkan pesan *"Rendering Button gaya Mobile."*, yang menunjukkan bagaimana tombol dirender sesuai gaya perangkat mobile. Dengan struktur ini, sistem dapat memiliki berbagai variasi tombol dengan implementasi render yang berbeda tanpa mengubah kontrak dasar pada interface Button, sehingga desain menjadi lebih fleksibel dan mudah dikembangkan.

4. MobileCheckbox

```
public class MobileCheckbox implements Checkbox {
    @Override
    public void render() {
        System.out.println(x: "Rendering Checkbox gaya Mobile.");
    }
}
```

File ini merupakan kelas konkret yang mengimplementasikan interface Checkbox dan merepresentasikan komponen checkbox dengan tampilan khusus untuk platform mobile. Kode di dalamnya mengimplementasikan method render() dengan cara menampilkan pesan *"Rendering Checkbox gaya Mobile."*, yang menunjukkan bagaimana checkbox dirender sesuai gaya perangkat mobile. Dengan pendekatan ini, sistem dapat menyediakan berbagai variasi checkbox dengan implementasi berbeda tanpa mengubah kontrak dasar pada interface Checkbox, sehingga struktur program menjadi lebih fleksibel dan mudah dikembangkan.

5. MobileUIFactory

```
public class MobileUIFactory implements UIFactory {
    @Override
    public Button createButton() {
        return new MobileButton();
    }

    @Override
    public Checkbox createCheckbox() {
        return new MobileCheckbox();
    }
}
```

File ini merupakan factory konkret dalam pola *Abstract Factory* yang bertugas membuat sekumpulan komponen UI khusus untuk platform mobile. Kode di dalamnya mengimplementasikan interface UIFactory dengan menyediakan method createButton() dan createCheckbox() yang masing-masing mengembalikan objek MobileButton dan MobileCheckbox. Dengan struktur ini, pembuatan komponen UI menjadi terpusat dan konsisten sesuai tema mobile, sehingga client dapat membuat berbagai elemen antarmuka tanpa perlu mengetahui detail kelas konkret yang digunakan.

6. UIFactory

```
1  public interface UIFactory {
2      Button createButton();
3      Checkbox createCheckbox();
4  }
```

File ini merupakan interface factory dalam pola *Abstract Factory* yang berfungsi sebagai kontrak untuk membuat keluarga komponen UI yang saling terkait. Kode di dalamnya mendefinisikan dua method, yaitu createButton() dan createCheckbox(), tanpa implementasi langsung. Setiap kelas factory konkret yang mengimplementasikan interface ini wajib menyediakan cara pembuatan objek Button dan Checkbox sesuai tema atau platform tertentu. Dengan struktur ini, proses pembuatan komponen menjadi lebih terorganisir, konsisten, serta memudahkan penggantian tema UI tanpa mengubah kode pada sisi client.

7. WebButton

```
1  public class WebButton implements Button {
2      @Override
3      public void render() {
4          System.out.println(x: "Rendering Button gaya Web.");
5      }
6  }
```

File ini merupakan kelas konkret yang mengimplementasikan interface Button dan merepresentasikan tombol dengan tampilan khusus untuk platform web. Kode di

dalamnya mengimplementasikan method render() dengan cara menampilkan pesan *"Rendering Button gaya Web."*, yang menunjukkan bagaimana tombol dirender sesuai gaya antarmuka web. Dengan pendekatan ini, sistem dapat menyediakan variasi tombol berbeda berdasarkan platform tanpa mengubah kontrak dasar pada interface Button, sehingga desain menjadi lebih fleksibel dan mudah dikembangkan.

8. WebCheckbox

```java
public class WebCheckbox implements Checkbox {
    @Override
    public void render() {
        System.out.println(x: "Rendering Checkbox gaya Web.");
    }
}
```

File ini merupakan kelas konkret yang mengimplementasikan interface Checkbox dan merepresentasikan komponen checkbox dengan tampilan khusus untuk platform web. Kode di dalamnya mengimplementasikan method render() dengan menampilkan pesan *"Rendering Checkbox gaya Web."*, yang menunjukkan bagaimana checkbox dirender sesuai gaya antarmuka web. Dengan pendekatan ini, sistem dapat menyediakan variasi checkbox untuk berbagai platform tanpa mengubah kontrak dasar pada interface Checkbox, sehingga mendukung fleksibilitas dan konsistensi dalam penerapan pola *Abstract Factory*.

9. WebUIFactory

```java
public class WebUIFactory implements UIFactory {
    @Override
    public Button createButton() {
        return new WebButton();
    }

    @Override
    public Checkbox createCheckbox() {
        return new WebCheckbox();
    }
}
```

File ini merupakan factory konkret dalam pola *Abstract Factory* yang bertugas membuat sekumpulan komponen UI khusus untuk platform web. Kode di dalamnya mengimplementasikan interface UIFactory dengan menyediakan method createButton() dan createCheckbox() yang masing-masing mengembalikan objek WebButton dan WebCheckbox. Dengan struktur ini, proses pembuatan komponen UI menjadi terpusat sesuai tema web, sehingga client dapat membuat berbagai elemen antarmuka tanpa perlu mengetahui detail kelas konkret yang digunakan, serta memudahkan penggantian tema atau platform secara fleksibel.

10. Client.java

```
public class Client {
    private Button button;
    private Checkbox checkbox;

    public Client(UIFactory factory) {
        this.button = factory.createButton();
        this.checkbox = factory.createCheckbox();
    }

    public void paint() {
        button.render();
        checkbox.render();
    }
}
```

File ini merupakan kelas client dalam pola *Abstract Factory* yang berfungsi menggunakan objek-objek UI tanpa bergantung langsung pada kelas konkret tertentu. Kode di dalamnya menerima objek UIFactory melalui konstruktor, lalu membuat komponen Button dan Checkbox menggunakan method createButton() dan createCheckbox(). Pada method paint(), client hanya memanggil render() dari kedua komponen tersebut tanpa mengetahui apakah yang digunakan adalah versi Web atau Mobile. Dengan pendekatan ini, client menjadi lebih fleksibel dan mudah dikembangkan karena perubahan platform

cukup dilakukan pada factory yang diberikan tanpa perlu mengubah logika di dalam kelas client.

11. Main.java

```java
public class Main {
    Run | Debug
    public static void main(String[] args) {
        // Jika ingin tampilan Web
        UIFactory factory = new MobileUIFactory();
        Client app = new Client(factory);

        System.out.println(x: "Aplikasi dijalankan:");
        app.paint();
    }
}
```

File ini merupakan kelas utama (entry point) dalam implementasi pola *Abstract Factory* yang berfungsi untuk menentukan jenis factory yang akan digunakan serta menjalankan aplikasi. Kode di dalamnya membuat objek UIFactory menggunakan MobileUIFactory, lalu mengirimkannya ke kelas Client sehingga seluruh komponen UI yang dibuat mengikuti gaya mobile. Setelah itu, method paint() dipanggil untuk merender komponen Button dan Checkbox. Dengan struktur ini, pergantian platform (misalnya dari Mobile ke Web) cukup dilakukan dengan mengganti jenis factory yang diinisialisasi tanpa mengubah logika pada kelas client maupun komponen UI lainnya.

3.2 Kelompok Structural Patterns

Kelompok *Structural Patterns* membahas bagaimana kelas dan objek disusun untuk membentuk struktur yang lebih besar dan fleksibel. Kami menggunakan empat pattern untuk menyederhanakan hubungan antar komponen yang berbeda di dalam marketplace.

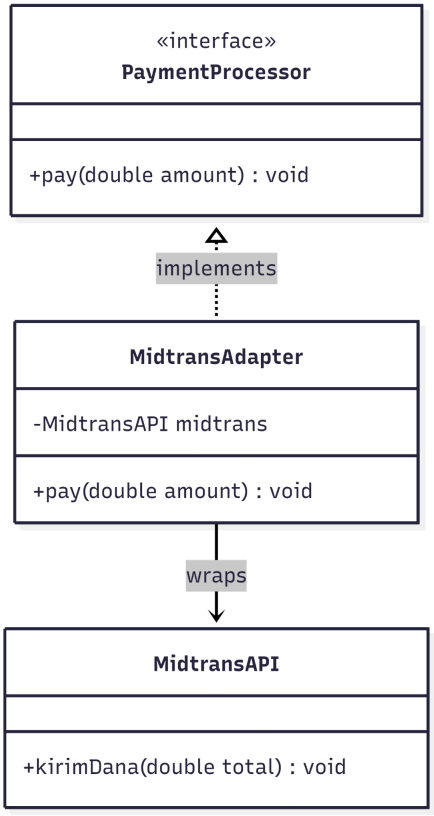
3.2.1 Adapter Pattern (Payment Gateway)

A. Alasan Pemilihan

Marketplace perlu terhubung dengan berbagai vendor pembayaran. Karena setiap vendor memiliki API yang berbeda, Adapter digunakan

untuk membungkus API pihak ketiga tersebut agar selaras dengan interface standar yang dimiliki sistem kami.

B. Class Diagram



C. Penjelasan Implementasi

Berikut adalah penjelasan implementasi adapter pattern:

1. Target Interface: Membuat interface PaymentProcessor sebagai standar.
2. Adaptee: Menyiapkan kelas MidtransAPI dengan metode asli kirimDana().
3. Adapter: Membuat MidtransAdapter sebagai penerjemah antar kedua komponen.

D. Penjelasan Kode Program

Berikut adalah penjelasan mengenai kode programnya :

1. PaymentProcessor.java

```java
package structural.adapter;

public interface PaymentProcessor {
    // Sistem hanya mau tahu cara manggil "pay"
    void pay(double amount);
}
```

Sebagai *Target Interface*, file ini berfungsi sebagai lapisan abstraksi yang menetapkan standar operasi pembayaran dalam domain marketplace kita. Dengan hanya mendefinisikan metode pay(double amount), interface ini mengisolasi logika bisnis inti dari detail teknis vendor pembayaran. Ini adalah penerapan prinsip *Dependency Inversion*, di mana sistem tingkat tinggi tidak bergantung pada detail teknis tingkat rendah (API Bank).

2. MidtransAPI.java

```java
package structural.adapter;

public class MidtransAPI {
    public void kirimDana(double total) {
        System.out.println("Pembayaran diproses oleh Midtrans API sebesar: Rp" + total);
    }
}
```

File ini mensimulasikan komponen *Legacy* atau library pihak ketiga yang memiliki kontrak interface berbeda. Penggunaan metode kirimDana(double total) menunjukkan adanya diskrepansi penamaan dan struktur parameter. Dalam skenario nyata, file ini seringkali bersifat *read-only* atau tidak boleh dimodifikasi, sehingga memerlukan perantara agar bisa berkomunikasi dengan sistem baru.

3. MidtransAdapter.java

```java
package structural.adapter;

public class MidtransAdapter implements PaymentProcessor {
    private MidtransAPI midtrans;

    public MidtransAdapter(MidtransAPI midtrans) {
        this.midtrans = midtrans;
    }

    @Override
    public void pay(double amount) {
        // Menerjemahkan 'pay' ke fungsi 'kirimDana'
        midtrans.kirimDana(amount);
    }
}
```

Merupakan kelas krusial yang menjembatani ketidakcocokan antara sistem internal dan API eksternal. Dengan mengimplementasikan PaymentProcessor, kelas ini mampu menyamar sebagai komponen standar sistem. Di dalamnya, terjadi proses delegasi di mana pemanggilan metode pay() secara internal diterjemahkan menjadi pemanggilan kirimDana(). Struktur ini memberikan fleksibilitas luar biasa; jika di masa depan marketplace berpindah vendor, pengembang cukup membuat Adapter baru tanpa menyentuh kode transaksi yang sudah ada.

4. Main.java

```java
package structural.adapter;

public class Main {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        // 1. Inisialisasi API pihak ketiga
        MidtransAPI apiLuar = new MidtransAPI();

        // 2. Masukkan ke dalam adapter agar bisa dikenali sistem
        PaymentProcessor payment = new MidtransAdapter(apiLuar);

        // 3. Jalankan pembayaran (sistem tidak perlu tahu ada Midtrans di dalamnya)
        System.out.println(x: "--- Memulai Proses Check Out ---");
        payment.pay(amount: 250000.0);

        System.out.println(x: "\nAdapter Berhasil: Sistem tetap menggunakan method 'pay()'");
    }
}
```

Unit testing ini mendemonstrasikan prinsip *Polymorphism*. Di sini, kita menunjukkan bahwa objek MidtransAdapter dapat diperlakukan sebagai PaymentProcessor. Penjelasan di laporan harus menyoroti betapa bersihnya kode pada sisi klien (Main),

karena klien tidak perlu mengetahui kerumitan integrasi Midtrans yang ada di balik layar.

Hasil tes:

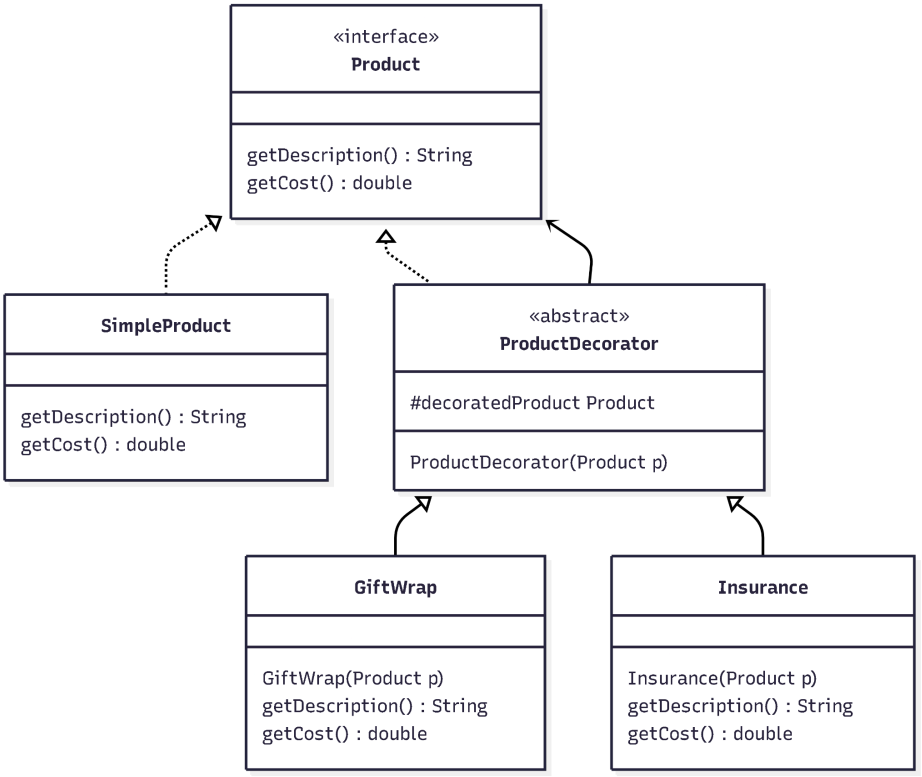
```
--- Memulai Proses Check Out ---
Pembayaran diproses oleh Midtrans API sebesar: Rp250000.0

Adapter Berhasil: Sistem tetap menggunakan method 'pay()'
```

3.2.2 Decorator Pattern (Add-on Service)

A. Alasan Pemilihan

Decorator Pattern dipilih untuk menangani penambahan fitur tambahan pada produk secara dinamis tanpa harus mengubah struktur kelas inti atau menggunakan pewarisan yang kaku. Dalam e-commerce, fitur seperti bungkus kado atau asuransi seringkali bersifat opsional dan bisa dikombinasikan satu sama lain. Pola ini memungkinkan kita untuk "membungkus" objek produk dasar dengan berbagai fitur tambahan kapan saja saat *runtime*, sehingga kita terhindar dari pembuatan sub-class yang terlalu banyak (misal:Smartphone Kado,SmartphoneAsuransi, SmartphoneKadoAsuransi) dan menjaga prinsip *Open-Closed Principle*.

B. Class Diagram



C. Penjelasan Implementasi

Berikut adalah penjelasan mengenai implementasi Decorator Pattern :

1. Mendefinisikan Interface: Membuat interface Product sebagai kontrak dasar untuk deskripsi dan harga produk.
2. Implementasi Concrete Product: Membuat SimpleProduct sebagai produk dasar (misalnya Smartphone).
3. Membuat Abstract Decorator: Kelas ProductDecorator menyimpan referensi ke objek Product dan bertindak sebagai pembungkus (*wrapper*).
4. Implementasi Concrete Decorator: Membuat GiftWrap dan Insurance yang menambahkan logika tambahan (deskripsi baru dan harga tambahan) di atas produk yang dibungkus.

D. Penjelasan Kode Program

Berikut adalah penjelasan mengenai kode programnya :

1. Product.java

```java
package structural.decorator;
public interface Product {
    String getDescription();
    double getCost();
}
```

File ini adalah sebuah *interface* dasar yang menetapkan identitas utama bagi setiap produk dalam aplikasi. Kodingan di sini sangat krusial karena ia mewajibkan setiap produk, baik yang polos maupun yang sudah dimodifikasi, untuk memiliki metode getDescription() dan getCost(), yang menjadi fondasi bagi pola Decorator untuk menumpuk fitur-fitur baru di atas produk standar secara konsisten.

2. SimpleProduct.java

```java
package structural.decorator;
public class SimpleProduct implements Product {
    public String getDescription() { return "Smartphone"; }
    public double getCost() { return 5000000; }
}
```

File ini berisi implementasi dari produk paling dasar atau "produk mentah" sebelum diberikan tambahan apa pun, seperti sebuah Smartphone standar. Kodingan ini sangat sederhana karena ia hanya mengembalikan nilai harga dan deskripsi awal, yang nantinya akan menjadi objek inti yang "dibungkus" oleh berbagai tambahan fitur lain di dalam proses dekorasi.

3. ProductDecorator.java

```java
package structural.decorator;
public abstract class ProductDecorator implements Product {
    protected Product decoratedProduct;
    public ProductDecorator(Product p) { this.decoratedProduct = p; }
}
```

File ini berbentuk kelas abstrak yang menjadi jembatan utama dalam mekanisme dekorasi produk. Kodingan di dalamnya menyimpan referensi ke objek Product lain (komposisi), sehingga ia bisa meneruskan perintah ke produk aslinya sambil memberikan ruang bagi kelas turunannya untuk memodifikasi atau menambah perilaku baru pada deskripsi dan harga tanpa merusak struktur objek aslinya.

4. GiftWrap.java

```
package structural.decorator;
public class GiftWrap extends ProductDecorator {
    public GiftWrap(Product p) { super(p); }
    public String getDescription() { return decoratedProduct.getDescription() + " + Bungkus Kado"; }
    public double getCost() { return decoratedProduct.getCost() + 25000; }
}
```

File ini adalah dekorator konkret yang secara khusus menangani fitur pembungkusan kado. Kodingan di dalamnya bekerja dengan cara mengambil deskripsi dari produk yang dibungkus lalu menambahkan teks " + Bungkus Kado", serta mengambil harga produk lama dan menjumlahkannya dengan biaya tambahan kado, sehingga memberikan fleksibilitas tinggi bagi pelanggan untuk menambah layanan ini secara dinamis.

5. Insurance.java

```
package structural.decorator;
public class Insurance extends ProductDecorator {
    public Insurance(Product p) { super(p); }
    public String getDescription() { return decoratedProduct.getDescription() + " + Asuransi"; }
    public double getCost() { return decoratedProduct.getCost() + 100000; }
}
```

File ini memiliki logika yang mirip dengan dekorator kado, namun fokus pada penambahan layanan perlindungan atau asuransi. Di dalam file ini, sistem akan memodifikasi objek produk dengan menyisipkan informasi asuransi pada deskripsi dan mengakumulasikan premi asuransi ke dalam total biaya akhir, yang menunjukkan betapa mudahnya menambah fitur baru pada produk melalui mekanisme dekorasi berlapis.

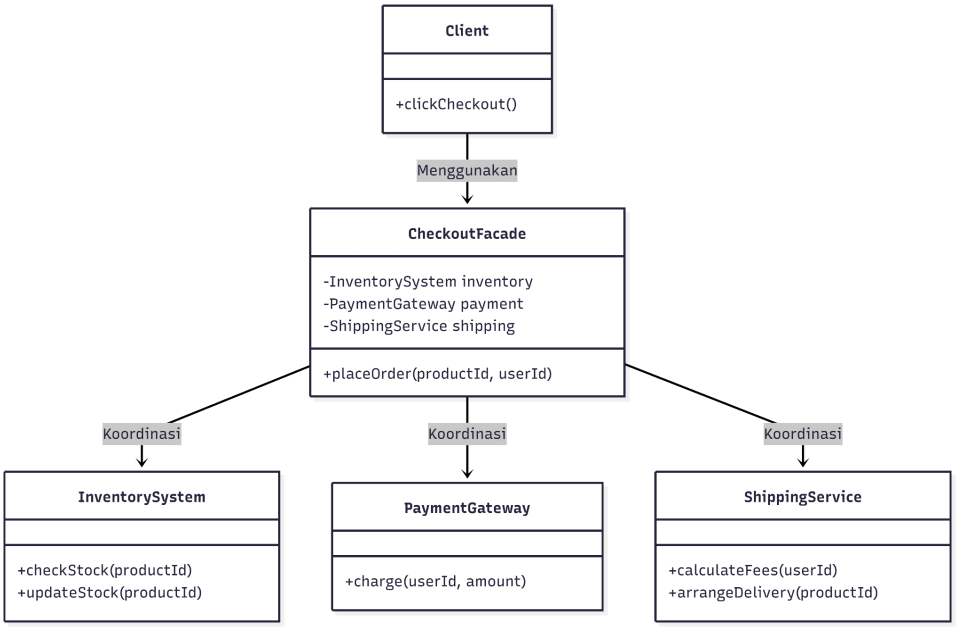
3.2.3 Facade Pattern (Checkout System)

A. Alasan Pemilihan

Proses checkout melibatkan banyak sistem yang rumit, mulai dari pengecekan stok di Inventory, penagihan di PaymentGateway, hingga pengaturan logistik di ShippingService. Jika Client harus memanggil satu per satu, tingkat ketergantungan (*coupling*) menjadi sangat tinggi dan rawan kesalahan urutan. Facade digunakan untuk membungkus

kompleksitas tersebut ke dalam satu pintu utama (CheckoutFacade), sehingga Client cukup memanggil satu metode sederhana untuk menyelesaikan seluruh proses.

B. Class Diagram



C. Penjelasan Implementasi

Berikut adalah penjelasan mengenai implementasi Facade pattern:

1. Identifikasi Subsystem: Membagi logika bisnis yang kompleks ke dalam kelas-kelas terpisah seperti InventorySystem, PaymentGateway, dan ShippingService.
2. Membuat Kelas Facade: Membuat kelas CheckoutFacade yang menyimpan referensi ke semua objek subsistem tersebut.
3. Penyederhanaan Antarmuka: Menyediakan metode tunggal placeOrder() di dalam Facade yang mengoordinasikan urutan panggilan antar subsistem secara internal.
4. Abstraksi Client: Kelas Client hanya perlu memanggil satu metode dari Facade tanpa perlu tahu detail kerumitan komunikasi antar subsistem.

D. Penjelasan Kode Program

1. CheckoutFacade

```java
package Structural.facade;

public class CheckoutFacade {
    private InventorySystem inventory = new InventorySystem();
    private PaymentGateway payment = new PaymentGateway();
    private ShippingService shipping = new ShippingService();

    public void placeOrder(String productId, String userId) {
        inventory.checkStock(productId);
        payment.charge(userId, amount: 50000);
        shipping.calculateFees(userId);
        inventory.updateStock(productId);
        shipping.arrangeDelivery(productId);
    }
}
```

File ini merupakan kelas Facade dalam pola *Facade Pattern* yang berfungsi menyederhanakan proses pemesanan dengan menyatukan beberapa subsystem menjadi satu titik akses. Kode di dalamnya mengelola objek seperti InventorySystem, PaymentGateway, dan ShippingService, lalu menyediakan method placeOrder() sebagai antarmuka tunggal bagi client. Saat method ini dipanggil, sistem secara otomatis menjalankan serangkaian proses seperti pengecekan stok, pembayaran, perhitungan biaya pengiriman, pembaruan stok, hingga pengaturan pengiriman. Dengan pendekatan ini, client tidak perlu berinteraksi langsung dengan banyak kelas yang kompleks, sehingga penggunaan sistem menjadi lebih mudah, rapi, dan terorganisir.

2. InventorySystem

```java
package Structural.facade;

public class InventorySystem {
    public void checkStock(String productId) {
        System.out.println("Mengecek stok untuk: " + productId);
    }
    public void updateStock(String productId) {
        System.out.println(x: "Stok diperbarui.");
    }
}
```

File ini merupakan subsystem dalam pola *Facade Pattern* yang bertanggung jawab mengelola data inventori atau stok barang. Kode di dalamnya menyediakan method checkStock() untuk melakukan pengecekan ketersediaan produk berdasarkan productId, serta method updateStock() untuk

memperbarui jumlah stok setelah proses pemesanan dilakukan. Kelas ini tidak berinteraksi langsung dengan client, melainkan dipanggil melalui kelas *Facade* sehingga proses manajemen stok dapat berjalan sebagai bagian dari alur pemesanan yang lebih sederhana dan terorganisir.

3. PaymentGateway

```java
package Structural.facade;

public class PaymentGateway {
    public void charge(String userId, double amount) {
        System.out.println("Menagih Rp" + amount + " ke " + userId);
    }
}
```

File ini merupakan subsystem dalam pola *Facade Pattern* yang bertanggung jawab menangani proses pembayaran pada sistem pemesanan. Kode di dalamnya menyediakan method charge() yang digunakan untuk melakukan penagihan kepada pengguna berdasarkan userId dan jumlah pembayaran (amount). Kelas ini tidak dipanggil langsung oleh client, melainkan diakses melalui kelas *Facade* sehingga proses pembayaran dapat terintegrasi dengan alur checkout secara lebih sederhana, terstruktur, dan mudah digunakan.

4. ShippingService

```java
package Structural.facade;

public class ShippingService {
    public void calculateFees(String userId) {
        System.out.println(x: "Menghitung ongkir...");
    }
    public void arrangeDelivery(String productId) {
        System.out.println(x: "Pesanan dikirim!");
    }
}
```

File ini merupakan subsystem dalam pola *Facade Pattern* yang bertanggung jawab menangani proses pengiriman pada sistem pemesanan. Kode di dalamnya menyediakan method calculateFees() untuk menghitung biaya pengiriman berdasarkan userId, serta method arrangeDelivery() untuk mengatur proses pengiriman produk menggunakan productId. Kelas ini biasanya tidak dipanggil langsung oleh client,

melainkan diakses melalui kelas *Facade*, sehingga seluruh proses logistik dapat dijalankan secara terkoordinasi dan lebih sederhana dalam satu alur checkout.

5. Client.java

```java
package Structural.facade;

public class Client {
    public void clickCheckout() {
        CheckoutFacade facade = new CheckoutFacade();
        facade.placeOrder(productId: "Barang-A", userId: "User-01");
    }
}
```

File ini merupakan kelas client dalam pola *Facade Pattern* yang berfungsi sebagai pemicu proses checkout dari sisi pengguna. Kode di dalamnya membuat objek CheckoutFacade lalu memanggil method placeOrder() dengan parameter productId dan userId. Dengan pendekatan ini, client tidak perlu berinteraksi langsung dengan subsystem seperti InventorySystem, PaymentGateway, atau ShippingService, karena seluruh proses kompleks sudah disederhanakan melalui satu antarmuka facade. Hal ini membuat kode client menjadi lebih ringkas, mudah dipahami, dan tidak bergantung pada detail implementasi di balik layar.

6. Main.java

```java
package Structural.facade;

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Client client = new Client();
        client.clickCheckout();
    }
}
```

File ini merupakan kelas utama (entry point) dalam implementasi pola *Facade Pattern* yang berfungsi menjalankan aplikasi dan memulai proses checkout. Kode di dalamnya membuat objek Client lalu memanggil method clickCheckout(), yang secara tidak langsung akan menggunakan CheckoutFacade untuk menjalankan seluruh alur pemesanan. Dengan struktur ini,

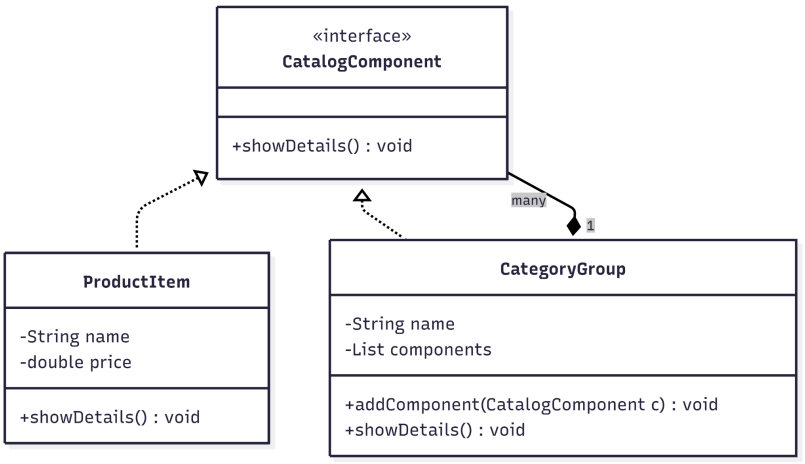
kelas Main hanya bertugas sebagai pemicu awal program, sementara detail proses kompleks tetap disederhanakan melalui facade sehingga alur eksekusi menjadi lebih rapi dan mudah dipahami.

3.2.4 Composite Pattern (Product Categories)

A. Alasan Pemilihan

Katalog *e-commerce* memiliki struktur hirarki yang bertingkat, seperti Kategori Utama > Sub-Kategori > Produk. Composite pattern dipilih agar sistem dapat memperlakukan "Produk tunggal" dan "Kategori (kumpulan produk)" secara seragam melalui satu *interface* yang sama. Hal ini mempermudah logika penambahan item baru ke dalam katalog tanpa harus mengubah struktur kode utama.

B. Class Diagram

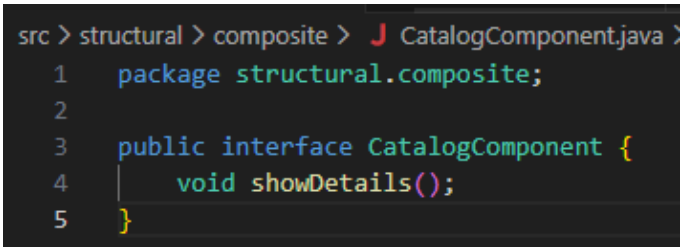


C. Penjelasan Implementasi

1. **Component** CatalogComponent: *Interface* standar untuk semua elemen katalog.
2. **Leaf** ProductItem: Objek akhir (produk) yang tidak memiliki anak.
3. **Composite** CategoryGroup: Objek yang menyimpan daftar CatalogComponent. Karena ia menyimpan tipe interface, maka kategori ini bisa berisi ProductItem maupun CategoryGroup lainnya (bersifat rekursif), memungkinkan pembuatan hirarki yang tidak terbatas.

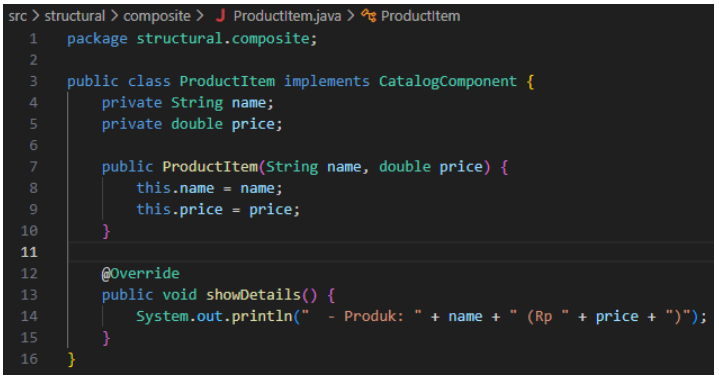
D. Penjelasan Kode Program :

1. CatalogComponent.java



Merupakan abstraksi yang menyatukan perilaku antara objek tunggal (produk) dan objek grup (kategori). Dengan mendefinisikan metode showDetails(), kelas ini memungkinkan klien untuk berinteraksi dengan seluruh struktur katalog secara seragam, tanpa perlu mengetahui apakah elemen tersebut adalah sebuah produk individual atau sebuah folder kategori besar.

2. ProductItem.java



Representasi dari elemen akhir atau terkecil dalam hirarki katalog, yaitu produk itu sendiri. Sebagai objek *Leaf*, kelas ini mengimplementasikan logika penampilan detail produk secara mandiri. Karena tidak memiliki "anak" atau dependensi ke objek lain di bawahnya, kelas ini menjadi ujung tombak informasi yang ditampilkan kepada pelanggan dalam sistem *e-commerce*.

3. CategoryGroup.java

```java
package structural.composite;

import java.util.ArrayList;
import java.util.List;

public class CategoryGroup implements CatalogComponent {
    private String name;
    private List<CatalogComponent> components = new ArrayList<>();

    public CategoryGroup(String name) {
        this.name = name;
    }

    public void addComponent(CatalogComponent component) {
        components.add(component);
    }

    @Override
    public void showDetails() {
        System.out.println("Kategori: " + name);
        for (CatalogComponent component : components) {
            component.showDetails();
        }
    }
}
```

Kelas ini memiliki peran krusial dalam membangun struktur pohon (*tree*) katalog. Dengan menyimpan koleksi dari CatalogComponent, kelas ini dapat menampung baik ProductItem maupun CategoryGroup lainnya secara rekursif. Mekanisme ini memberikan fleksibilitas tanpa batas bagi pengelola toko untuk membangun hirarki kategori yang sangat dalam (seperti Elektronik > Gadget > Smartphone) hanya dengan satu struktur kode yang elegan.

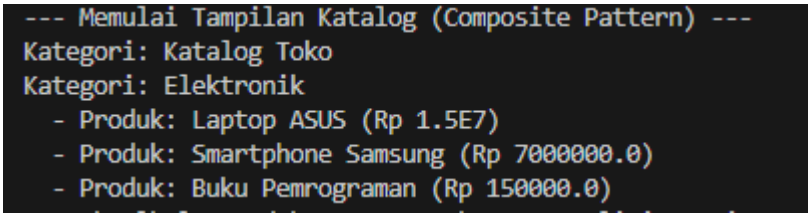
4. CompositeDemo.java

```java
package structural.composite;

public class CompositeDemo {
    public static void main(String[] args) {
        CategoryGroup mainCatalog = new CategoryGroup("Katalog Toko");

        CategoryGroup electronics = new CategoryGroup("Elektronik");
        electronics.addComponent(new ProductItem("Laptop ASUS", 15000000));
        electronics.addComponent(new ProductItem("Smartphone Samsung", 7000000));

        mainCatalog.addComponent(electronics);
        mainCatalog.addComponent(new ProductItem("Buku Pemrograman", 150000));

        System.out.println("--- Memulai Tampilan Katalog (Composite Pattern) ---");
        mainCatalog.showDetails();
    }
}
```

Kelas ini merupakan titik masuk untuk mensimulasikan struktur hirarki katalog yang kompleks. Implementasi di dalam kelas ini menunjukkan bagaimana sistem dapat membangun pohon kategori secara dinamis, mulai dari pembuatan kategori besar, sub-kategori, hingga item produk satuan—dan menyatukannya ke dalam satu

objek akar (*root*). Dengan memanggil method showDetails() pada objek kategori tertinggi, kelas ini memvalidasi kekuatan pola Composite dalam melakukan penelusuran data secara rekursif, di mana perintah tunggal dapat memicu penampilan detail seluruh cabang kategori di bawahnya secara otomatis.

Hasil Tes :



3.3 Kelompok Behavioral Patterns

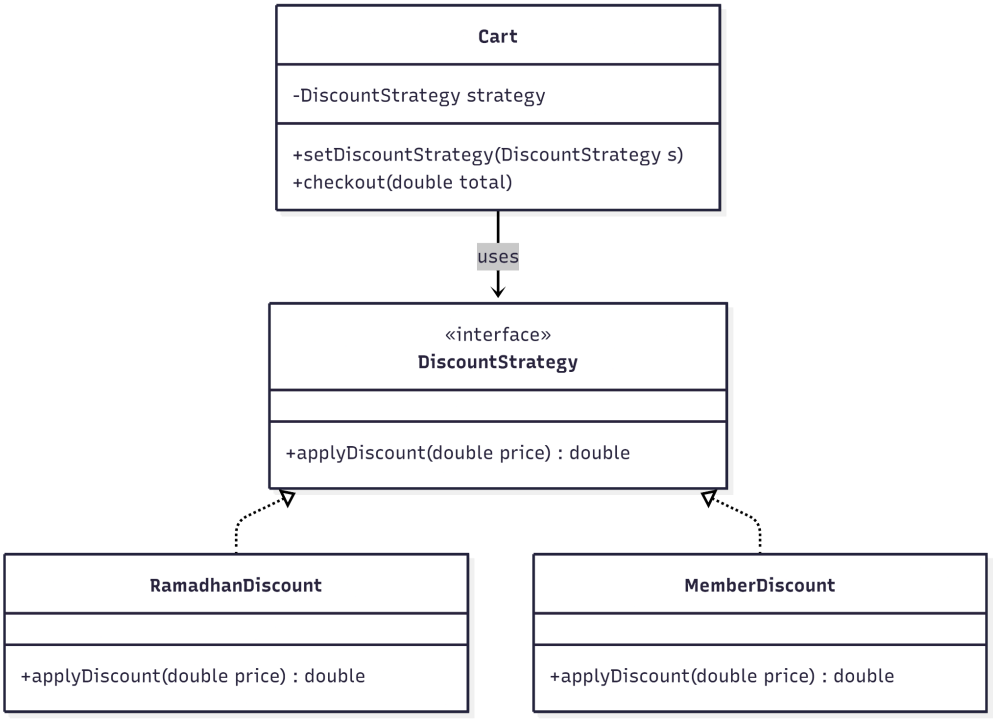
Kelompok *Behavioral Patterns* menitikberatkan pada komunikasi dan interaksi antar objek. Implementasi pattern di kategori ini bertujuan untuk mengelola algoritma dan tanggung jawab setiap entitas agar sistem lebih dinamis.

3.3.1 Strategy Pattern (Discount System)

A. Alasan Pemilihan

Logika perhitungan diskon pada marketplace sangat bervariasi. Dengan Strategy Pattern, setiap algoritma diskon dipisahkan ke dalam kelas mandiri sehingga penambahan jenis promo baru tidak akan mengganggu stabilitas kode utama pada keranjang belanja.

B. Class Diagram



C. Penjelasan Implementasi

Berikut adalah penjelasan implementasi strategy pattern:

1. Strategy Interface: Membuat interface DiscountStrategy sebagai kontrak dasar.
2. Concrete Strategy: Membuat implementasi seperti RamadhanDiscount dan MemberDiscount.
3. Context: Kelas Cart menggunakan objek strategi untuk menghitung total harga secara dinamis.

D. Penjelasan Kode Program

Berikut adalah penjelasan mengenai kode programnya :

1. DiscountStrategy.java

```
package behavioral.strategy;

public interface DiscountStrategy {
    double applyDiscount(double price);
}
```

File ini bertindak sebagai basis kontrak bagi seluruh variasi algoritma perhitungan harga. Dengan mendefinisikan metode applyDiscount(), interface ini memungkinkan sistem untuk memperlakukan berbagai jenis logika promosi secara seragam. Ini

memberikan pondasi bagi sistem yang *extensible* (mudah diperluas).

2. RamadhanDiscount.java

```java
package behavioral.strategy;

public class RamadhanDiscount implements DiscountStrategy {
    @Override
    public double applyDiscount(double price) {
        return price * 0.8; // Potongan 20%
    }
}
```

File ini mengapsulasi aturan bisnis spesifik untuk periode promosi musiman. Di dalamnya, metode applyDiscount() diimplementasikan dengan logika pemotongan harga sebesar 20% (price x 0.8). Pemisahan logika ini ke dalam file tersendiri sangat krusial; jika manajemen ingin mengubah besaran diskon Ramadhan di tahun depan menjadi 25%, pengembang hanya perlu mengubah satu baris kode di file ini tanpa berisiko merusak logika diskon lainnya atau mengganggu kode pada keranjang belanja (*Cart*).

3. MemberDiscount.java

```java
package behavioral.strategy;

public class MemberDiscount implements DiscountStrategy {
    @Override
    public double applyDiscount(double price) {
        return price * 0.9; // Potongan 10%
    }
}
```

Berbeda dengan diskon musiman, file ini menangani logika loyalitas pelanggan. Implementasi di dalam file ini berfokus pada pemberian apresiasi kepada pengguna terdaftar dengan potongan harga sebesar 10% (price x 0.9). Dengan memisahkan kelas ini dari RamadhanDiscount, sistem dapat membedakan tanggung jawab setiap aturan promo dengan jelas. Hal ini juga memungkinkan sistem untuk memvalidasi apakah seorang pengguna berhak mendapatkan strategi ini berdasarkan status keanggotaannya sebelum strategi ini disuntikkan ke dalam objek *Cart*.

4. Cart.java

```java
package behavioral.strategy;

public class Cart {
    private DiscountStrategy strategy;

    public void setDiscountStrategy(DiscountStrategy strategy) {
        this.strategy = strategy;
    }

    public void checkout(double total) {
        double finalPrice = strategy.applyDiscount(total);
        System.out.println("Total Awal: Rp" + total);
        System.out.println("Total Setelah Diskon: Rp" + finalPrice);
    }
}
```

Kelas ini mengelola referensi ke objek strategi secara dinamis. Melalui metode setDiscountStrategy(), kelas Cart dapat mengubah perilakunya saat aplikasi sedang berjalan (*runtime switching*). Ini sangat penting dalam e-commerce di mana jenis diskon bisa berubah tergantung pada status login pengguna atau kalender promo tanpa harus melakukan *restart* pada aplikasi.

5. Main.java

```java
package behavioral.strategy;

public class Main {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Cart keranjang = new Cart();

        System.out.println(x: "--- Skenario Promo Ramadhan ---");
        keranjang.setDiscountStrategy(new RamadhanDiscount());
        keranjang.checkout(total: 100000);

        System.out.println(x: "\n--- Skenario Diskon Member ---");
        keranjang.setDiscountStrategy(new MemberDiscount());
        keranjang.checkout(total: 100000);
    }
}
```

File ini mendemonstrasikan fungsionalitas dinamis dari pola Strategy. Penjelasan kodenya harus memaparkan bagaimana satu objek keranjang belanja yang sama dapat menghasilkan output harga yang berbeda hanya dengan menyuntikkan (*injecting*) strategi yang berbeda. Ini membuktikan bahwa sistem kita sangat adaptif terhadap perubahan kebijakan marketing.

Hasil tes:

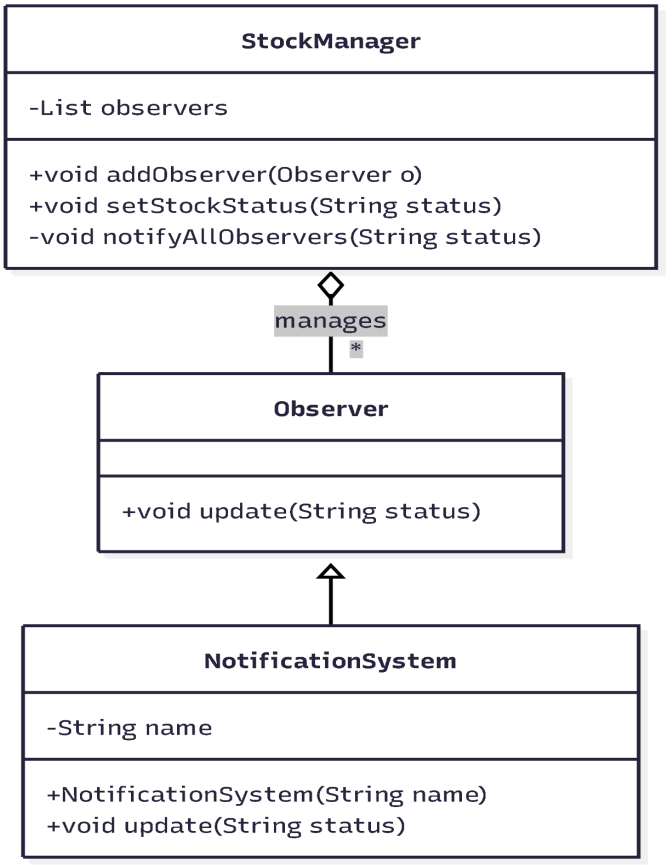
```
--- Skenario Promo Ramadhan ---
Total Awal: Rp100000.0
Total Setelah Diskon: Rp80000.0

--- Skenario Diskon Member ---
Total Awal: Rp100000.0
Total Setelah Diskon: Rp90000.0
```

3.3.2 Observer Pattern (Notification System)

A. Alasan Pemilihan

Penggunaan Observer Pattern bertujuan untuk menciptakan sistem komunikasi yang otomatis dan tidak saling bergantung (*loosely coupled*) antara manajemen stok dan sistem notifikasi. Dalam alur bisnis, banyak pihak (seperti Admin Gudang dan Pelanggan) perlu mengetahui perubahan status barang secara instan, namun sistem stok tidak perlu tahu secara mendetail siapa saja yang dikirimi notifikasi tersebut. Dengan pola ini, kita cukup mendaftarkan pihak-pihak yang berkepentingan sebagai *observer*, sehingga ketika terjadi perubahan stok atau pengiriman, semua pihak akan menerima pembaruan secara otomatis tanpa perlu mengubah logika internal kelas manajemen stok.

B. Class Diagram



C. Penjelasan Implementasi

Berikut adalah penjelasan implementasi Observer Pattern:

1. Mendefinisikan Observer: Membuat interface Observer dengan metode update() yang akan dipanggil saat terjadi perubahan.
2. Membuat Subject: Kelas StockManager mengelola daftar observer dan bertanggung jawab mengirim sinyal melalui notifyAllObservers().
3. Implementasi Concrete Observer: Kelas NotificationSystem (seperti Admin Notif dan User Notif) bereaksi terhadap perubahan status dengan menampilkan pesan yang relevan.
4. Otomasi: Begitu metode setStockStatus() dipanggil, seluruh pihak yang terdaftar akan menerima pembaruan secara instan.

D. Penjelasan Kode Program

Berikut adalah penjelasan mengenai kode programnya :

1. Observer.java

```java
package behavioral.observer;

public interface Observer {
    void update(String status);
}
```

File ini berfungsi sebagai kontrak utama atau *interface* yang mendefinisikan standar bagi semua kelas yang ingin menjadi pengamat dalam sistem. Di dalamnya terdapat metode update(String status) yang bertindak sebagai gerbang masuk bagi informasi baru, sehingga setiap kelas yang mengimplementasikan *interface* ini dijamin memiliki cara yang seragam untuk menerima dan merespons pembaruan status dari pusat informasi tanpa peduli siapa pengirimnya.

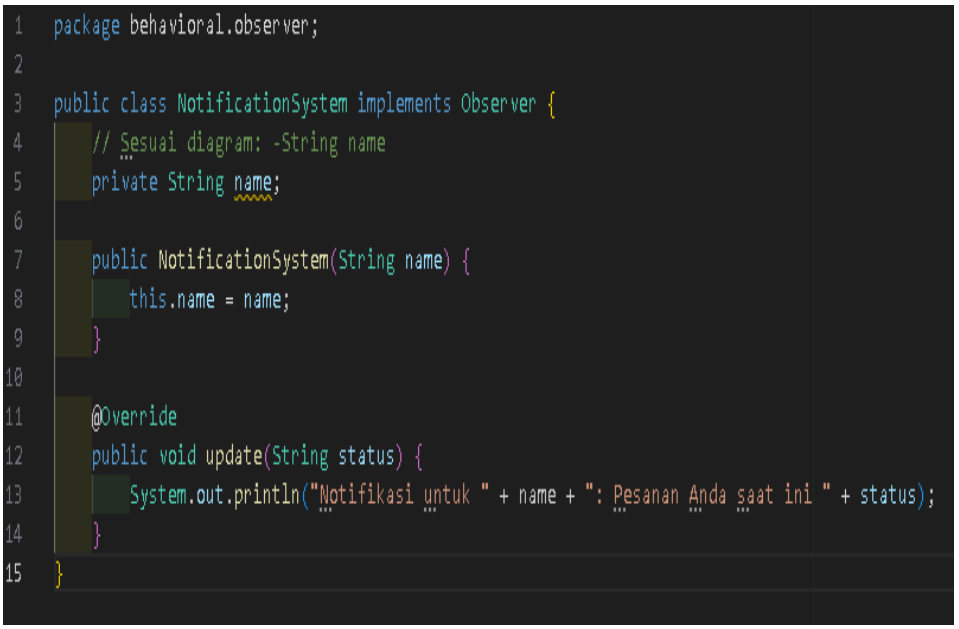
2. StockManager.java

```java
package behavioral.observer;
import java.util.ArrayList;
import java.util.List;

public class StockManager {
    // Sesuai diagram: -List observers
    private List<Observer> observers = new ArrayList<>();

    public void addObserver(Observer o) {
        observers.add(o);
    }

    public void setStockStatus(String status) {
        System.out.println("\n[StockManager] Perubahan status: " + status);
        notifyAllObservers(status);
    }

    // Sesuai diagram: -void notifyAllObservers
    private void notifyAllObservers(String status) {
        for (Observer o : observers) {
            o.update(status);
        }
    }
}
```

File ini berperan sebagai pusat kendali atau *Subject* yang memiliki tanggung jawab besar dalam mengelola daftar seluruh pengamat (*observers*) melalui koleksi ArrayList. Logika utama dalam file ini adalah kemampuannya untuk melakukan *broadcast* atau siaran informasi; setiap kali terjadi perubahan data pada stok, file ini akan

menjalankan iterasi otomatis untuk memanggil metode update pada setiap objek yang terdaftar, sehingga memastikan sinkronisasi data terjadi secara *real-time* di seluruh sistem.

3. NotificationSystem.java



```java
package behavioral.observer;

public class NotificationSystem implements Observer {
    // Sesuai diagram: -String name
    private String name;

    public NotificationSystem(String name) {
        this.name = name;
    }

    @Override
    public void update(String status) {
        System.out.println("Notifikasi untuk " + name + ": Pesanan Anda saat ini " + status);
    }
}
```

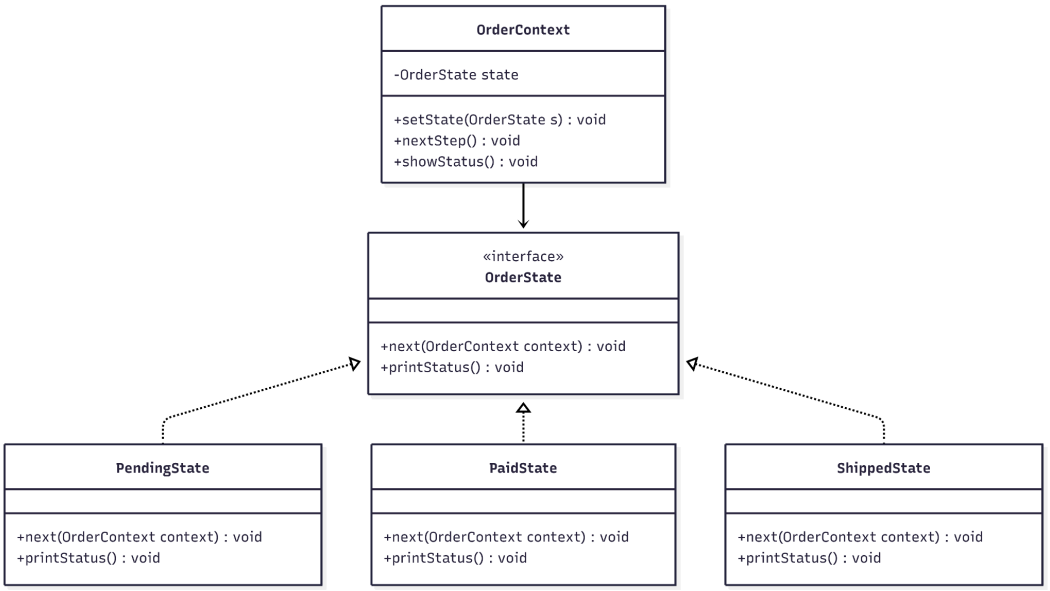
File ini adalah implementasi nyata dari pengamat yang bertugas menerima pesan dan menampilkannya kepada pengguna. Di dalam kodingan ini, terdapat atribut name yang berfungsi untuk membedakan identitas penerima, seperti Admin Gudang atau Pembeli, sehingga saat metode update dipicu oleh StockManager, file ini dapat mencetak pesan notifikasi yang spesifik dan personal sesuai dengan peran masing-masing pengguna di dalam sistem e-commerce.

3.3.3 State Pattern (Order Life Cycle)

A. Alasan Pemilihan

Pesanan dalam *e-commerce* memiliki siklus hidup yang kompleks (Menunggu Pembayaran -> Dibayar -> Dikirim -> Selesai). State pattern dipilih untuk menghindari penggunaan blok if-else atau switch-case yang terlalu panjang dan sulit diatur. Dengan *pattern* ini, logika perilaku setiap status dipisahkan ke dalam kelas tersendiri, sehingga sistem lebih fleksibel jika di masa depan ada penambahan status baru (misal: "Dibatalkan").

B. Class Diagram

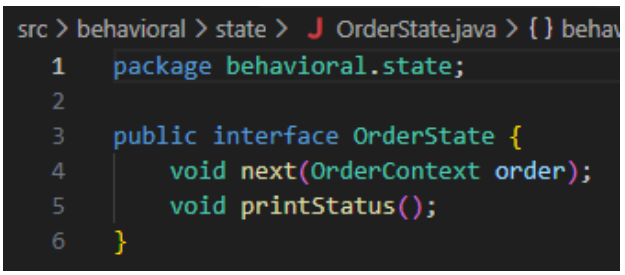


C. Penjelasan Implementasi

1. **Interface OrderState**: Menentukan metode transisi status.
2. **Concrete States (PendingState, PaidState, ShippedState)**: Masing-masing kelas menentukan aksi apa yang terjadi saat ini dan status apa yang akan dituju selanjutnya.
3. **Context OrderContext**: Objek utama yang berinteraksi dengan pengguna. Ia menyimpan status saat ini dan mendelegasikan perintah nextStep() ke objek State yang sedang aktif.

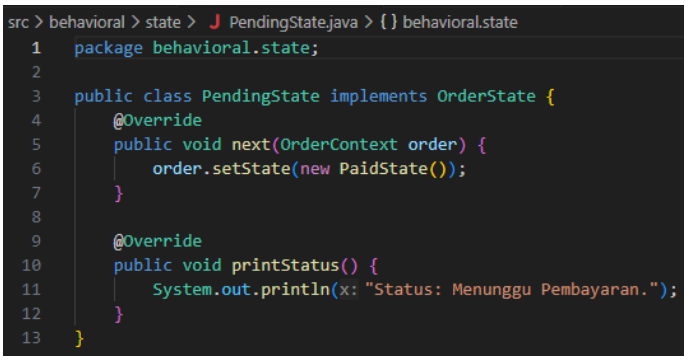
D. Penjelasan Kode Program :

1. OrderState



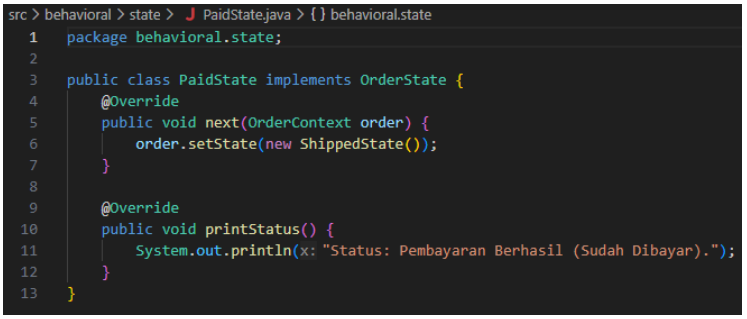
Interface ini mendefinisikan perilaku standar untuk setiap status yang mungkin dimiliki oleh sebuah pesanan. Dengan mengisolasi logika transisi ke dalam interface ini, sistem dapat menjamin bahwa setiap perubahan status (seperti dari *Pending* ke *Paid*) mengikuti aturan bisnis yang ketat dan terpusat, mencegah terjadinya perubahan status yang tidak logis secara sistem.

2. PendingState



Kelas ini merepresentasikan fase awal dari siklus transaksi, yaitu saat pesanan telah dibuat namun belum dilakukan pembayaran. Fokus utama dari kelas ini adalah mengunci logika bisnis pada tahap pra-pembayaran, di mana satu-satunya transisi legal yang diizinkan adalah menuju status pembayaran berhasil. Dengan mengisolasi logika ini, sistem menjamin bahwa pesanan tidak dapat dikirim sebelum melewati verifikasi di kelas ini.

3. PaidState



Kelas ini merepresentasikan kondisi di mana transaksi telah divalidasi dan dana telah diterima oleh sistem. Pada tahap ini, perilaku objek berubah untuk fokus pada persiapan logistik. Enkapsulasi dalam kelas ini memastikan bahwa hanya pesanan yang telah berstatus "*Paid*" yang dapat memicu fungsi pengemasan dan pengiriman, memberikan lapisan keamanan tambahan yang memisahkan antara logika finansial dan logika logistik.

4. ShippedState

Kelas ini menangani perilaku objek pada fase akhir, yaitu saat barang sedang dalam perjalanan menuju pelanggan. Sebagai State akhir dalam simulasi ini, ia bertindak sebagai titik terminasi alur atau transisi ke status "Diterima". Implementasi kelas ini menunjukkan bagaimana sistem dapat menonaktifkan fitur-fitur tertentu (seperti pembatalan pesanan) yang sudah tidak diizinkan lagi ketika barang sudah keluar dari gudang.

5. OrderContext

```
src > behavioral > state > J OrderContext.java > { } behavioral.state
1   package behavioral.state;
2
3   public class OrderContext {
4       private OrderState state = new PendingState();
5
6       public void setState(OrderState state) {
7           this.state = state;
8       }
9
10      public void nextStep() {
11          state.next(this);
12      }
13
14      public void showStatus() {
15          state.printStatus();
16      }
17  }
```

Bertindak sebagai penghubung utama bagi pengguna dan penyimpan status pesanan saat ini. Kelas ini mendelegasikan semua permintaan perubahan status kepada objek State yang sedang aktif. Dengan menjaga referensi ke status saat ini, OrderContext memastikan bahwa perilaku sistem selalu konsisten dengan kondisi riil pesanan, memberikan transparansi data yang akurat bagi pembeli maupun admin toko.

6. StateDemo

```
src > behavioral > state > J StateDemo.java > StateDemo > main(String[])
1   package behavioral.state;
2
3   public class StateDemo {
        Run | Debug
4       public static void main(String[] args) {
5           OrderContext myOrder = new OrderContext();
6
7           System.out.println(x: "--- State Pattern Demo ---");
8           myOrder.showStatus();
9           myOrder.nextStep();
10          myOrder.showStatus();
11          myOrder.nextStep();
12          myOrder.showStatus();
13      }
14  }
```

Kelas ini bertindak sebagai simulator siklus hidup transaksi yang menguji logika perpindahan status pada sebuah pesanan. Inti dari kelas ini adalah pengujian terhadap OrderContext untuk memastikan bahwa transisi perilaku antar status (seperti *Pending* ke

Paid) berjalan sesuai dengan aturan bisnis yang telah dienkapsulasi. Melalui kelas demo ini, sistem membuktikan kemampuannya untuk mengubah respon fungsional secara dinamis saat objek berpindah dari satu kondisi ke kondisi lain, sekaligus menjamin bahwa kode klien tetap bersih dari logika percabangan if-else yang rumit.

Hasil Tes :

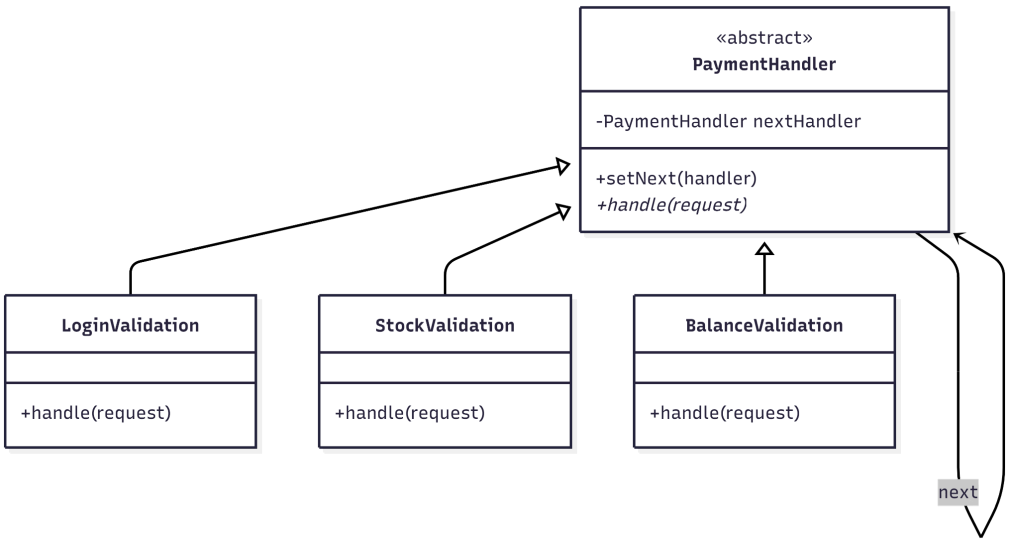
```
--- State Pattern Demo ---
Status: Menunggu Pembayaran.
Status: Pembayaran Berhasil (Sudah Dibayar).
Status: Barang dalam Pengiriman.
```

3.3.4 Chain of Responsibility Pattern (Checkout Validation)

A. Alasan Pemilihan

Sebuah transaksi pembayaran harus melewati berbagai tahap validasi seperti status login, ketersediaan stok, dan kecukupan saldo. Menggabungkan semua logika ini dalam satu fungsi besar akan membuat kode sulit dibaca dan kaku. Chain of Responsibility dipilih agar setiap tahap validasi menjadi satu kelas mandiri yang terpisah. Ini memungkinkan kita untuk mengubah urutan validasi atau menambah aturan baru (misalnya cek promo) dengan hanya menyambungkan rantai (*chain*) tanpa merusak logika validasi yang sudah ada.

B. Class Diagram



C. Penjelasan Implementasi

Berikut adalah penjelasan mengenai implementasi Chain of Responsibility pattern:

1. Membuat Base Handler: Membuat kelas abstrak PaymentHandler yang memiliki atribut nextHandler untuk menyimpan referensi ke pemroses berikutnya.
2. Metode Perantai (Chaining): Menyediakan method setNext() yang memungkinkan kita untuk menyusun urutan validasi secara dinamis.
3. Implementasi Concrete Handler: Membuat kelas validasi spesifik seperti LoginValidation, StockValidation, dan BalanceValidation yang masing-masing memiliki logika cek tunggal.
4. Mekanisme Delegasi: Setiap handler mengecek permintaan; jika valid, ia akan meneruskan ke passToNext(), namun jika gagal, ia akan menghentikan rantai dan mengembalikan error.

D. Penjelasan Kode Program :

1. BalanceValidation

```java
package behavioral.chain_of_responsibility;

public class BalanceValidation extends PaymentHandler {
    @Override
    public void handle(String request) {
        System.out.println("Memeriksa saldo...");
        if (request.contains("enough_balance")) {
            System.out.println("✓ Saldo mencukupi.");
            passToNext(request);
        } else {
            System.out.println("✗ Error: Saldo tidak cukup!");
        }
    }
}
```

File ini merupakan handler konkret dalam pola *Chain of Responsibility* yang bertugas melakukan validasi saldo pada proses pembayaran. Kode di dalamnya bekerja dengan cara memeriksa apakah request mengandung informasi “enough_balance”. Jika saldo mencukupi, sistem akan menampilkan pesan bahwa saldo valid lalu meneruskan proses ke handler berikutnya melalui method passToNext(request). Namun jika saldo tidak cukup, proses akan dihentikan dan ditampilkan pesan error, sehingga setiap tahap validasi dapat

berjalan berantai dan terpisah sesuai tanggung jawabnya masing-masing.

2. LoginValidation

```java
package behavioral.chain_of_responsibility;

public class LoginValidation extends PaymentHandler {
    @Override
    public void handle(String request) {
        System.out.println(x: "Memeriksa status Login...");
        if (request.contains(s: "logged_in")) {
            System.out.println(x: "✓ User sudah login.");
            passToNext(request);
        } else {
            System.out.println(x: "✗ Error: User belum login!");
        }
    }
}
```

File ini merupakan handler konkret dalam pola *Chain of Responsibility* yang bertugas melakukan validasi status login pengguna sebelum proses pembayaran dilanjutkan. Kode di dalamnya bekerja dengan cara memeriksa apakah request mengandung status "logged_in". Jika pengguna sudah login, sistem akan menampilkan pesan bahwa user valid lalu meneruskan proses ke handler berikutnya melalui method passToNext(request). Namun jika pengguna belum login, proses akan dihentikan dan ditampilkan pesan error, sehingga mekanisme validasi dapat berjalan bertahap sesuai tanggung jawab masing-masing handler dalam rantai proses.

3. PaymentHandler

```java
package behavioral.chain_of_responsibility;

public abstract class PaymentHandler {
    protected PaymentHandler nextHandler;

    public void setNext(PaymentHandler handler) {
        this.nextHandler = handler;
    }

    public abstract void handle(String request);

    protected void passToNext(String request) {
        if (nextHandler != null) {
            nextHandler.handle(request);
        } else {
            System.out.println(x: "Semua validasi selesai. Request diterima!");
        }
    }
}
```

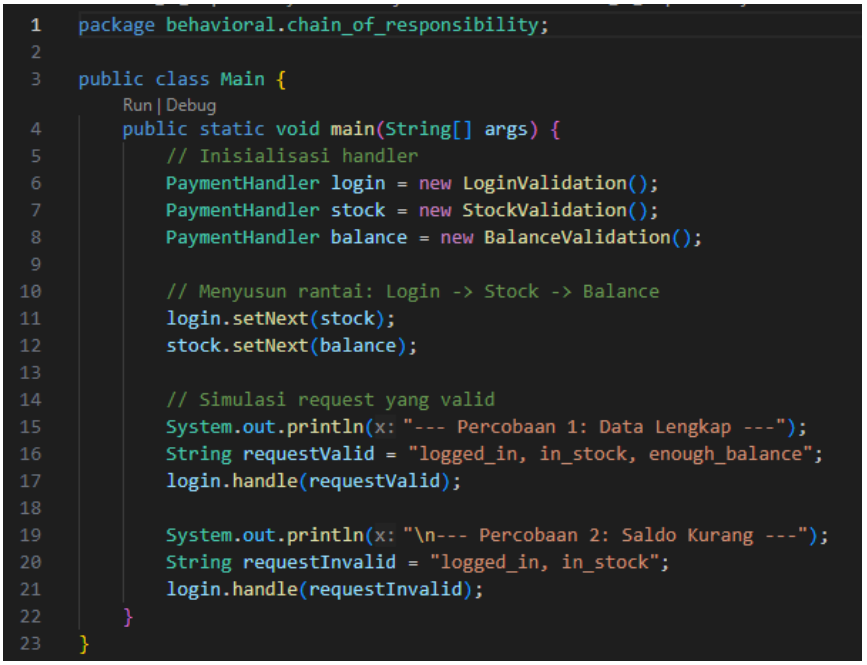
File ini merupakan kelas abstrak handler dalam pola *Chain of Responsibility* yang berperan sebagai dasar bagi semua proses validasi pembayaran. Kode di dalamnya menyediakan atribut nextHandler untuk menyimpan handler berikutnya dalam rantai, serta method setNext() untuk menghubungkan satu handler dengan handler lain. Method abstrak handle() digunakan sebagai kontrak yang wajib diimplementasikan oleh setiap handler turunan sesuai tanggung jawabnya masing-masing. Selain itu, terdapat method passToNext() yang berfungsi meneruskan request ke handler selanjutnya jika masih ada proses validasi, atau menampilkan pesan bahwa semua validasi selesai jika rantai telah berakhir. Struktur ini memungkinkan alur pengecekan berjalan fleksibel, modular, dan mudah dikembangkan.

4. StockValidation

```java
package behavioral.chain_of_responsibility;

public class StockValidation extends PaymentHandler {
    @Override
    public void handle(String request) {
        System.out.println(x: "Memeriksa stok barang...");
        if (request.contains(s: "in_stock")) {
            System.out.println(x: "✓ Stok tersedia.");
            passToNext(request);
        } else {
            System.out.println(x: "✗ Error: Stok habis!");
        }
    }
}
```

File ini merupakan handler konkret dalam pola *Chain of Responsibility* yang bertugas melakukan validasi ketersediaan stok barang pada alur proses pembayaran. Kode di dalamnya bekerja dengan cara memeriksa apakah request mengandung status “in_stock”. Jika stok tersedia, sistem akan menampilkan pesan bahwa stok valid lalu meneruskan proses ke handler berikutnya melalui method passToNext(request). Namun jika stok habis, proses akan dihentikan dan ditampilkan pesan error, sehingga setiap tahapan validasi dapat berjalan berurutan sesuai tanggung jawab masing-masing handler dalam rantai proses.

5. Main.java



```java
package behavioral.chain_of_responsibility;

public class Main {
    Run | Debug
    public static void main(String[] args) {
        // Inisialisasi handler
        PaymentHandler login = new LoginValidation();
        PaymentHandler stock = new StockValidation();
        PaymentHandler balance = new BalanceValidation();

        // Menyusun rantai: Login -> Stock -> Balance
        login.setNext(stock);
        stock.setNext(balance);

        // Simulasi request yang valid
        System.out.println(x: "--- Percobaan 1: Data Lengkap ---");
        String requestValid = "logged_in, in_stock, enough_balance";
        login.handle(requestValid);

        System.out.println(x: "\n--- Percobaan 2: Saldo Kurang ---");
        String requestInvalid = "logged_in, in_stock";
        login.handle(requestInvalid);
    }
}
```

File ini merupakan kelas utama (client) dalam implementasi pola *Chain of Responsibility* yang berfungsi untuk menginisialisasi handler, menyusun urutan rantai proses, serta menjalankan simulasi request. Kode di dalamnya bekerja dengan membuat beberapa handler seperti LoginValidation, StockValidation, dan BalanceValidation, lalu menghubungkannya secara berurutan menggunakan method setNext() sehingga membentuk alur validasi berantai (Login → Stok → Saldo). Setelah rantai terbentuk, program mengirimkan request untuk diuji melalui handler pertama. Jika setiap validasi terpenuhi maka request akan diteruskan hingga akhir, sedangkan jika salah satu validasi gagal maka proses akan berhenti pada handler tersebut. Struktur ini menunjukkan bagaimana client mengontrol alur tanpa perlu mengetahui detail implementasi setiap handler.

LINK GITHUB :

https://github.com/RadityaNalendraU/Tugas_Analisis_Design_Pattern_Domain_Kasus_E-Commerce_Kelompok_Perpusdimari_CCDP1