ANALISIS SOFTWARE DESIGN PATTERN PADA E-COMMERCE

TUGAS KELOMPOK “THE SEMICOLON” CLEAN CODE DAN DESIGN PATTERN

HABEAHAN LANDO RAY MAHANUGRA	**10119327**
KARYADI SIMAMORA	**10119322**
DAFFA SACOFI MUTAWAKKIL	**10118401**

PROGRAM STUDI TEKNIK INFORMATIKA

FAKULTAS TEKNIK DAN ILMU KOMPUTER

UNIVERSITAS KOMPUTER INDONESIA

2023

SOFTWARE DESIGN PATTERN PADA E-COMMERCE

Situs E-Commerce adalah situs yang paling produktif dibanding dengan situs-situs yang lain. Waktu merupakan elemen penting, semakin cepat sebuah situs online dan beroperasi semakin cepat keuntungan pemilik usaha. Untuk itu pengembangan situs e-Commerce diharapkan singkat. Namun waktu pengembangan yang singkat menghasilkan situs yang kurang optimal. Diperlukan sebuah metode yang cepat namun mampu menghasilkan rancangan e-commerce yang baik.

Bentuk situs e-commerce bervariasi. Situs yang diperuntukkan untuk menjual barang langsung pada pelanggan disebut e-commerce B2C (Business to Customer). Situs e-commerce yang diperuntukkan untuk menjual barang atau jasa untuk sesama entitas bisnis disebut sebagai B2B (Business to Business). Software Design Pattern pada Situs E-Commerce Dengan melihat banyaknya situs-situs e-commerce yang ada, maka ditemukan beberapa persamaan-persamaan di antara situs-situs e-commerce tersebut. Kesamaan-kesamaan tersebut bisa digunakan untuk mempersingkat proses pengembangan sistem yaitu pada tahap identifikasi kebutuhan informasi atau requirement specification.

Design pattern yang umum digunakan oleh e-marketplace juga terdapat dalam situs-situs web yang lain. Namun beberapa design pattern yang spesifik teridentifikasi dalam web portal e-marketplace adalah sebagai berikut. dalam kasus ini design pattern yang akan di analisis adalah pada bagian bagian berikut:

1. metode pembayaran
2. status pembayaran
3. pencarian produk
4. kerjang belanja
5. promosi dan diskon
6. rekomendasi produk
7. notifikasi
8.riwayat pembelian
9.pengelolaan stok

Behavioral Design Patterns (karyadi simamora 10119322)

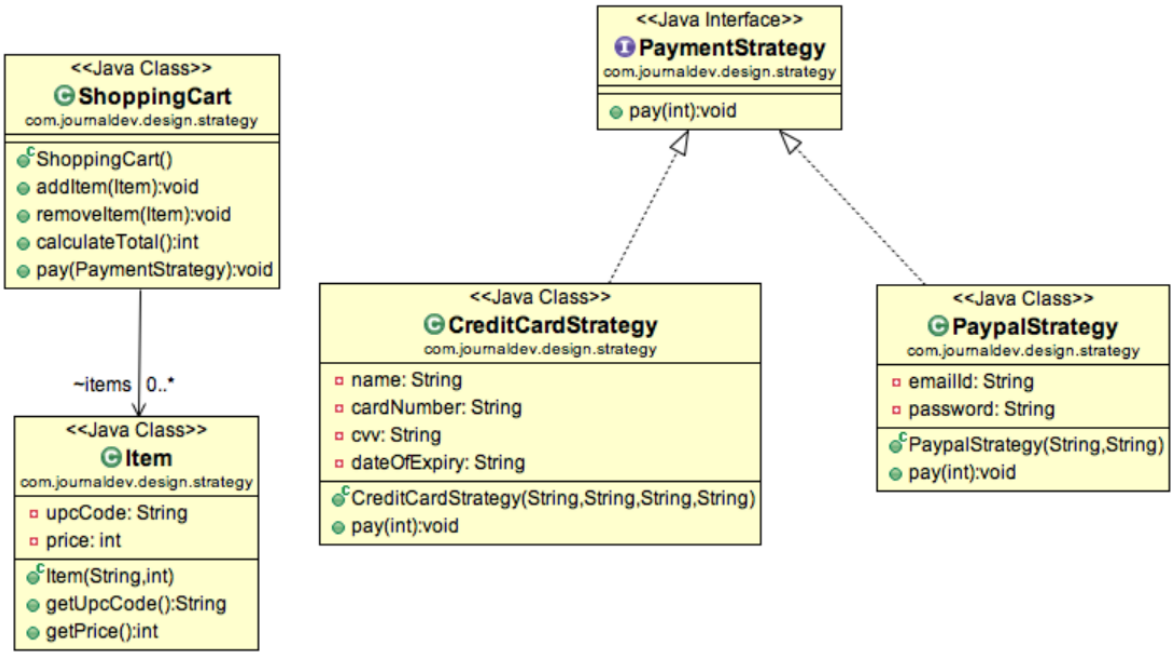
1. **Fitur: Metode Pembayaran**

Design Pattern: Strategy Pattern

Deskripsi: Strategy Pattern dapat digunakan untuk mengimplementasikan berbagai metode pembayaran yang berbeda, seperti kartu kredit, transfer bank, atau dompet digital. Dengan

Strategy Pattern, kita dapat dengan mudah menambahkan metode pembayaran baru tanpa harus mengubah kode yang ada.

code dan design pattern nya



PaymentStrategy.java

```
package com.pattern.design.strategy;

public interface PaymentStrategy {

    public void pay(int amount);
}
```

Interface PaymentStrategy berfungsi sebagai dasar untuk semua strategi pembayaran yang akan diimplementasikan nanti. Method pay akan digunakan untuk melakukan pembayaran dengan jumlah tertentu.

CreditCardStrategy.java

```
package com.pattern.design.strategy;

public class CreditCardStrategy implements PaymentStrategy {
```

```
        private String name;
        private String cardNumber;
        private String cvv;
        private String dateOfExpiry;

          public CreditCardStrategy(String nm, String ccNum, String
cvv, String expiryDate){
            this.name=nm;
            this.cardNumber=ccNum;
            this.cvv=cvv;
            this.dateOfExpiry=expiryDate;
        }
        @Override
        public void pay(int amount) {
                System.out.println(amount +" paid with credit/debit
card");
        }

    }
```

Kelas CreditCardStrategy adalah salah satu dari strategi pembayaran yang mengimplementasikan interface PaymentStrategy. Kelas ini digunakan untuk melakukan pembayaran dengan menggunakan kartu kredit. Ketika metode pay dipanggil, kelas ini akan mencetak pesan pembayaran dengan kartu kredit dengan jumlah tertentu.

PayPalStrategy.java

```
    package com.pattern.design.strategy;

    public class PaypalStrategy implements PaymentStrategy {

        private String emailId;
        private String password;

        public PaypalStrategy(String email, String pwd){
            this.emailId=email;
            this.password=pwd;
        }
```

```
    @Override
    public void pay(int amount) {
        System.out.println(amount + " paid using Paypal.");
    }
}
```

Kelas PayPalStrategy adalah salah satu dari strategi pembayaran yang juga mengimplementasikan interface PaymentStrategy. Kelas ini digunakan untuk melakukan pembayaran dengan menggunakan PayPal. Ketika metode pay dipanggil, kelas ini akan mencetak pesan pembayaran dengan PayPal dengan jumlah tertentu.

ShoppingCart.java

```
package com.pattern.design.strategy;

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.List;

public class ShoppingCart {

    //List of items
    List<Item> items;

    public ShoppingCart(){
        this.items=new ArrayList<Item>();
    }

    public void addItem(Item item){
        this.items.add(item);
    }

    public void removeItem(Item item){
        this.items.remove(item);
    }

    public int calculateTotal(){
        int sum = 0;
        for(Item item : items){
```

```
            sum += item.getPrice();
        }
        return sum;
    }

    public void pay(PaymentStrategy paymentMethod){
        int amount = calculateTotal();
        paymentMethod.pay(amount);
    }
}
```

Kelas ShoppingCart berfungsi untuk merepresentasikan keranjang belanja. Di sini, kita dapat menambahkan atau menghapus item dari keranjang belanja. Metode calculateTotal digunakan untuk menghitung total harga semua item dalam keranjang belanja. Metode pay menerima objek PaymentStrategy sebagai argumen, dan ketika metode ini dipanggil, kelas ini akan melakukan pembayaran dengan strategi pembayaran yang dipilih.

item.java

```
package com.pattern.design.strategy;

public class Item {

    private String upcCode;
    private int price;

    public Item(String upc, int cost){
        this.upcCode=upc;
        this.price=cost;
    }

    public String getUpcCode() {
        return upcCode;
    }

    public int getPrice() {
        return price;
    }
```

```
    }
```

Kelas Item digunakan untuk merepresentasikan item dalam shopping cart. Setiap item memiliki atribut upCode (kode unik) dan harga (price). Di sini, kita hanya menyediakan getter untuk mendapatkan upCode dan harga dari item.

shoppingcardTest.java

```
package com.pattern.design.strategy;

public class ShoppingCartTest {

    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        Item item1 = new Item("1234",10);
        Item item2 = new Item("5678",40);

        cart.addItem(item1);
        cart.addItem(item2);

        //pay by paypal
        cart.pay(new PaypalStrategy("myemail@example.com", "mypwd"));

        //pay by credit card
                    cart.pay(new   CreditCardStrategy("Pankaj   Kumar",
"1234567890123456", "786", "12/15"));
    }

}
```

Contoh dari implementasi strategi pembayaran menggunakan Strategy Pattern. Dalam contoh ini, sebuah objek ShoppingCart digunakan untuk melakukan pembelian beberapa item, dan kemudian

metode pay dari objek ShoppingCart dipanggil dengan berbagai strategi pembayaran yang berbeda.

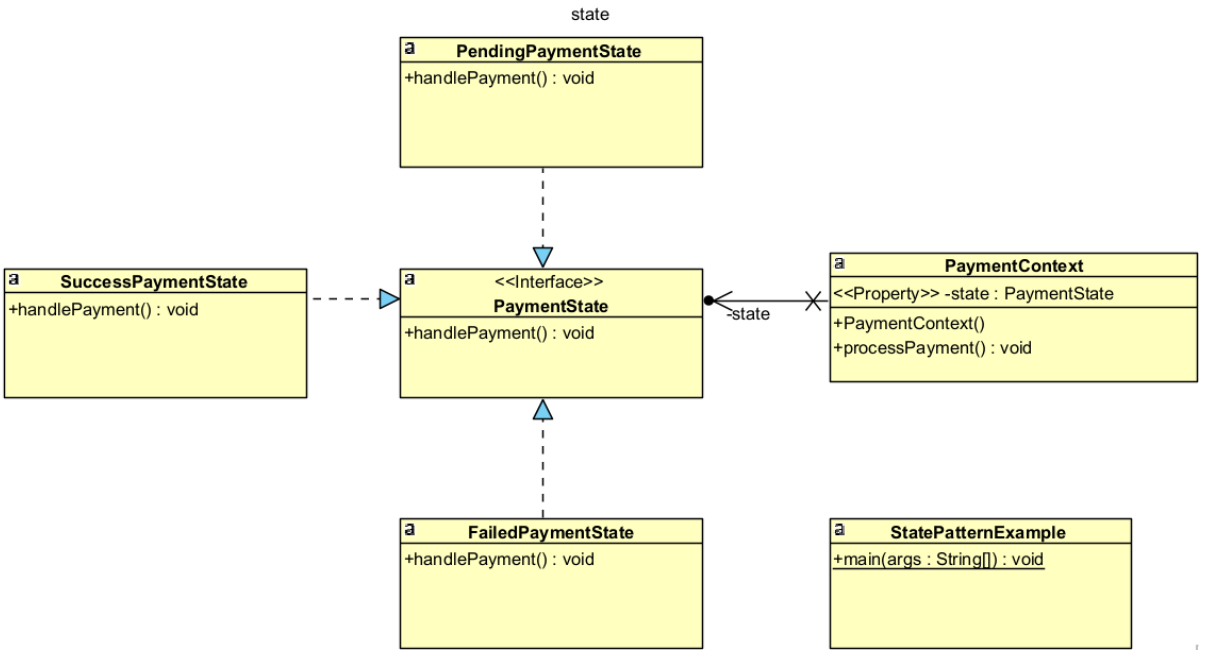
```
50 paid using Paypal.
50 paid with credit/debit card
```

2. **Fitur: Pemberitahuan status pembayaran**

Design Pattern : State Pattern

Pola ini termasuk dalam kategori Behavioral Design Pattern. State Pattern digunakan untuk mengelola status dari suatu objek dan mengubah perilakunya sesuai dengan statusnya. Dalam kasus pembayaran, State Pattern dapat digunakan untuk mengatur status pembayaran seperti "menunggu pembayaran", "pembayaran gagal", "pembayaran berhasil", dan lain sebagainya.

code dan design patternya



gambar 2 state pattern

paymentstateInterface.java

```
package com.pattern.design.state;
public interface PaymentState {
    void handlePayment();
}
```

Interface ini mendefinisikan kontrak untuk berbagai status pembayaran yang akan diimplementasikan oleh kelas-kelas lain.

PaymentState memiliki satu method handlePayment() yang akan diimplementasikan oleh kelas-kelas konkrit untuk menangani aksi yang sesuai dengan status pembayaran tertentu.

PaymentState.java

```
package com.pattern.design.state;
public class PendingPaymentState implements PaymentState {
    @Override
    public void handlePayment() {
```

```
                    System.out.println("Pembayaran sedang menunggu
konfirmasi.");
        }
    }

    public class SuccessPaymentState implements PaymentState {
        @Override
        public void handlePayment() {
            System.out.println("Pembayaran berhasil.");
        }
    }

    public class FailedPaymentState implements PaymentState {
        @Override
        public void handlePayment() {
            System.out.println("Pembayaran gagal.");
        }
    }
```

PendingPaymentState:

- Kelas ini adalah salah satu implementasi dari PaymentState.
- Ketika objek PaymentContext berada dalam status PendingPaymentState, jika method processPayment() dipanggil, maka akan mencetak pesan "Pembayaran sedang menunggu konfirmasi."

SuccessPaymentState:

- Kelas ini adalah implementasi lain dari PaymentState.
- Ketika objek PaymentContext berada dalam status SuccessPaymentState, jika method processPayment() dipanggil, maka akan mencetak pesan "Pembayaran berhasil."

FailedPaymentState:

- Kelas ini adalah implementasi lain dari PaymentState.
- Ketika objek PaymentContext berada dalam status FailedPaymentState, jika method processPayment() dipanggil, maka akan mencetak pesan "Pembayaran gagal."

PaymentContext.java

```
package com.pattern.design.state;

public class PaymentContext {

    private PaymentState state;

    public PaymentContext() {

        state = new PendingPaymentState();

    }

    public void setState(PaymentState state) {

        this.state = state;

    }

    public void processPayment() {

        state.handlePayment();}}
```

- Kelas ini berperan sebagai konteks atau kelas utama yang menggunakan State Pattern.
- Kelas ini memiliki atribut state yang bertipe PaymentState untuk menyimpan status pembayaran saat ini.

- Method setState(PaymentState state) digunakan untuk mengubah status pembayaran dengan state yang baru.

paymentStateTest.java

```
package com.pattern.design.state;

public class StatePatternExample {

    public static void main(String[] args) {

        PaymentContext paymentContext = new PaymentContext();

        // Pembayaran sedang menunggu konfirmasi

        paymentContext.processPayment();

        // Ubah status pembayaran menjadi berhasil

        paymentContext.setState(new SuccessPaymentState());

        paymentContext.processPayment();

        // Ubah status pembayaran menjadi gagal

        paymentContext.setState(new FailedPaymentState());

        paymentContext.processPayment();

    }

}
```

- Kelas ini berfungsi sebagai kelas utama yang berisi method main().
- Di dalam method main(), kita membuat objek PaymentContext, lalu mengubah status pembayaran menjadi SuccessPaymentState dan FailedPaymentState, serta memanggil processPayment() untuk melihat hasil keluaran sesuai dengan status pembayaran yang berbeda.

  keluaran:

```
// Pembayaran sedang menunggu konfirmasi.

// Pembayaran berhasil.
```

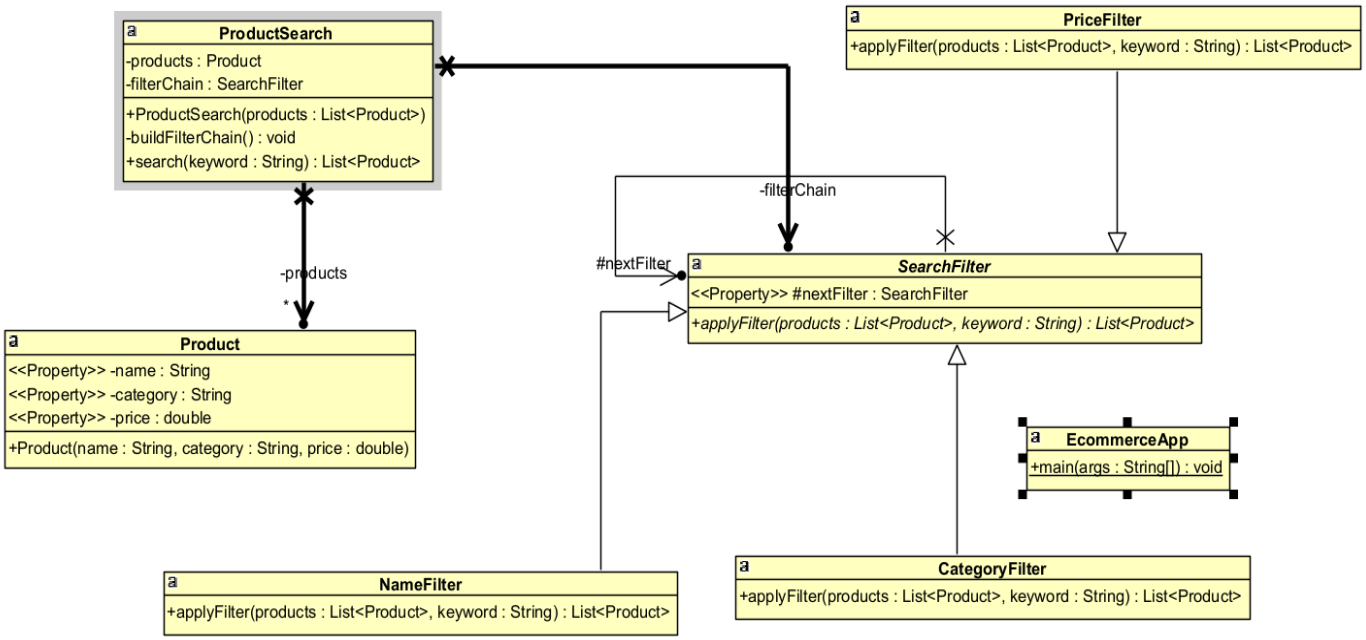
```
// Pembayaran gagal.
```

3. **Fitur: Pencarian Produk**

Design Pattern :Chain of Responsibility

Chain of Responsibility digunakan untuk mengizinkan beberapa objek untuk berusaha menangani permintaan secara berantai. Dalam kasus pencarian produk, Chain of Responsibility dapat digunakan untuk mencoba mencari produk berdasarkan berbagai kriteria (misalnya pencarian berdasarkan nama, harga, kategori) sampai produk yang sesuai ditemukan.

code:



CategoryFilter.java

```
public class CategoryFilter extends SearchFilter {
    @Override
    public List<Product> applyFilter(List<Product> products, String
keyword) {
        // Implementasi filter berdasarkan kategori produk
        // ...
    }
}
```

Kelas CategoryFilter merupakan implementasi dari SearchFilter untuk melakukan pencarian berdasarkan kategori produk.

EcommerceApp.java

```
public class EcommerceApp {
    public static void main(String[] args) {
        List<Product> products = new ArrayList<>();
        products.add(new Product("Laptop", "Electronics", 1000));
        products.add(new Product("T-Shirt", "Fashion", 25));
        products.add(new Product("Book", "Books", 20));
        products.add(new Product("Phone", "Electronics", 500));

        ProductSearch productSearch = new ProductSearch(products);

        // Cari produk dengan keyword "Phone"
        List<Product> searchResult = productSearch.search("Phone");

        // Tampilkan hasil pencarian
        for (Product product : searchResult) {
            System.out.println(product.getName() + " - " +
product.getCategory() + " - " + product.getPrice());
        }
    }
```

```
}
```

- Kelas EcommerceApp adalah kelas utama untuk menjalankan aplikasi.
- Di dalam method main(), kita membuat objek ProductSearch, lalu mencari produk dengan keyword "Phone" dan menampilkan hasil pencariannya.

NameFilter.java

```
public class NameFilter extends SearchFilter {
    @Override
    public List<Product> applyFilter(List<Product> products, String
keyword) {
        // Implementasi filter berdasarkan nama produk
        // ...
    }
}
```

Kelas NameFilter merupakan implementasi dari SearchFilter untuk melakukan pencarian berdasarkan nama produk.

PriceFilter.java

```
public class PriceFilter extends SearchFilter {
    @Override
    public List<Product> applyFilter(List<Product> products, String
keyword) {
        // Implementasi filter berdasarkan harga produk
        // ...
    }
}
```

Kelas PriceFilter merupakan implementasi dari SearchFilter untuk melakukan pencarian berdasarkan harga produk.

Product.java

```
public class Product {
```

```
    private String name;
    private String category;
    private double price;

    public Product(String name, String category, double price) {
        this.name = name;
        this.category = category;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getCategory() {
        return category;
    }

    public double getPrice() {
        return price;
    }
}
```

ProductSearch.java

```
public class ProductSearch {
    private List<Product> products;
    private SearchFilter filterChain;

    public ProductSearch(List<Product> products) {
        this.products = products;
        buildFilterChain();
    }

    private void buildFilterChain() {
        filterChain = new NameFilter();
        SearchFilter categoryFilter = new CategoryFilter();
        SearchFilter priceFilter = new PriceFilter();

        filterChain.setNextFilter(categoryFilter);
        categoryFilter.setNextFilter(priceFilter);
    }
```

```
    public List<Product> search(String keyword) {
        return filterChain.applyFilter(products, keyword);
    }
}
```

- Kelas ProductSearch berperan sebagai kelas utama yang menggunakan Chain of Responsibility Pattern.
- Memiliki atribut products untuk menyimpan daftar produk yang akan dicari.
- Memiliki atribut filterChain yang bertipe SearchFilter untuk menyimpan filter chain.
- Memiliki method buildFilterChain() untuk menginisialisasi filter chain dengan menambahkan filter pencarian berantai (NameFilter, CategoryFilter, PriceFilter).
- Memiliki method search() untuk mencari produk dengan menggunakan filter chain yang sudah dibuat sebelumnya.

SearchFilter.java

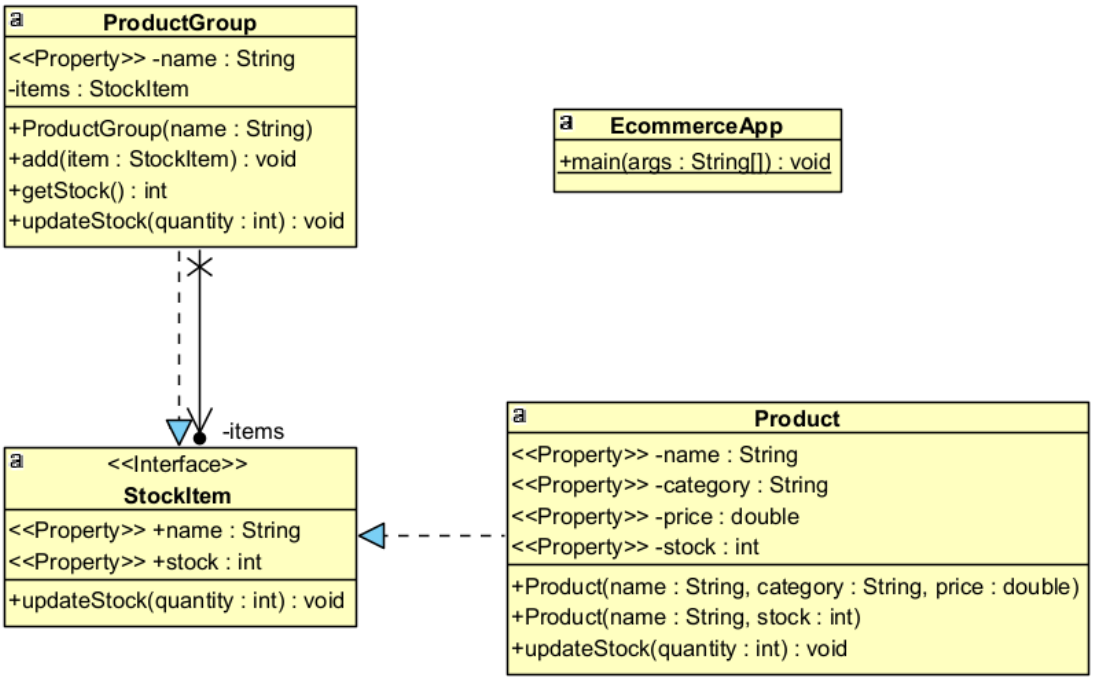
```
public abstract class SearchFilter {
    protected SearchFilter nextFilter;

    public void setNextFilter(SearchFilter nextFilter) {
        this.nextFilter = nextFilter;
    }

    public abstract List<Product> applyFilter(List<Product> products,
String keyword);
}
```

- Kelas SearchFilter adalah kelas abstrak yang berperan sebagai dasar untuk semua filter pencarian.
- Memiliki atribut nextFilter yang bertipe SearchFilter untuk menyimpan filter berikutnya dalam chain.
- Memiliki method setNextFilter untuk mengatur filter berikutnya dalam chain.
- Memiliki method abstrak applyFilter yang harus diimplementasikan oleh kelas turunannya untuk menangani filter pencarian sesuai kriteria tertentu.

Structural Design Patterns()

1. **Pengelolaan Stok**

- Composite Pattern memungkinkan kita untuk mengelompokkan objek individual dan objek grup ke dalam struktur pohon hirarki yang serupa, sehingga kita dapat mengelola mereka dengan cara yang seragam.
- Dalam konteks pengelolaan stok, kita dapat menggunakan Composite Pattern untuk mengelola stok produk secara hierarkis, di mana produk tunggal dianggap sebagai leaf node, dan kategori atau kelompok produk dianggap sebagai composite node. Setiap node (baik leaf node maupun composite node) dalam struktur akan memiliki metode yang sama, sehingga kita dapat mengelola stok dengan cara yang seragam tanpa memerhatikan apakah itu adalah produk tunggal atau kelompok produk.



EcommerceApp.java

```
public class EcommerceApp {
    public static void main(String[] args) {
        // Membuat beberapa produk tunggal
        Product product1 = new Product("Laptop", 10);
        Product product2 = new Product("T-Shirt", 50);
        Product product3 = new Product("Book", 30);

        // Membuat kelompok produk
        ProductGroup electronicsGroup = new
ProductGroup("Electronics");
        electronicsGroup.add(product1);
        electronicsGroup.add(product2);

        ProductGroup fashionGroup = new ProductGroup("Fashion");
        fashionGroup.add(product2);
        fashionGroup.add(product3);

        // Menampilkan stok produk dan kelompok produk
        System.out.println("Stok " + product1.getName() + ": " +
product1.getStock());
        System.out.println("Stok " + product2.getName() + ": " +
product2.getStock());
```

```
        System.out.println("Stok " + product3.getName() + ": " +
product3.getStock());

        System.out.println("Stok " + electronicsGroup.getName() +
": " + electronicsGroup.getStock());
        System.out.println("Stok " + fashionGroup.getName() + ":
" + fashionGroup.getStock());

        // Update stok kelompok produk
        electronicsGroup.updateStock(5);
        fashionGroup.updateStock(-10);

        // Menampilkan stok produk dan kelompok produk setelah
update
        System.out.println("Stok " + product1.getName() + ": " +
product1.getStock());
        System.out.println("Stok " + product2.getName() + ": " +
product2.getStock());
        System.out.println("Stok " + product3.getName() + ": " +
product3.getStock());

        System.out.println("Stok " + electronicsGroup.getName() +
": " + electronicsGroup.getStock());
        System.out.println("Stok " + fashionGroup.getName() + ":
" + fashionGroup.getStock());
    }
}
```

Product.java

```
public class Product implements StockItem {
    private String name;
    private int stock;

    public Product(String name, int stock) {
        this.name = name;
        this.stock = stock;
    }

    @Override
    public String getName() {
        return name;
```

```
    }

    @Override
    public int getStock() {
        return stock;
    }

    @Override
    public void updateStock(int quantity) {
        stock += quantity;
    }
}
```

ProductGroup.java

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

public class ProductGroup implements StockItem {
    private String name;
    private List<StockItem> items;

    public ProductGroup(String name) {
        this.name = name;
        items = new ArrayList<>();
    }

    public void add(StockItem item) {
        items.add(item);
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public int getStock() {
        int totalStock = 0;
        for (StockItem item : items) {
            totalStock += item.getStock();
        }
```

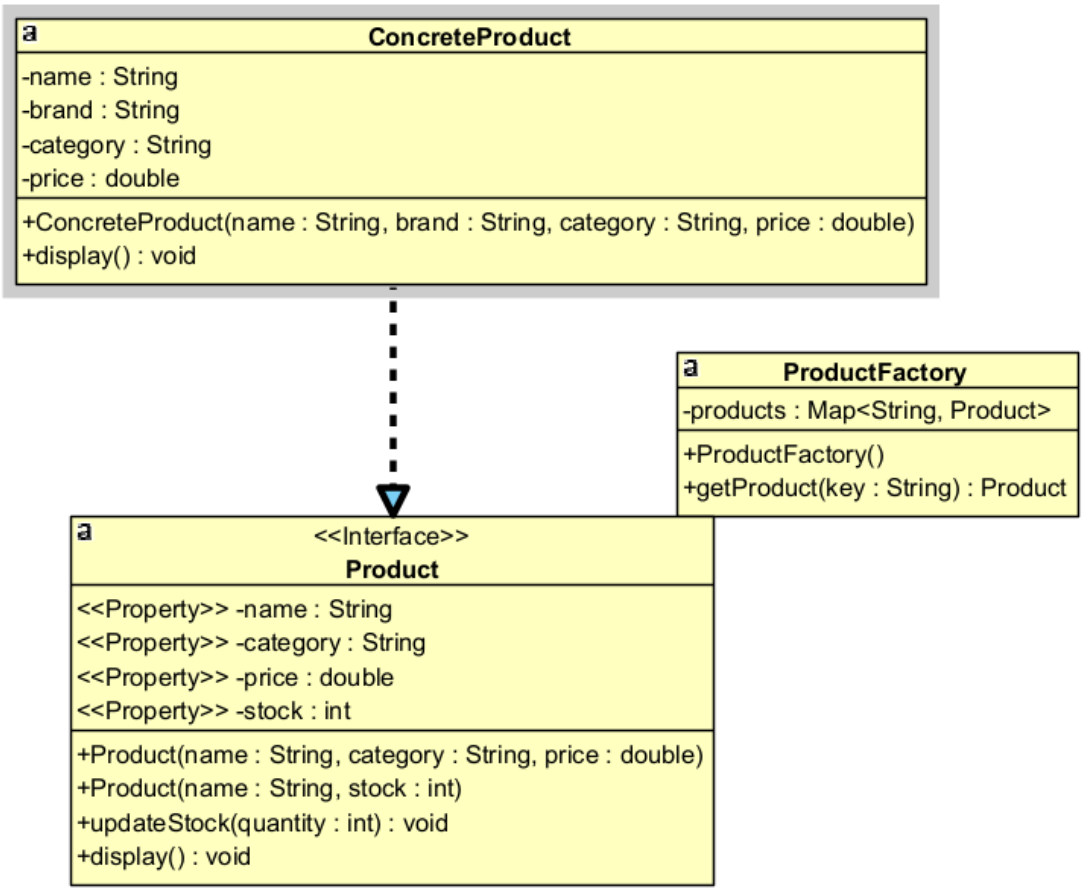
```
        return totalStock;
    }

    @Override
    public void updateStock(int quantity) {
        for (StockItem item : items) {
            item.updateStock(quantity);
        }
    }
}
```

StockItem.java

```
public interface StockItem {
    String getName();
    int getStock();
    void updateStock(int quantity);
}
```

2. Rekomendasi Produk

Rekomendasi produk pada e-commerce dengan menggunakan Flyweight Pattern dapat diimplementasikan untuk mengurangi penggunaan memori dengan berbagi data produk yang serupa. Pada e-commerce, terdapat banyak produk dengan atribut yang sama (misalnya, merek, kategori, dan deskripsi) di antara berbagai item yang dijual. Dengan menggunakan Flyweight Pattern, kita dapat menghindari pengulangan data yang sama untuk setiap produk yang serupa dan berbagi data tersebut secara efisien di seluruh sistem.



ConcreteProduct.java

```java
public class ConcreteProduct implements Product {
    private String name;
    private String brand;
    private String category;
    private double price;

    public ConcreteProduct(String name, String brand, String
category, double price) {
        this.name = name;
        this.brand = brand;
        this.category = category;
        this.price = price;
    }

    @Override
    public void display() {
        System.out.println("Product: " + name + ", Brand: " +
brand + ", Category: " + category + ", Price: " + price);
```

```
    }
}
```

EcommerceApp.java

```
EcommerceApppublic class EcommerceApp {
    public static void main(String[] args) {
        ProductFactory productFactory = new ProductFactory();

        // Mendapatkan produk dari pabrik
        Product product1 = productFactory.getProduct("123");
        Product product2 = productFactory.getProduct("456");
        Product product3 = productFactory.getProduct("123");

        // Menampilkan produk
        product1.display(); // Produk dengan key "123" diambil
dari pabrik
        product2.display(); // Produk dengan key "456" diambil
dari pabrik
        product3.display(); // Produk dengan key "123" diambil
dari cache, karena sudah ada sebelumnya
    }
}
```

Product.java

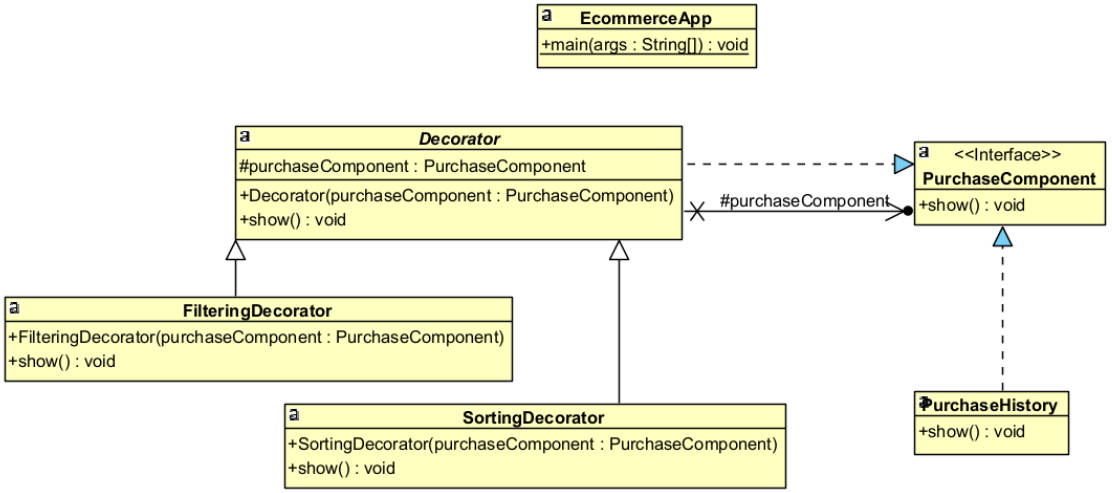
```
public interface Product {
    void display();
}
```

ProductFactory.java

```
import java.util.HashMap;
import java.util.Map;

public class ProductFactory {
    private Map<String, Product> products;

    public ProductFactory() {
```

```
        products = new HashMap<>();
    }

    public Product getProduct(String key) {
        // Jika produk dengan key yang sama sudah ada, kembalikan
produk yang sudah ada
        if (products.containsKey(key)) {
            return products.get(key);
        } else {
            // Jika produk dengan key yang sama belum ada, buat
produk baru dan tambahkan ke daftar produk
            Product product = new ConcreteProduct("Product " +
key, "Brand " + key, "Category " + key, 100.0);
            products.put(key, product);
            return product;
        }
    }
}
```

3. Riwayat Pembelian



Decorator.java

```
public abstract class Decorator implements PurchaseComponent {
    protected PurchaseComponent purchaseComponent;
```

```
    public Decorator(PurchaseComponent purchaseComponent) {
        this.purchaseComponent = purchaseComponent;
    }

    @Override
    public void show() {
        purchaseComponent.show();
    }
}
```

EcommerceApp.java

```
public class EcommerceApp {
    public static void main(String[] args) {
        PurchaseComponent purchaseHistory = new PurchaseHistory();
        purchaseHistory.show();

        System.out.println();

        PurchaseComponent sortedHistory = new SortingDecorator(new
PurchaseHistory());
        sortedHistory.show();

        System.out.println();

        PurchaseComponent filteredHistory = new FilteringDecorator(new
PurchaseHistory());
        filteredHistory.show();

        System.out.println();

        PurchaseComponent sortedAndFilteredHistory = new
SortingDecorator(new FilteringDecorator(new PurchaseHistory()));
        sortedAndFilteredHistory.show();
    }
}
```

PurchaseComponent.java

```
public interface PurchaseComponent {
```

```
    void show();
}
```

PurchaseHistory.java

```
public class PurchaseHistory implements PurchaseComponent {
    @Override
    public void show() {
        System.out.println("Menampilkan riwayat pembelian.");
    }
}
```

SortingDecorator.java

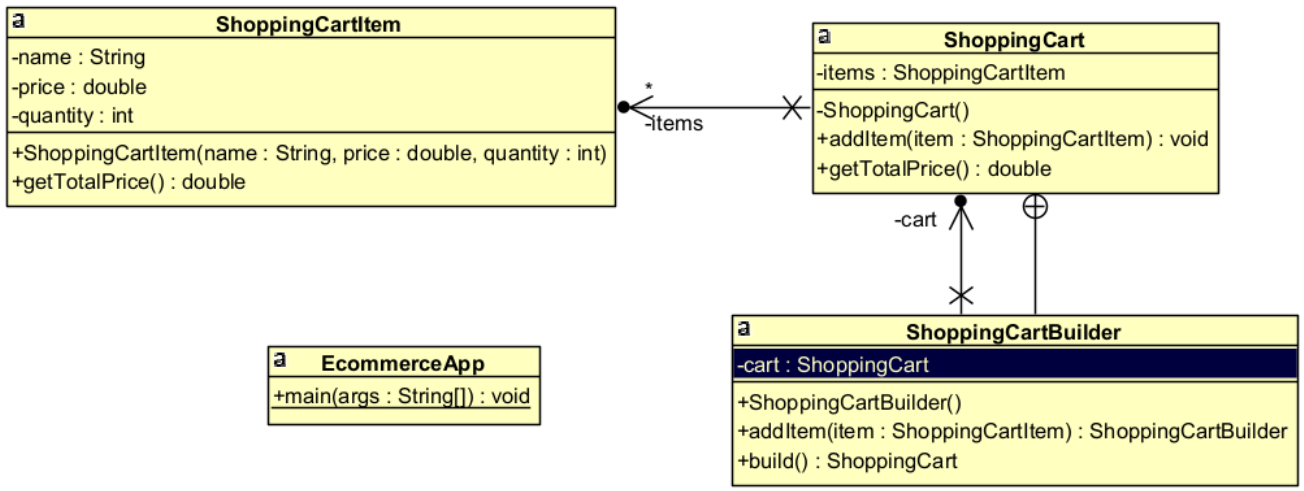
```
public class SortingDecorator extends Decorator {
    public SortingDecorator(PurchaseComponent purchaseComponent) {
        super(purchaseComponent);
    }

    @Override
    public void show() {
        super.show();
        System.out.println(" - Menambahkan fitur sorting pada riwayat
pembelian.");
    }
}

public class FilteringDecorator extends Decorator {
    public FilteringDecorator(PurchaseComponent purchaseComponent) {
        super(purchaseComponent);
    }

    @Override
    public void show() {
        super.show();
        System.out.println(" - Menambahkan fitur filter pada riwayat
pembelian.");
    }
}
```

Creational Design Patterns()

1. Keranjang Belanja

Untuk desain creational pada Keranjang Belanja, kita dapat menggunakan Builder Pattern. Builder Pattern digunakan untuk membuat objek kompleks dengan berbagai konfigurasi, sehingga membantu dalam membangun objek secara bertahap tanpa membuat konstruktor yang rumit.

Pada Keranjang Belanja, Builder Pattern dapat membantu dalam membangun keranjang dengan berbagai item dan konfigurasi yang berbeda. Setiap item dapat memiliki berbagai atribut seperti nama, harga, dan jumlah, dan Builder Pattern memungkinkan kita untuk menambahkan item ke dalam keranjang secara terstruktur.



EcommerceApp.java

```
public class EcommerceApp {

    public static void main(String[] args) {

        ShoppingCartItem item1 = new ShoppingCartItem("Laptop", 1500.0,
2);

          ShoppingCartItem item2 = new ShoppingCartItem("Mouse", 25.0,
3);



        ShoppingCart cart = new ShoppingCart.ShoppingCartBuilder()
```

```
                .addItem(item1)
                .addItem(item2)
                .build();

                System.out.println("Total Harga Keranjang: $" +
cart.getTotalPrice());
    }
}
```

ShoppingCart.java

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

public class ShoppingCart {
    private List<ShoppingCartItem> items;

    private ShoppingCart() {
        items = new ArrayList<>();
    }

    public void addItem(ShoppingCartItem item) {
        items.add(item);
    }

    public double getTotalPrice() {
        double totalPrice = 0;
        for (ShoppingCartItem item : items) {
            totalPrice += item.getTotalPrice();
        }
```

```
        return totalPrice;
    }

    public static class ShoppingCartBuilder {
        private ShoppingCart cart;

        public ShoppingCartBuilder() {
            cart = new ShoppingCart();
        }

        public ShoppingCartBuilder addItem(ShoppingCartItem item) {
            cart.addItem(item);
            return this;
        }

        public ShoppingCart build() {
            return cart;
        }
    }
}
```

ShoppingCartItem.java

```
public class ShoppingCartItem {
    private String name;
    private double price;
    private int quantity;

    public ShoppingCartItem(String name, double price, int quantity) {
        this.name = name;
```

```
        this.price = price;
        this.quantity = quantity;
    }

    public double getTotalPrice() {
        return price * quantity;
    }
}
```

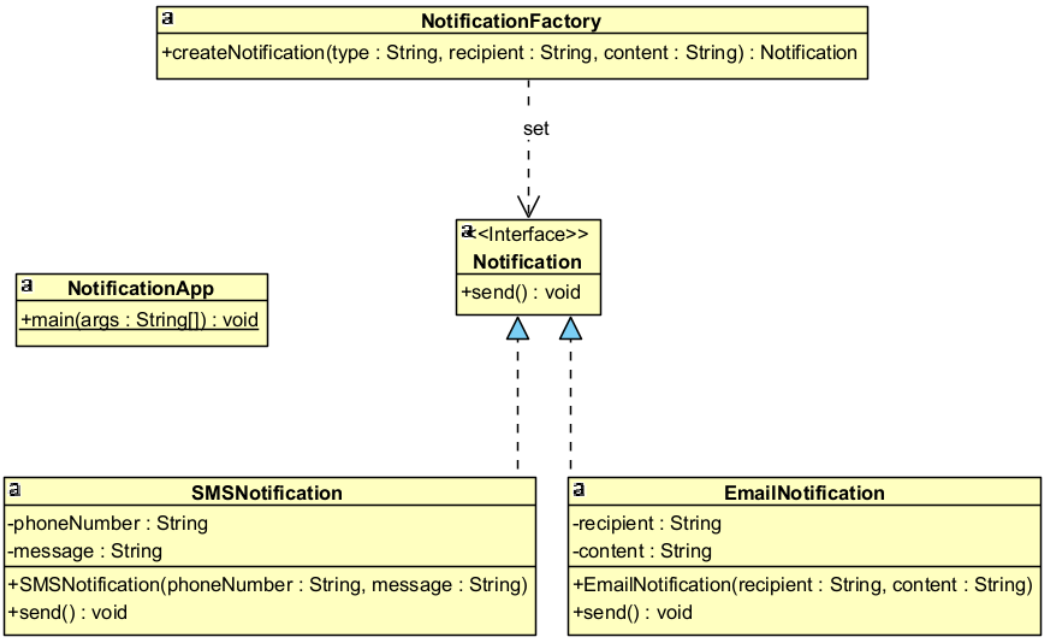
hasil keluaran

```
Total Harga Keranjang: $3075.0
```

2. **Notifikasi**

Design Pattern: factory method

Deskripsi: Observer Pattern dapat digunakan untuk memberikan notifikasi kepada pengguna tentang perubahan terkait dengan pesanan atau produk. Misalnya, memberikan notifikasi ketika status pesanan berubah, produk yang diminati kembali tersedia, atau ada perubahan harga pada produk tertentu.



EmailNotification.java

```java
public class EmailNotification implements Notification {
    private String recipient;
    private String content;

    public EmailNotification(String recipient, String content) {
        this.recipient = recipient;
        this.content = content;
    }

    @Override
    public void send() {
        System.out.println("Sending Email Notification to " +
recipient + ": " + content);
    }
}

public class SMSNotification implements Notification {
    private String phoneNumber;
    private String message;

    public SMSNotification(String phoneNumber, String message) {
        this.phoneNumber = phoneNumber;
```

```
        this.message = message;
    }

    @Override
    public void send() {
        System.out.println("Sending SMS Notification to " +
phoneNumber + ": " + message);
    }
}
```

Notification.java

```
public interface Notification {
    void send();
}
```

NotificationApp.java

```
public class NotificationApp {
    public static void main(String[] args) {
        NotificationFactory factory = new NotificationFactory();

        // Membuat notifikasi email
        Notification emailNotification =
factory.createNotification("email", "john@example.com", "Hello,
this is an email notification.");
        emailNotification.send();

        // Membuat notifikasi SMS
        Notification smsNotification =
factory.createNotification("sms", "+123456789", "Hello, this is
an SMS notification.");
        smsNotification.send();
    }
}


// Sending Email Notification to john@example.com: Hello, this is
an email notification.
```

```
// Sending SMS Notification to +123456789: Hello, this is an SMS
notification.
```

NotificationFactory.java

```
public class NotificationFactory {
    public Notification createNotification(String type, String
recipient, String content) {
        if (type.equalsIgnoreCase("email")) {
            return new EmailNotification(recipient, content);
        } else if (type.equalsIgnoreCase("sms")) {
            return new SMSNotification(recipient, content);
        } else {
            throw new IllegalArgumentException("Invalid
notification type: " + type);
        }
    }
}
```

hasil keluaran

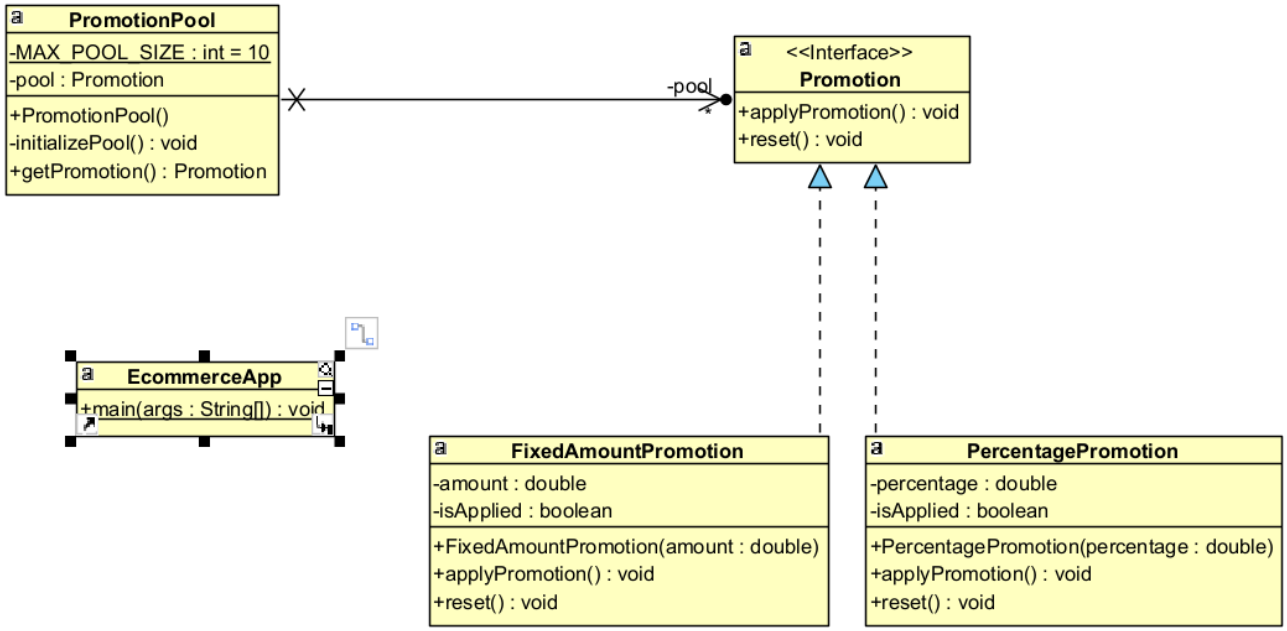
```
Sending Email Notification to john@example.com: Hello, this is an
email notification.
Sending SMS Notification to +123456789: Hello, this is an SMS
notification.
```

3. **Diskon dan Promosi**

Design Pattern: Decorator Pattern

Deskripsi: Decorator Pattern dapat digunakan untuk menambahkan fitur diskon atau promosi ke produk. Misalnya, kita

dapat menggunakan decorator untuk menambahkan diskon 10% atau promo beli satu gratis satu pada produk tertentu.



EcommerceApp.java

```java
public class EcommerceApp {
    public static void main(String[] args) {
        PromotionPool promotionPool = new PromotionPool();

        // Mendapatkan objek Diskon dari pool
        Promotion promotion1 = promotionPool.getPromotion();
        if (promotion1 != null) {
            promotion1.applyPromotion();
        }

        // Mendapatkan objek Diskon lainnya dari pool
        Promotion promotion2 = promotionPool.getPromotion();
        if (promotion2 != null) {
            promotion2.applyPromotion();
        }

        // Reset objek Diskon pertama dan kembalikan ke pool
        if (promotion1 != null) {
            promotion1.reset();
```

```
        }

        // Mendapatkan objek Diskon yang telah di-reset dari pool
        Promotion promotion3 = promotionPool.getPromotion();
        if (promotion3 != null) {
            promotion3.applyPromotion();
        }
    }
}
```

FixedAmountPromotion.java

```
public class FixedAmountPromotion implements Promotion {
    private double amount;
    private boolean isApplied;

    public FixedAmountPromotion(double amount) {
        this.amount = amount;
    }

    @Override
    public void applyPromotion() {
        System.out.println("Applying Fixed Amount Promotion: -$" +
amount);
        isApplied = true;
    }

    @Override
    public void reset() {
        isApplied = false;
    }
}

public class PercentagePromotion implements Promotion {
    private double percentage;
    private boolean isApplied;

    public PercentagePromotion(double percentage) {
        this.percentage = percentage;
    }

    @Override
    public void applyPromotion() {
```

```
        System.out.println("Applying Percentage Promotion: -" +
percentage + "%");
        isApplied = true;
    }

    @Override
    public void reset() {
        isApplied = false;
    }
}
```

Promotion.java

```
public interface Promotion {
    void applyPromotion();
    void reset();
}
```

PromotionPoo.java

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

public class PromotionPool {
    private static final int MAX_POOL_SIZE = 10;
    private List<Promotion> pool;

    public PromotionPool() {
        pool = new ArrayList<>();
        initializePool();
    }

    private void initializePool() {
        for (int i = 0; i < MAX_POOL_SIZE; i++) {
            pool.add(new FixedAmountPromotion(50.0)); // Contoh
inisialisasi pool dengan FixedAmountPromotion
        }
    }

    public Promotion getPromotion() {
```

```
        for (Promotion promotion : pool) {
            if (!promotion.isApplied()) {
                return promotion;
            }
        }
        return null;
    }
}
```

Dalam contoh di atas, Promotion adalah interface yang digunakan sebagai dasar untuk semua jenis promosi. Kelas FixedAmountPromotion dan PercentagePromotion adalah implementasi konkret dari Promotion. Kelas PromotionPool berfungsi sebagai kelas pengelola yang menyediakan objek Diskon dan Promosi dari pool yang telah diinisialisasi sebelumnya. Saat aplikasi memerlukan objek Diskon atau Promosi, ia akan mendapatkan objek dari pool yang tersedia (jika ada), dan ketika selesai digunakan, objek tersebut dapat di-reset dan dikembalikan ke pool untuk digunakan kembali. Dengan cara ini, kita dapat menghindari overhead pembuatan objek yang baru dan mengoptimalkan penggunaan sumber daya pada aplikasi e-commerce.

linkgithub : https://github.com/ksmora/DESIGN_PATTERN/tree/main