LAPORAN PENGERJAAN PRAKTIKUM PERTEMUAN 11
TEKNIK PEMROGRAMAN
ANALISIS DESIGN PATTERN

Oleh:

Ihsan Fauzi 241524048

PROGRAM STUDI D4-TEKNIK INFORMATIKA

JURUSAN TEKNIK KOMPUTER DAN INFORMATIKA

POLITEKNIK NEGERI BANDUNG

2025

DAFTAR ISI

BAB I
PENDAHULUAN

Dalam pengembangan perangkat lunak, desain yang baik merupakan fondasi yang sangat penting untuk menciptakan sistem yang maintainable, scalable, dan robust. Design pattern adalah solusi yang telah terbukti untuk mengatasi masalah-masalah umum dalam desain perangkat lunak. Design pattern menyediakan template atau blueprint yang dapat digunakan kembali untuk menyelesaikan masalah desain yang sering muncul dalam pengembangan software.

Tujuan dari praktikum ini adalah:

1. Memahami konsep dan implementasi Design Pattern
2. Menganalisis kelebihan dan kekurangan dari setiap jenis Design Pattern
3. Mengevaluasi dan mengetahui bagaimana Design Pattern dapat meningkatkan fleksibilitas dan maintainability kode
4. Memahami hubungan antara Decorator Pattern dengan prinsip-prinsip desain perangkat lunak

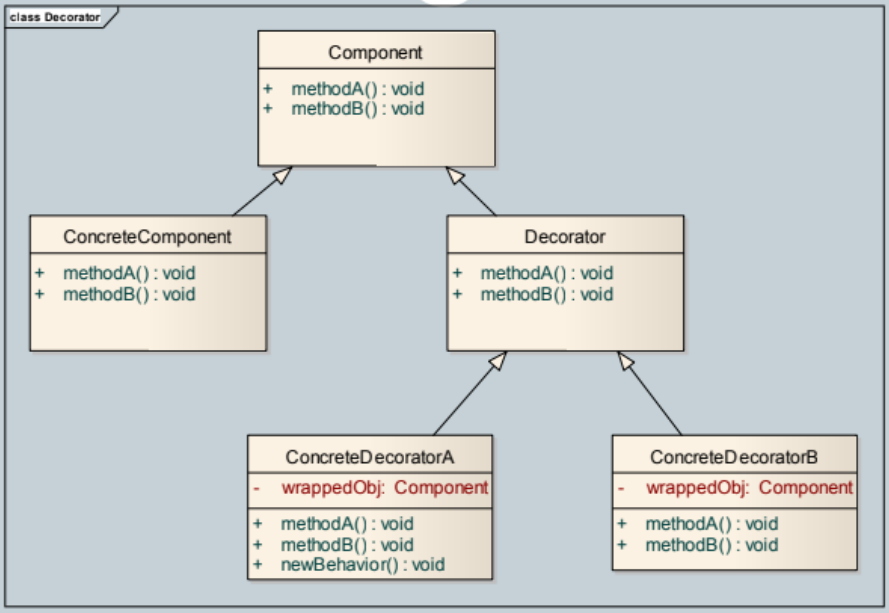
BAB II
ANALISIS STRUCTURAL PATTERNS

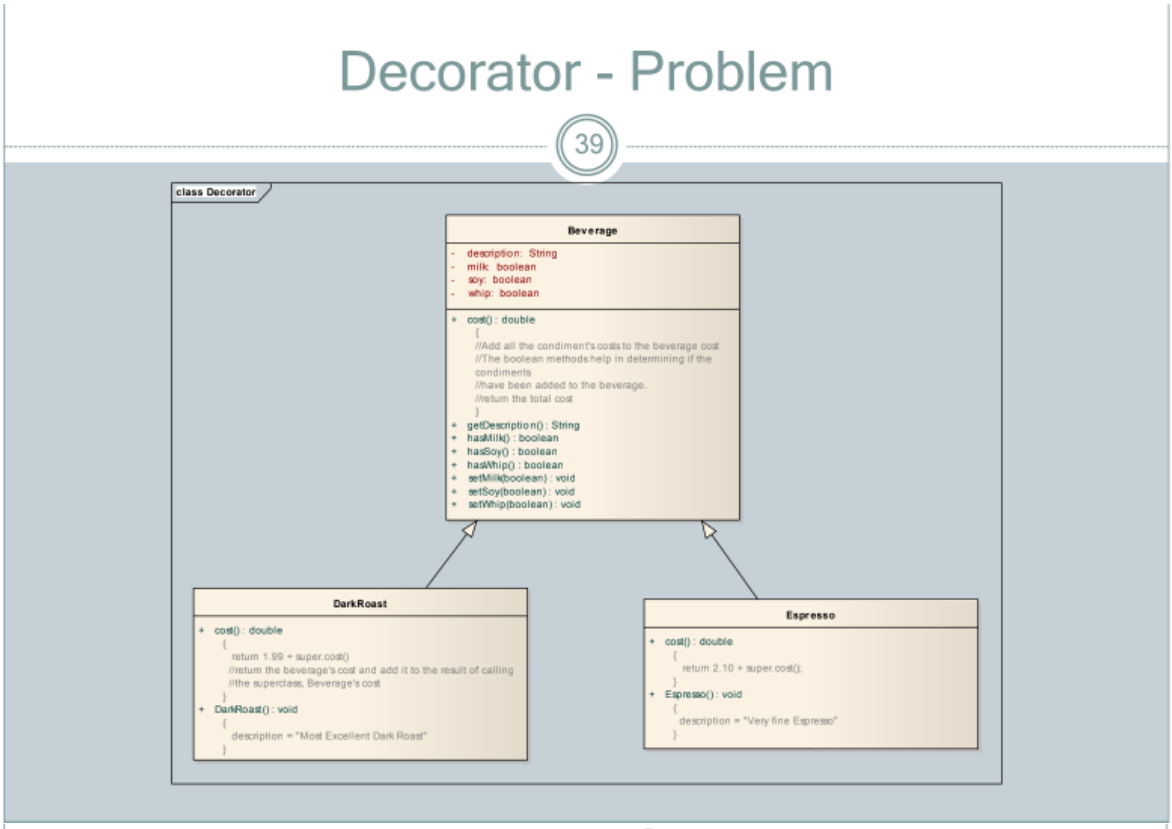
A. Decorator

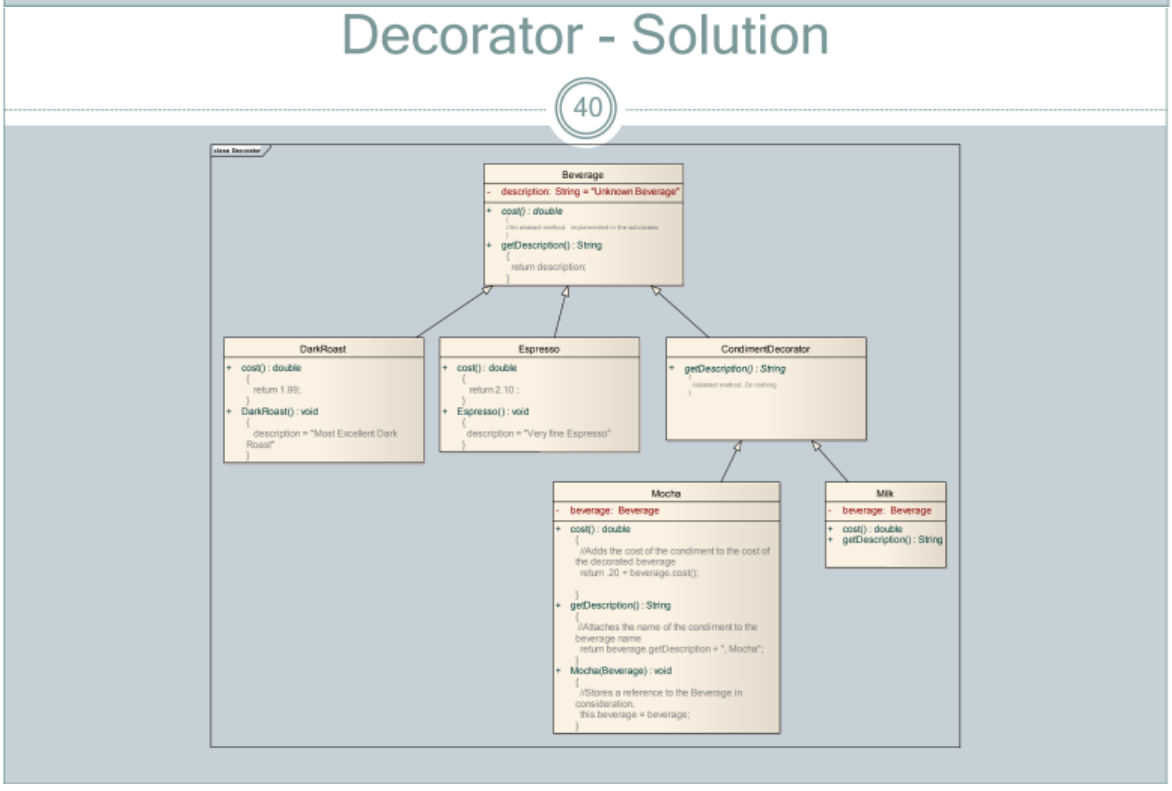
1. Masalah
    - Sebuah class memiliki fitur dasar, tetapi seiring waktu, pengguna membutuhkan variasi fitur tambahan (misal: log, enkripsi, validasi, UI, topping pada makanan/minuman).
    - Solusi seperti subclassing (inheritance) membuat struktur hierarki class menjadi rumit dan sulit dirawat, apalagi jika kombinasi fitur banyak.
    - Kode sulit diubah tanpa merusak class yang sudah ada (kode lama harus diubah untuk tambah fitur baru).
2. Kapan digunakan
    - Saat ingin menambahkan fitur tambahan ke objek tertentu secara dinamis, tanpa mengubah kode class aslinya.
    - Saat ada banyak kombinasi fitur opsional, yang jika dipecah dengan inheritance akan menghasilkan class yang terlalu banyak.
    - Saat ingin mematuhi prinsip Open-Closed, yaitu class harus terbuka untuk ekstensi, tapi tertutup untuk modifikasi.
    - Saat ingin menghindari pewarisan yang berlebihan dan lebih mengandalkan komposisi.
3. Solusi
    - Buat interface/abstraksi Component yang digunakan oleh objek asli dan decorator.
    - Buat class ConcreteComponent yang mewakili objek dasar (misal: Espresso, TextBox, dll).
    - Buat class Decorator yang:
      Memiliki referensi ke objek Component (komposisi)
      Mengimplementasikan interface yang sama (agar bisa diperlakukan seperti objek aslinya)
      Menambahkan fitur tambahan di dalam metode override, sambil tetap memanggil metode objek asli.
    - Objek bisa didekorasi berlapis-lapis, setiap decorator membungkus objek sebelumnya.
4. Konsekuensi
    a. Pro
    - Menambahkan fungsionalitas secara fleksibel ke objek, tanpa ubah class utama.
    - Menghindari subclassing berlebihan.
    - Bisa membuat kombinasi fitur yang dinamis dan reusable.
    - Mendukung prinsip Open-Closed.
    - Menggunakan komposisi, bukan inheritance.
    b. Kontra
    - Kompleksitas meningkat saat banyak decorator ditumpuk (wrapper berlapis-lapis).
    - Debugging dan tracing sulit karena banyak lapisan wrapper.
    - Tidak cocok jika fitur tambahan harus mengakses properti spesifik objek asli (karena semuanya dibungkus interface).

5. Diagram Class

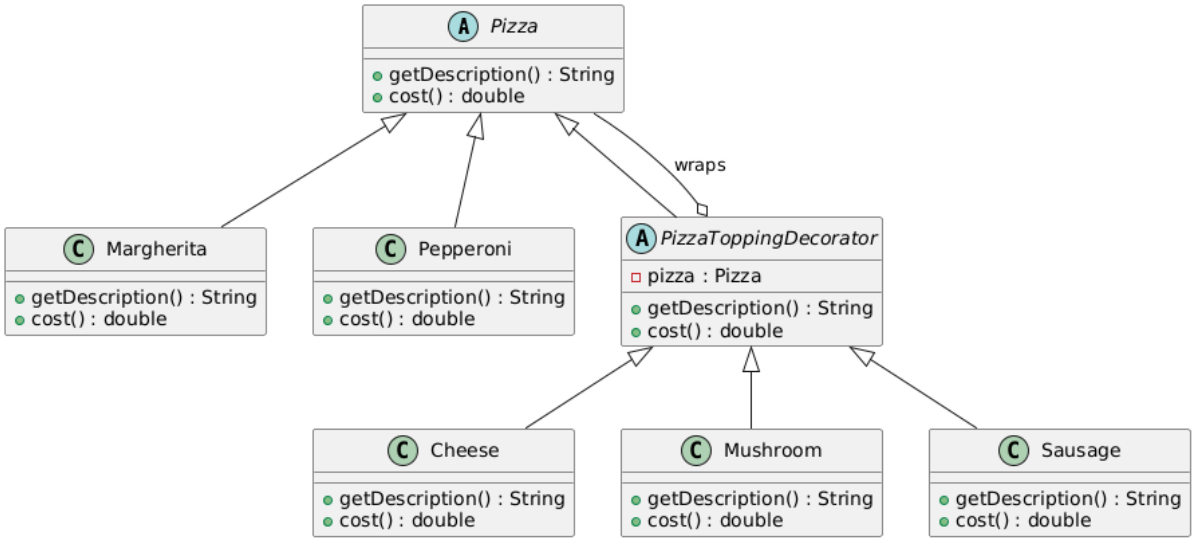
Decorator - Class diagram







6. Diagram Class dari source code



7. Source code
   Pizza Decorator

   Component (Pizza)

```
abstract class Pizza {
    public String getDescription() {
        return "Unknown Pizza";
    }
    public abstract double cost();
}
```

   Concrete Components (Basic Pizza)

```
class Margherita extends Pizza {
    public String getDescription() {
        return "Margherita";
    }
    public double cost() {
        return 5.00;
    }
}
```

```
class Pepperoni extends Pizza {
    public String getDescription() {
        return "Pepperoni";
    }
    public double cost() {
        return 6.50;
    }
}
```

   Decorator Base Class

```
abstract class PizzaToppingDecorator extends Pizza {
    protected Pizza pizza;

    public PizzaToppingDecorator(Pizza pizza) {
```

```
        this.pizza = pizza;
    }
}
```

Concrete Decorators (Toppings)

```
class Cheese extends PizzaToppingDecorator {
    public Cheese(Pizza pizza) {
        super(pizza);
    }

    public String getDescription() {
        return pizza.getDescription() + ", Extra Cheese";
    }

    public double cost() {
        return 1.00 + pizza.cost();
    }
}
```

```
class Mushroom extends PizzaToppingDecorator {
    public Mushroom(Pizza pizza) {
        super(pizza);
    }

    public String getDescription() {
        return pizza.getDescription() + ", Mushroom";
    }

    public double cost() {
        return 0.75 + pizza.cost();
    }
}
```

```
class Sausage extends PizzaToppingDecorator {
    public Sausage(Pizza pizza) {
        super(pizza);
    }

    public String getDescription() {
        return pizza.getDescription() + ", Sausage";
    }

    public double cost() {
        return 1.25 + pizza.cost();
    }
}
```

Main

```
public class PizzaOrder {
    public static void main(String[] args) {
```

```
        Pizza pizza = new Margherita();
        pizza = new Cheese(pizza);
        pizza = new Mushroom(pizza);
        pizza = new Sausage(pizza);

        System.out.println(pizza.getDescription()); // Margherita, Extra
Cheese, Mushroom, Sausage
        System.out.println("Total Price: $" + pizza.cost()); // 5.00 + 1.00
+ 0.75 + 1.25 = 8.00
    }
}
```

8. Analisis Source code
    - Inheritance (IS-A): Semua class pizza dan topping adalah Pizza.
    - Composition (HAS-A): Setiap decorator menyimpan referensi ke objek Pizza.
    - Polymorphism: Semua objek bertindak sebagai Pizza, meskipun telah dihias.
    - Open-Closed Principle: Tambahan fitur (topping) tanpa mengubah class lama.
9. Inti dari source code
    - Pattern Decorator terletak pada struktur class yang bersifat hierarkis dan fleksibel. Fungsionalitas tambahan (topping) tidak ditanamkan langsung dalam Margherita, tapi dipisah ke dalam decorator seperti Cheese, Mushroom, dll.
    - Polimorfisme: semua decorator bertindak sebagai Pizza.
    - Komposisi: decorator memiliki referensi ke objek Pizza, memungkinkan penambahan fitur tanpa mengubah kode lama.
10. Kesimpulan

    Decorator Pattern = Menambahkan fitur ke objek, tanpa mengubah kode objek asli.

    Decorator Pattern adalah design pattern yang memungkinkan kita menambahkan fungsionalitas baru ke objek secara dinamis tanpa mengubah struktur class aslinya. Pattern ini menggunakan komposisi dan pewarisan agar objek bisa dibungkus oleh "lapisan-lapisan" tambahan (decorator), seperti topping pada pizza.

B. Proxy

1. Penjelasan

   Memungkinkan kita membuat objek perantara (proxy) yang dapat mengontrol akses ke objek asli. Jika proxy ini diimplementasi maka klien atau user tidak bisa mengakses objek asli jika belum berhasil melewati proxy. Di proxy juga bisa ditambahkan logika tambahan, seperti pemeriksaan keamanan atau penundaan instalasi.

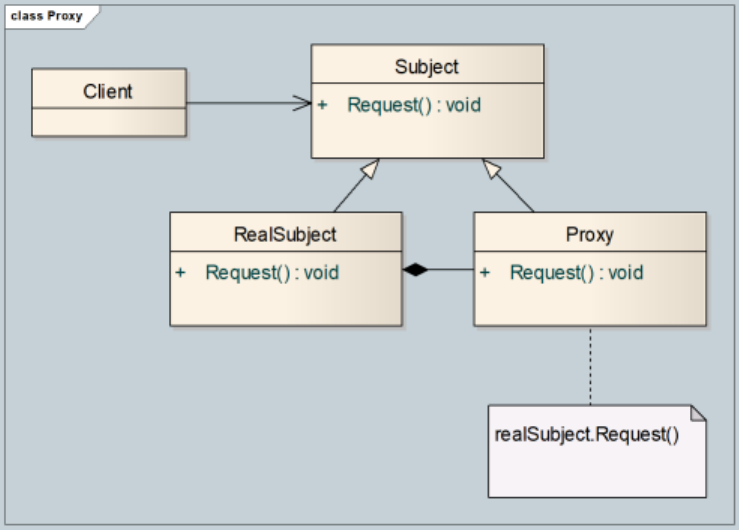
2. Kapan digunakan
   - Saat ingin mengontrol akses ke objek tertentu (misal: hanya admin yang bisa mengubah data
   - Saat ingin data aman karena tidak bisa langsung diakses oleh user
   - Saat ingin menambahkan logging, caching, validasi, atau otorisasi sebelum akses objek utama
3. Solusi
   - Buat interface subject yang didefinisikan untuk objek asli dan proxy.
   - Implemntasikan RealSubject sebagai objek sesungguhnya
   - Buat proxy yang:
     Menyimpan referensi ke objek nyata
     Mengimplementasikan interface yang sama (subject)
     Menyisipkan logika tambahan seperti hak akses, logging, caching, dll
4. Konsekuensi
   a. Pro
   - Keamanan: Dapat membatasi akses terhadap objek sensitig
   - Fleksibilitas: Bisa menyisipkan logika tambahan tanpa ubah objek asli.
   - Modularitas: Memisahkan logika kontrol dan logika inti sistem.

   b. Kontra
   - Kompleksitas meningkat karena penambahan lapisan (proxy)
   - Pemeliharaan lebih sulit jika banyak jenis proxy berbeda (virtual, remote, protection)
   - Overhead kecil saat harus meneruskan setiap permintaan melalui proxy

5. Diagram Class

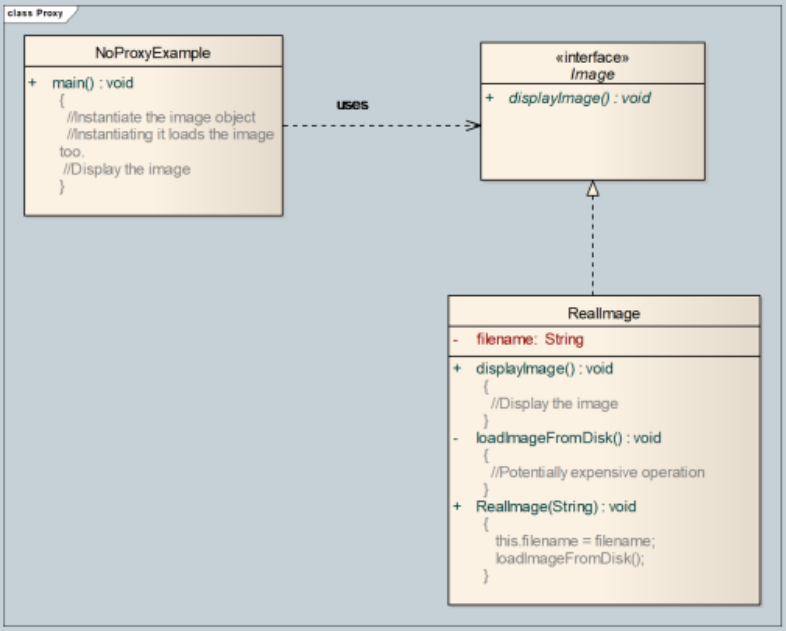
Proxy - Class diagram

43

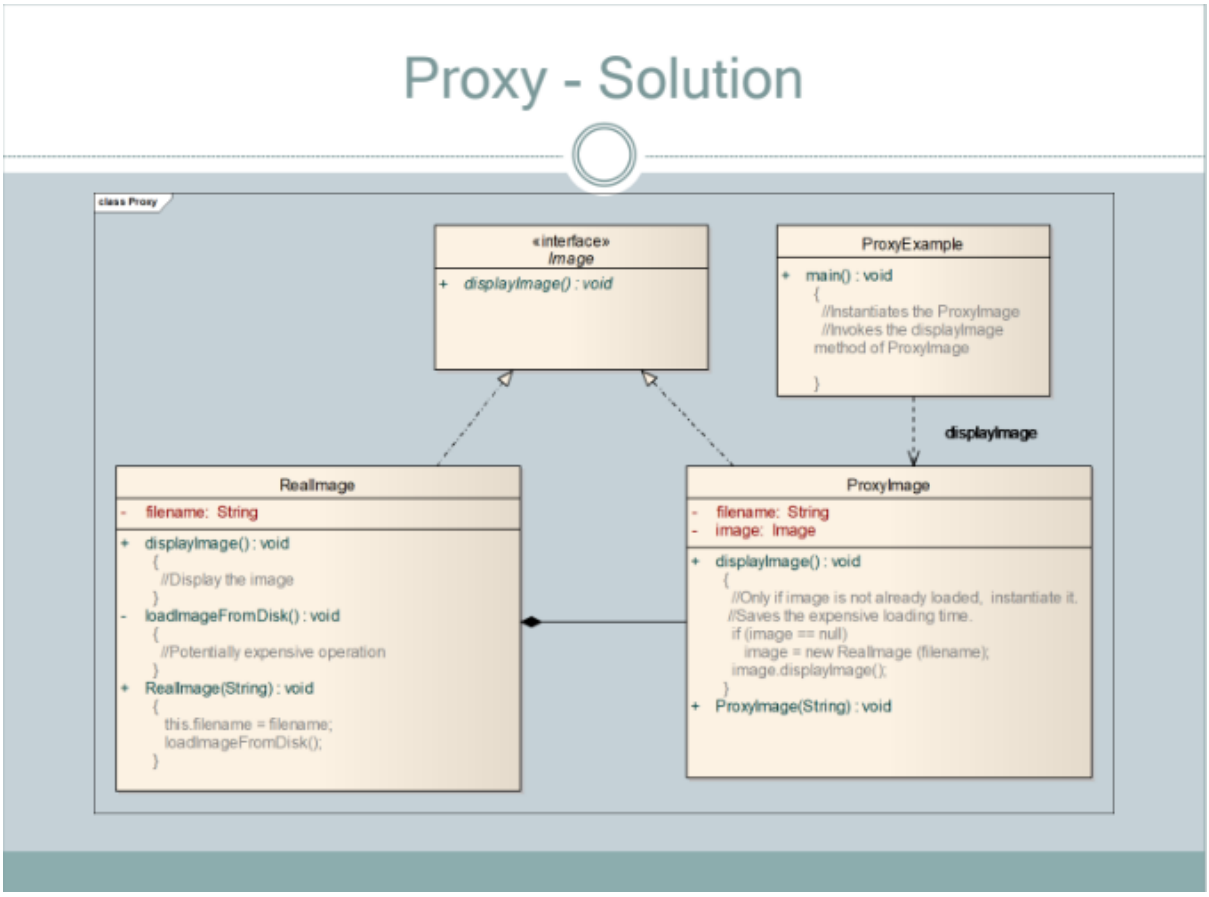


Proxy - Problem

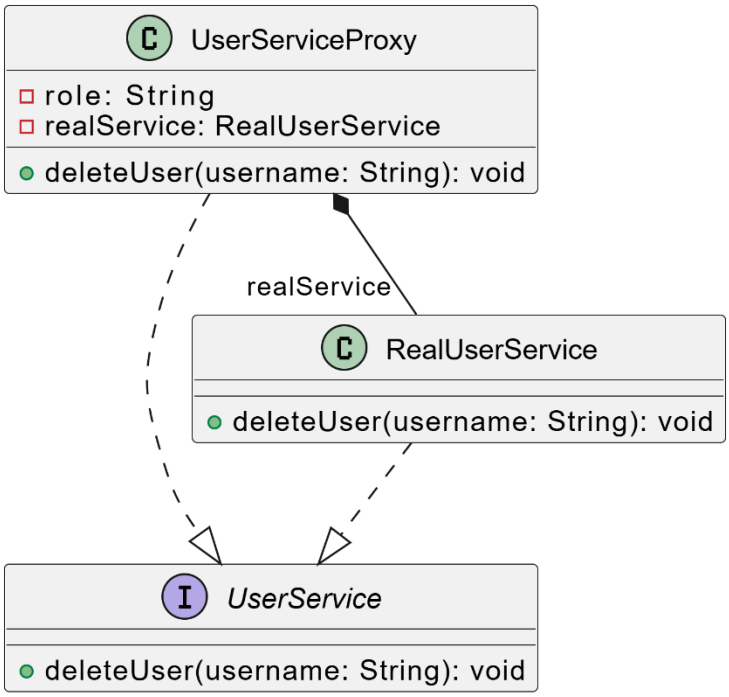
44



11/28/2018



6. Diagram Class dari source code



7. Source code (Protection Proxy)

// Interface Subject

```
interface UserService {
    void deleteUser(String username);
}
```

// RealSubject

```
class RealUserService implements UserService {
    public void deleteUser(String username) {
        System.out.println("User '" + username + "' has been deleted.");
    }
}
```

// Proxy

```
class UserServiceProxy implements UserService {
    private RealUserService realService;
    private String role;

    public UserServiceProxy(String role) {
        this.role = role;
        this.realService = new RealUserService();
    }

    public void deleteUser(String username) {
        if ("admin".equalsIgnoreCase(role)) {
            realService.deleteUser(username);
        } else {
            System.out.println("Access denied. Only admin can delete
users.");
        }
    }
}
```

// Client

```
public class ProxyDemo {
    public static void main(String[] args) {
        UserService adminService = new UserServiceProxy("admin");
        adminService.deleteUser("ihsan"); // Output: User 'ihsan' has been
deleted.

        UserService userService = new UserServiceProxy("user");
        userService.deleteUser("fauzi"); // Output: Access denied. Only
admin can delete users.
    }
}
```

8. Analisis Source code
    - Inheritance (IS-A): Proxy dan RealUserService sama-sama UserService.
    - Composition (HAS-A): Proxy menyimpan referensi ke RealUserService.
    - Polymorphism: Client memanggil UserService tanpa tahu di baliknya ada proxy.

- Open-Closed Principle: Bisa menambah fitur (validasi, log) tanpa ubah kode utama.

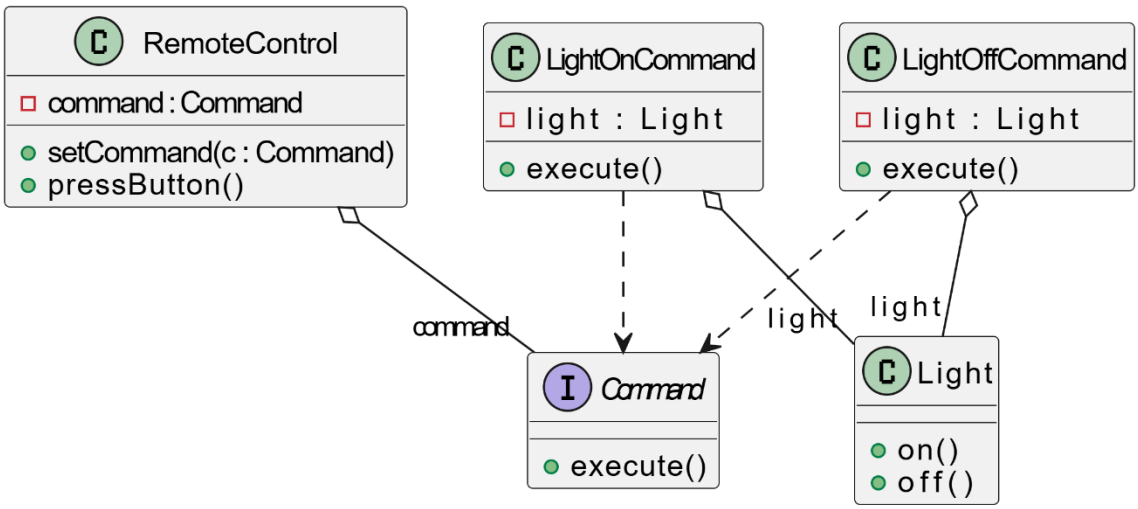
9. Inti dari source code
    - UserServiceProxy bertindak sebagai lapisan kontrol akses terhadap RealUserService.
    - Client tidak tahu mana objek asli dan mana proxy.
    - Fitur tambahan (validasi role) ditambahkan tanpa modifikasi class utama.
10. Kesimpulan
    Proxy Pattern = Wakil/penjaga objek asli.
    yang memungkinkan kontrol akses dan logika tambahan (misalnya validasi peran) tanpa harus mengubah objek utama, dengan cara menyisipkan proxy di antara client dan real object. Cocok untuk sistem yang butuh kontrol hak akses, efisiensi, dan keamanan.

BAB III
ANALISIS BEHAVIORAL PATTERNS

A. Command

1. Masalah
   - Sistem ingin memisahkan objek yang meminta perintah (invoker) dari objek yang menjalankan aksi (receiver).
2. Kapan digunakan
   - Saat ingin memisahkan perintah dari eksekusinya.
   - Saat membuat macro (perintah dijalankan sekaligus).
3. Solusi
   - Buat interface dengan metode execute().
   - Setiap aksi diwakili oleh class konkret yang mengimplementasikan command.
   - Aksi sebenernya dilakukan oleh Receiver (misalnya lampu).
   - Invoker menyimpan referensi ke Command, dan hanya tahu cara mengeksekusinya, tidak tahu detail aksi.
4. Konsekuensi
   a. Pro
   - Memisahkan logika perintah dan eksekusi.
   - Memudahkan testing.
   - Menyederhanakan struktur invoker
   b. Kontra
   - Terlalu banyak class: satu perintah = satu class
   - Struktur bisa terlihat berat untuk aksi yang sangat sederhana.

5. Diagram Class dari source code



6. Source code (Remote Control)

```
interface Command {
    void execute();
}
```

```
public class Light {
    public  void on(){
        System.out.println("Light On");
    }

    public  void off (){
        System.out.println("Light off");
    }
}
```

```
public class LightOffCommand implements  Command{
    private  Light light;

    LightOffCommand (Light light){
        this.light = light;
    }

    public  void execute (){
        light.off();
    }
}
```

```
public class LightOnCommand implements  Command{
    private Light light;

    public LightOnCommand (Light light){
        this.light = light;
    }

    @Override
    public void execute() {
        light.on();
    }
}
```

```
//Invoker
public class RemoteControl {
    private Command command;

    public void setCommand (Command command){
        this.command = command;
    }

    public void pressButton(){
        if (command != null){
            command.execute();
        } else {
            System.out.println("No command set.");
        }
    }
}
```

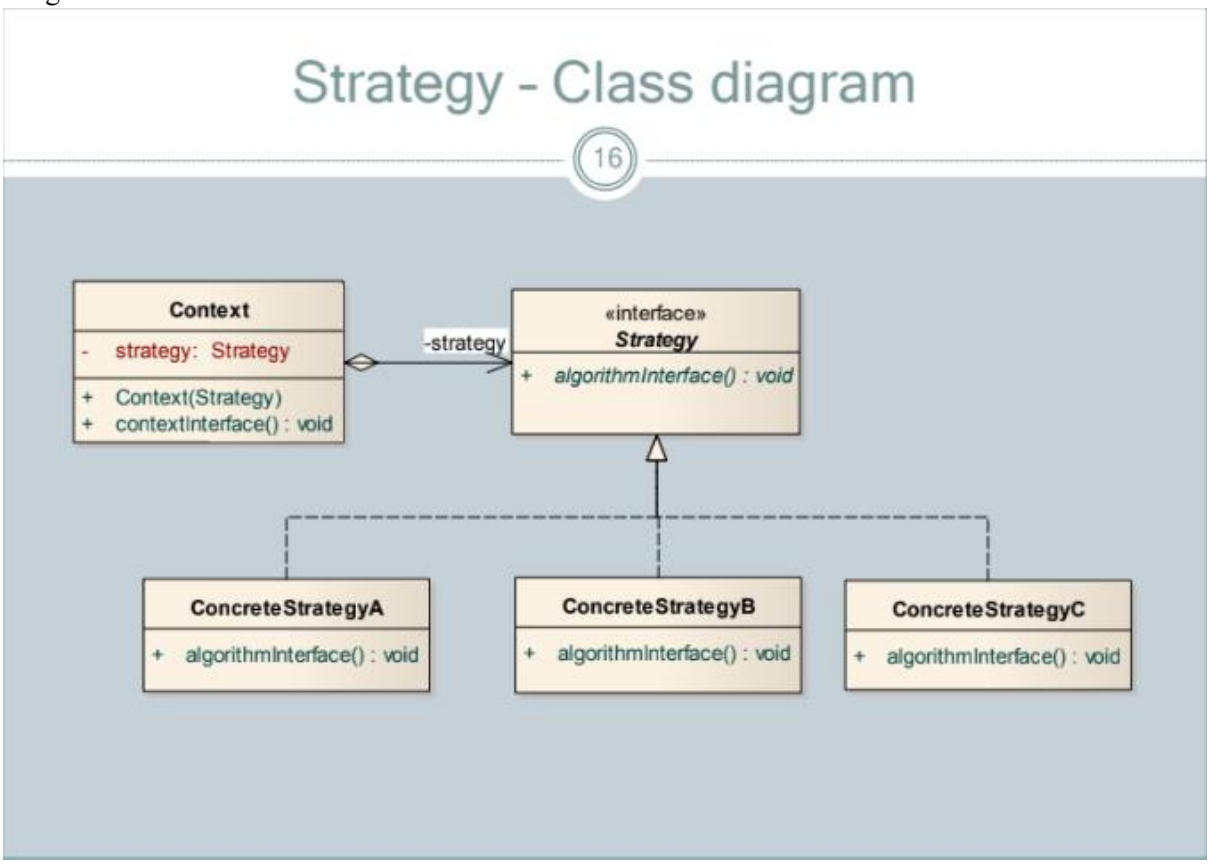
```
public class Main {
    public static  void main (String[] args){
        Light light = new Light();
        RemoteControl remote = new RemoteControl();

        remote.setCommand(new LightOnCommand(light));
        remote.pressButton();

        remote.setCommand((new LightOffCommand(light)));
        remote.pressButton();
    }
}
```

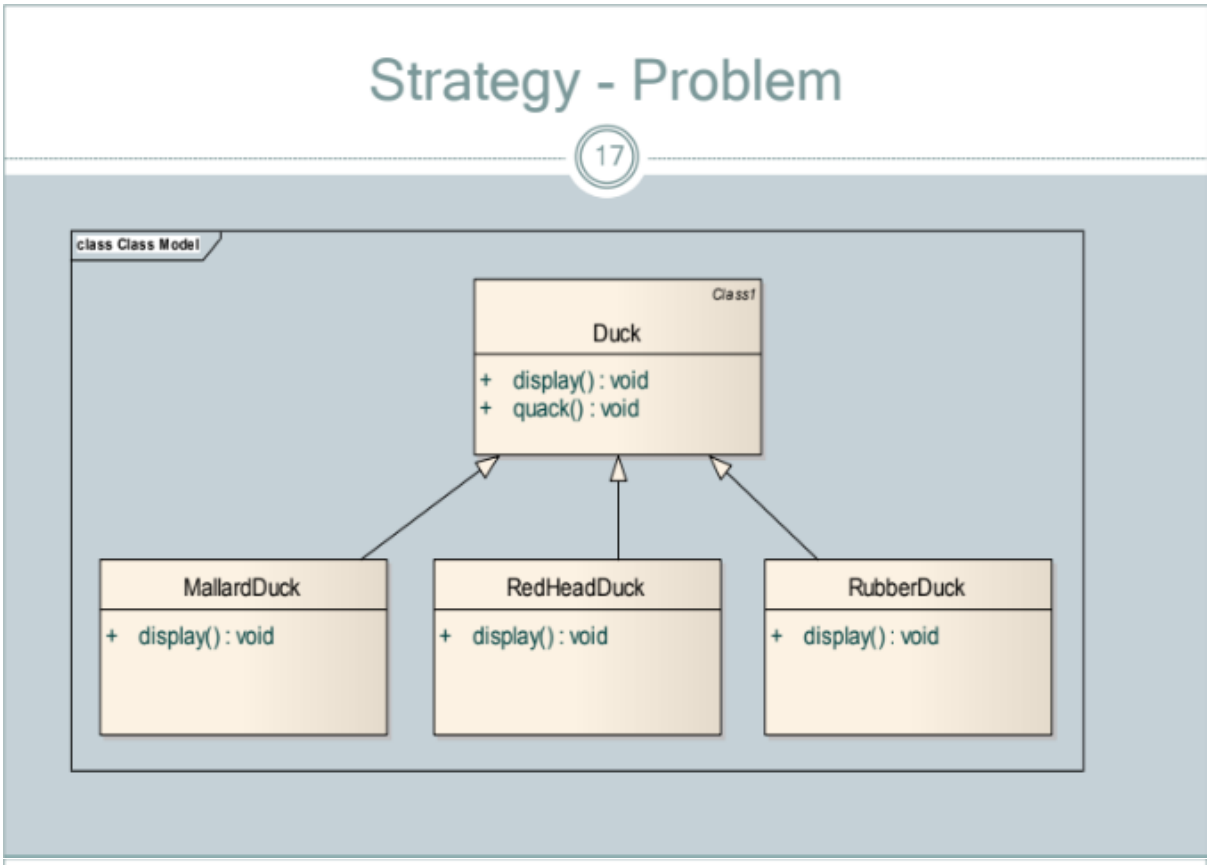
7. Analisis Source code
    - Inheritance (IS-A): LightOnCommand dan LightOffCommand adalah Command.
    - Composition (Has-a): Command memiliki referensi ke Light (receiver).
    - Polymorphism: RemoteControl bisa menerima command apapun yang implementasi dari interface Command.
8. Inti dari source code
    - RemoteControl adalah Invoker: tidak tahu detail perintah, hanya tahu ada tombol.
    - LightOnCommand dan LightOffCommand adalah ConcreteCommand.
    - Light adalah Receiver yang melakukan aksi sebenarnya.
9. Kesimpulan
   Command Pattern= Membungkus permintaan sebagai objek.
   Command pattern memisahkan siapa yang meminta aksi dan siapa yang melakukan aksi.

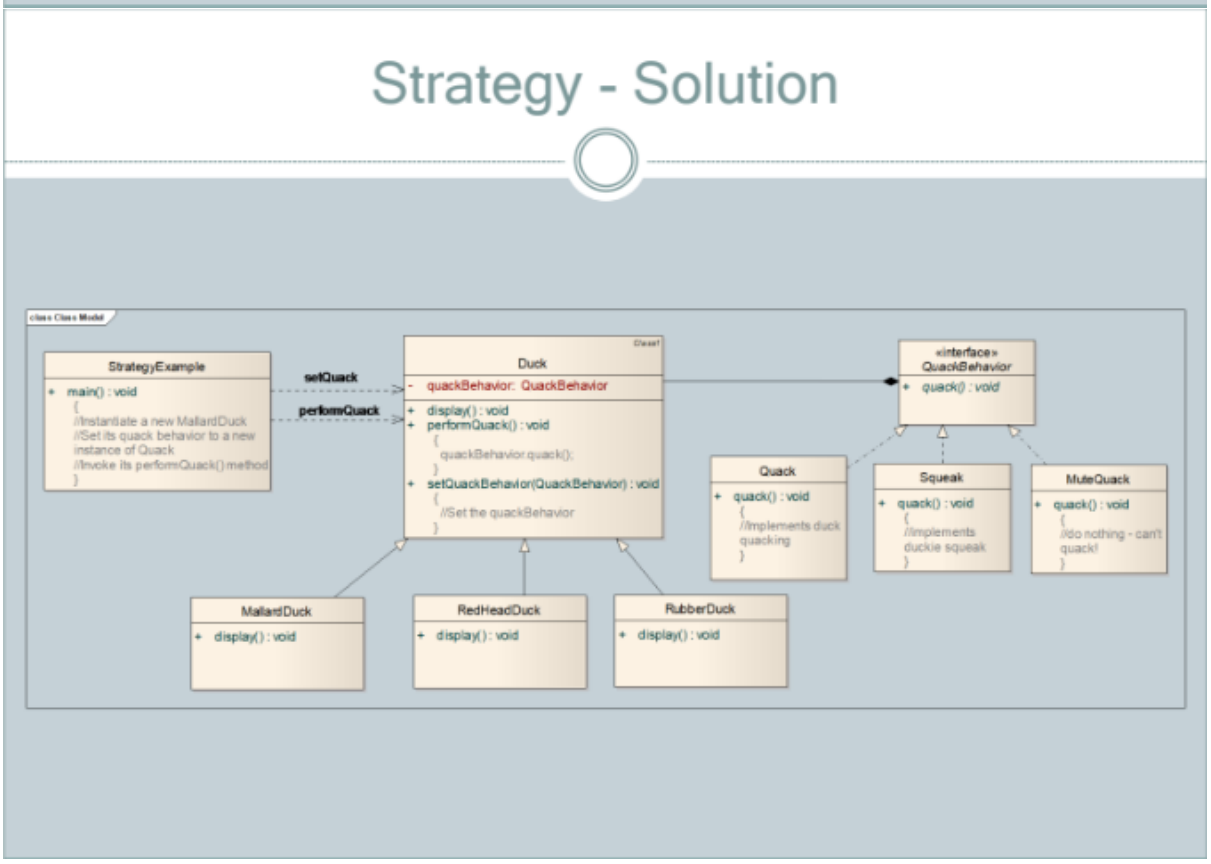
B. Strategy

1. Pengertian
    - Digunakan untuk mendefinisikan sekumpulan algoritma, membungkus masing-masing algoritma ke dalam class yang terpisah, dan membuatnya dapat saling dipertukarkan. Dengan begini, strategi dapat diganti secara dinamis saat runtime, tanpa mengubah class yang menggunakannya
2. Kapan digunakan
    - Menghindari kode if-else atau switch-case.
    - Saat strategi bisa berubah di runtime.
    - Jika ingin melakukan satu tugas dari beberapa tugas.
3. Solusi
    - Buat interface sebagai strategi umum.
    - Buat class strategi konkret (pilihan strategi)
    - Buat class yang memiliki referensi ke strategi dan dapat menggantinya kapan pun.
4. Konsekuensi
    a. Pro
    - Terpisah & mudah diuji.
    - Fleksibel & modular
    b. Kontra
    - Terlalu banyak class jika strategi banyak.
    - Keterpisahan bisa membuat kebingungan.

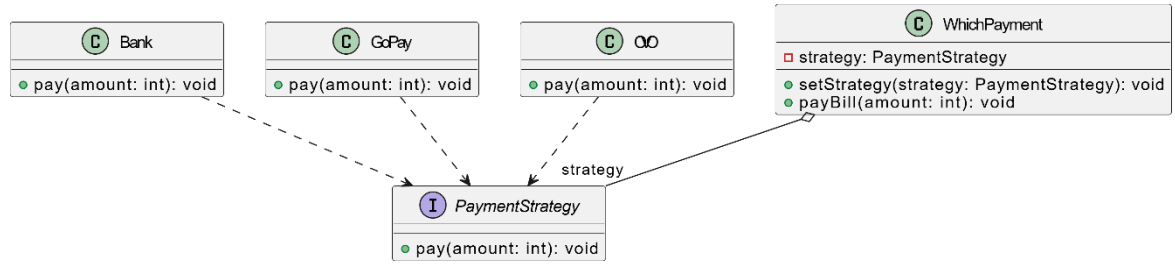
5. Diagram Class







6. Diagram Class dari source code



7. Source code (Metode pembayaran)
   Cara bayar bisa pilih GoPay, OVO, transfer, dll. Bisa ganti strategi saat runtime

```
//strategi interface
public interface PaymentStrategy {
    void pay (int amount);
}
```

```
public class Bank implements  PaymentStrategy{
    public void pay(int amount) {
        System.out.println("Bayar Rp "+ amount + "via Bank.");
    }
}
```

```
public class GoPay implements  PaymentStrategy{
    public void pay (int amount){
        System.out.println("Bayar Rp "+ amount + "via GoPay.");
    }
}
```

```
public class Ovo implements PaymentStrategy{
    public void pay (int amount){
        System.out.println("Bayar Rp " + amount + "via OVO.");
    }
}
```

```
public class WhichPayment {
    private PaymentStrategy strategy;

    public  void setStrategy (PaymentStrategy strategy){
        this.strategy= strategy;
    }

    public  void payBill (int amount){
        if (strategy != null){
            strategy.pay(amount);
        } else {
            System.out.println("Strategi payment belum dipilih.");
        }
    }
}
```

```
public class Main {
    public static void main (String[] args){
        WhichPayment resultchoose = new WhichPayment();
```

```
        //ini harusnya gagal karena belum ada pilihan paymentnya
        resultchoose.payBill(1000);

        resultchoose.setStrategy(new GoPay());
        resultchoose.payBill(10000);

        resultchoose.setStrategy(new Ovo());
        resultchoose.payBill(50000);

        resultchoose.setStrategy(new Bank());
        resultchoose.payBill(20000);
    }
}
```

8. Analisis Source code
    - IS-A: Semua opsi pembayaran adalah paymentstrategy
    - Has-a: Resultchoose dengan strategi sebagai komposisi
9. Inti dari source code
    - Strategi pembayaran punya class masing-masing
    - Resultchoose bisa berganti strategi kapanpun
10. Kesimpulan
    Memungkinkan sebuah system untuk memilih perilaku yang berbeda secara dinamis tanpa harus memodifikasi class utama. Tidak harus menggunakan switch-case beruntun. Memilih salah satu strategi dari beberapa strategi.

BAB IV
ANALISIS CREATIONAL PATTERNS

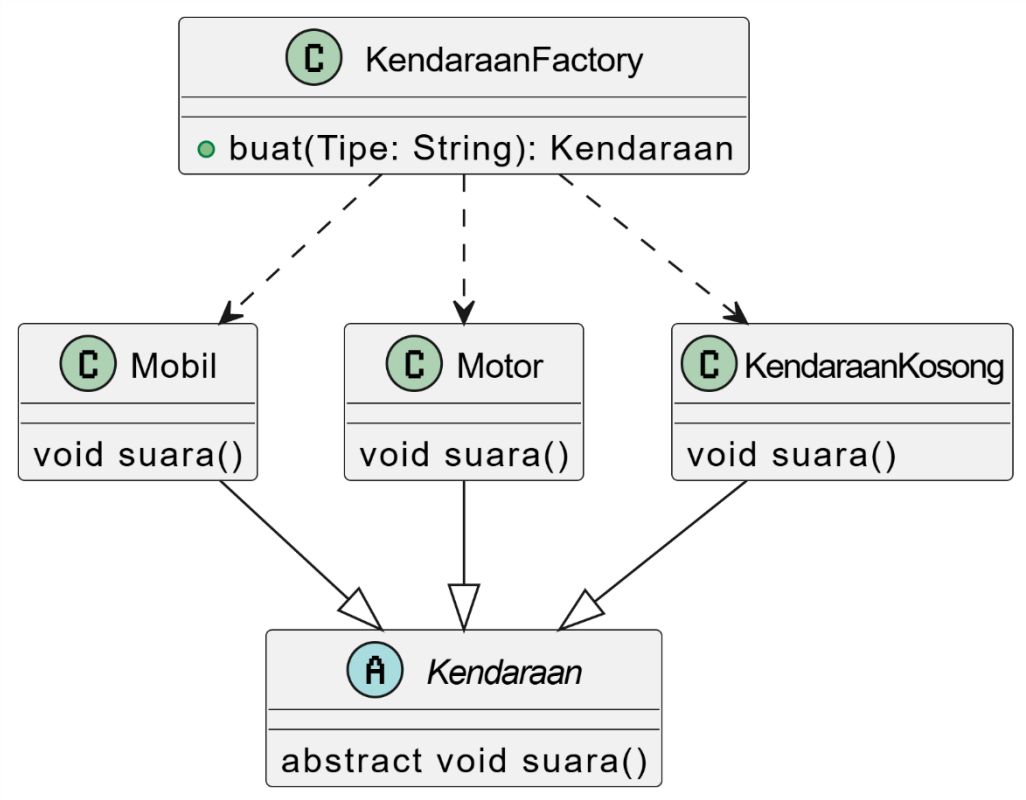
A. Factory

1. Pengertian

   Digunakan untuk membuat objek tanpa mengungkapkan logika instansiasi secara langsung kepada client. Singkatnya kita tidak perlu new di main. Tapi saat di main, jika client/dev ingin suatu objek, maka akan dibuat oleh factory yang akan return objek sesuai dengan parameter atau kondisi yang diinginkan client.

2. Kapan digunakan
   - Ingin menghindari penggunaan new di banyak tempat
   - Class yg dijadikan objek hanya yg diminta client atau kondisi tertentu
   - Ingin menyembunyikan detail pembuatan objek
3. Solusi
   - Buat superclass atau interface untuk semua produk (contoh kendaraan).
   - Buat subclass konkret (contoh mobil, motor, dll).
   - Buat class yang akan menentukan objek mana yang akan dibuat
4. Konsekuensi
   a. Pro
   - Pemisahan logika pembuaran objek
   - Client tidak perlu tahu cara membuat objek
   b. Kontra
   - Meningkatkan kompleksitas karena akan banyak class, dan kondisi tambahan
   - Factory bisa menjadi rumit jika tidak dikelola dengan baik

5. Diagram Class dari source code



6. Source code (Memilih jenis kendaraan berdasarkan permintaan)

```
public abstract class Kendaraan {
    abstract void suara ();
}
```

```
// untuk kendaraan yang tidak ada di spesifikasi seperti mobil dan
motor
// agar tidak return null dan masih bisa panggil suaranya
public class KendaraanKosong extends Kendaraan{
    public void suara (){
        System.out.println("Kendaraan tidak dikenali atau tidak
ada suaranya");
    }
}
```

```
public class Mobil extends Kendaraan{
    public void suara (){
        System.out.println("stutututut");
    }
}
```

```
public class Motor extends Kendaraan{
    public void suara (){
        System.out.println("ngengg mberrr cret");
    }
}
```

```
public class KendaraanFactory {
    static Kendaraan buat (String Tipe){
        if (Tipe.equals("Mobil")){
            return new Mobil();
        }
        else if (Tipe.equals("Motor")){
            return new Motor();
        }
        else{
            return new KendaraanKosong();
        }
    }
}
```

```
public class Main {
    public static void main (String[] args){
        Kendaraan Kendaraan0 =
KendaraanFactory.buat("failedcase");
        Kendaraan0.suara();

        Kendaraan Kendaraan1 = KendaraanFactory.buat("Mobil");
        Kendaraan1.suara();

        Kendaraan Kendaraan2 = KendaraanFactory.buat("Motor");
        Kendaraan2.suara();
    }
}
```

Output

```
Kendaraan tidak dikenali atau tidak ada suaranya
stutututut
ngengg mberrr cret
```

7. Analisis Source code
    - IS-A: Mobil dan Motor adalah turunan dari kendaraan
    - Has-a: KendaraanFactory mengembalikan objek Kendaraan.
8. Inti dari source code
    - Pembuatan objek (new) hanya melalui class KendaraanFactory.
    - Menyederhanakan client dan membuat kode lebih fleksibel.
9. Kesimpulan
   Digunakan jika membutuhkan berbagai jenis objek, tanpa harus membuat seluruh class menjadi objek, jadi bisa hanya memilih salah satu atau lebih. Juga digunakan untuk menyembunyikan detail pembuatan dari pengguna. Design ini lebih modular, dan lebih clean dalam hal pemisahan tanggung jawab.

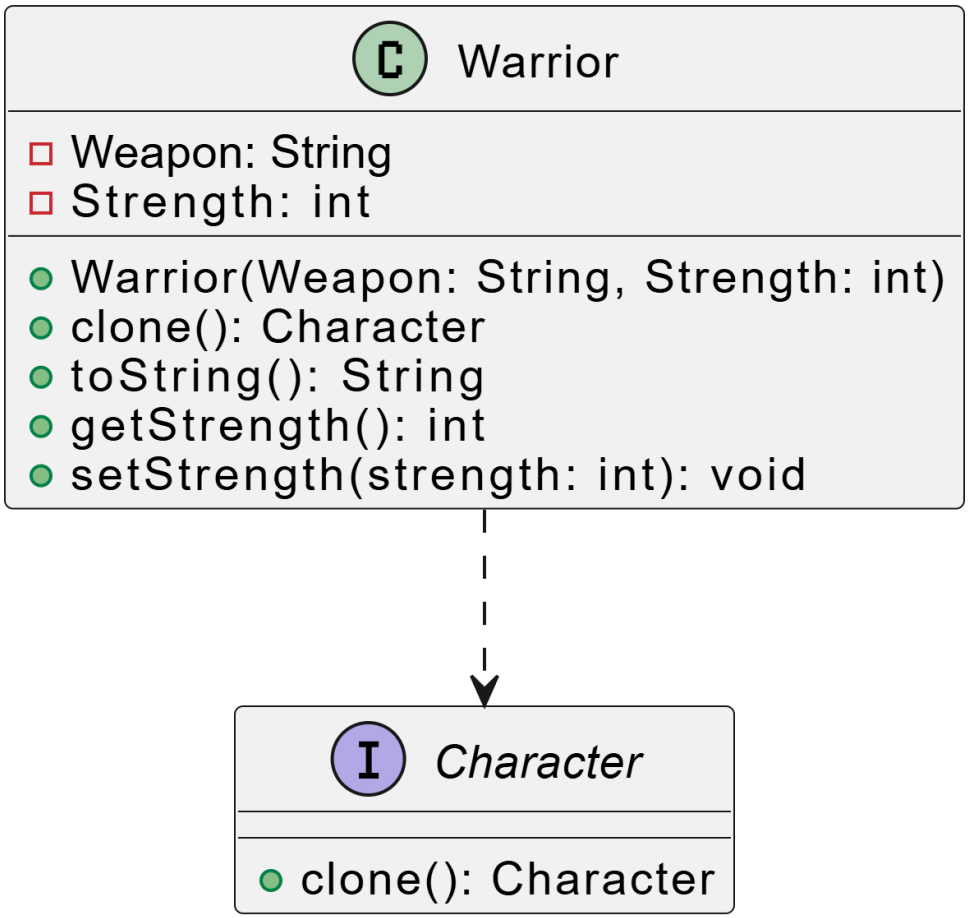
B. Prototype

1. Pengertian

   Design pattern yang digunakan untuk membuat objek baru dengan menyalin (clone) objek yang sudah ada dan telah memiliki konfigurasi atau nilai tertentu. Berbeda dengan membuat objek baru dari awal (new), cloning menghemat waktu saat kita membutuhkan duplikasi dari objek yang sudah diubah atau dikonfigurasi sebelumnya.
2. Kapan digunakan
   - Saat objek memiliki proses pembuatan yang berat, hampir sama, sama persis.
   - Saat ingin menghindari duplikasi kode inisiasi.
   - Saat ingin membuat salinan objek tanpa harus tahu tipe pastinya.
3. Solusi
   - Buat interface atau class yang meng-override method clone().
   - Objek yang ingin diklon barus mengimplemntasikan interface Cloneable
   - Alih-alih memanggil new, client cukup memanggil clone() pada objek yang sudah ada.
4. Konsekuensi
   a. Pro
   - Lebih Efisien karena tidak perlu new
   - Fleksibel
   - Mengurangi duplikasi kode

   b. Kontra
   - Perlu implementasi manual di awal menggunakan method clone() di setiap class.
   - **Bisa sulit jika objek punya relasi kompleks antar field**

5. Diagram Class dari source code



6. Source code (Clone character)

```
public interface Character extends Cloneable{
    Character clone();
}
```

```
public class Warrior implements Character{
    private String Weapon;
    private int Strength;

    public Warrior (String Weapon, int Strength){
        this.Weapon=Weapon;
        this.Strength = Strength;
    }

    @Override
    public Character clone() {
        return new Warrior(this.Weapon, this.Strength);
    }

    @Override
    public String toString() {
        return "Warrior with "+ Weapon + "Strength" + Strength;
    }
```

```
    public int getStrength() {
        return Strength;
    }

    public void setStrength(int strength) {
        this.Strength = strength;
    }
}
```

```
public class Main {
    public static  void main (String[] args){
        Warrior original = new Warrior("vandal", 160);
        System.out.println("Warrior asal 160");
        original.setStrength(156);
        System.out.println("Warrior strength rubah ke 156");

        Warrior copy= (Warrior) original.clone();

        System.out.println("Original : "+ original);
        System.out.println("Copy : "+copy);
    }
}
```

Output

```
Warrior asal 160
Warrior strength rubah ke 156
Original : Warrior with vandalStrength156
Copy : Warrior with vandalStrength156
```

7. Analisis Source code
    - original.clone() → membuat objek baru dari template.
    - Tanpa buat ulang dengan constructor.
    - Main tidak tahu detail bagaimana objek Warrior dikloning.
    - IS-A: Warrior adalah Character.
8. Inti dari source code
    - Objek diklon dari template, bukan dibuat ulang.
    - Sangat berguna untuk sistem yang butuh banyak objek serupa.

9. Kesimpulan
   Pattern ini menyalin objek yang sudah ada, tanpa perlu mengetahui detail proses pembuatan objek tersebut.

BAB IV
KESIMPULAN

Dalam pengembangan perangkat lunak, kita sering dihadapkan pada masalah yang berulang — entah itu soal bagaimana menambahkan fitur ke objek, mengatur hak akses, memisahkan logika perintah, atau membuat objek baru secara fleksibel. Design pattern hadir sebagai solusi dari masalah-masalah tersebut.

Selama analisis ini, kita sudah mempelajari beberapa pattern yang masing-masing punya kegunaan dan cara kerja yang berbeda:

- **Decorator** membantu kita menambahkan fitur ke objek secara fleksibel, tanpa harus mengubah struktur aslinya. Ibaratnya seperti menambahkan topping ke pizza, kita bisa lapis-lapis menambahkan fitur sesuai kebutuhan.
- **Proxy** digunakan saat kita ingin mengontrol akses ke suatu objek. Misalnya, hanya user admin yang boleh menghapus data. Proxy bertindak seperti penjaga gerbang sebelum sebuah objek bisa digunakan.
- **Command** memisahkan siapa yang meminta dan siapa yang melakukan aksi. Cocok digunakan saat kita ingin tombol-tombol tertentu bisa diatur ulang, seperti remote yang bisa diprogram.
- **Strategy** digunakan saat kita punya banyak cara untuk menyelesaikan satu tugas, seperti berbagai metode pembayaran. Pola ini memudahkan kita untuk menukar-tukar strategi tanpa harus menulis ulang logika utama.
- **Factory** berguna saat kita ingin membuat objek tanpa memperlihatkan proses pembuatannya ke client. Dengan pola ini, kita tinggal minta objek dari pabriknya, dan pabrik akan tentukan sendiri apa yang harus diberikan berdasarkan input.
- **Prototype** memungkinkan kita untuk menggandakan objek yang sudah ada, terutama saat kita ingin menyalin objek yang sudah dikonfigurasi, bukan membuat dari nol lagi. Ini sangat efisien, apalagi kalau proses inisialisasi objek cukup berat.