POLBAN

Jurusan Teknik Komputer dan Informatika

Politeknik Negeri Bandung

Design Pattern

Dosen Pengampu :
Zulkifli Arsyad
19.05.2025

Topik

- Introduction to OO concepts
- Introduction to Design Patterns
  - What are Design Patterns?
  - Why use Design Patterns?
  - Elements of a Design Pattern
  - Design Patterns Classification
  - Pros/Cons of Design Patterns
- Popular Design Patterns
- Conclusion
- References

What are Design Patterns?

- What Are Design Patterns?
- Wikipedia definition
  - “a design pattern is a general repeatable solution to a commonly occurring problem in software design”
- Quote from Christopher Alexander
  - “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice” (GoF,1995)

Oop Design Pattern The Gang of four

"Gang of Four" (GoF) merujuk pada empat penulis buku terkenal berjudul **"Design Patterns: Elements of Reusable Object-Oriented Software"** yang terbit tahun **1994**, yaitu:

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides
- Mereka memperkenalkan **23 pola desain** (design patterns) yang sangat berpengaruh dalam pengembangan perangkat lunak berbasis **Object-Oriented Programming (OOP)**.

It all started around 1994 when a group of 4 IBM programmers: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides based on their coding experience , were able to observe and documen a set of about 23 common problems and their best accepted solutions

< https://www.drupal.org/project/oop_design_patterns >

Why use Design Patterns

- Design Objectives
- Good Design (the “ilities”)
  - High readability and maintainability
  - High extensibility
  - High scalability
  - High testability
  - High reusability

High readability and maintainability

- Kemudahan dalam memperbaiki bug, melakukan pembaruan, dan meningkatkan sistem tanpa mengganggu fungsionalitas yang ada.

Ciri-ciri:	Manfaat
Penamaan variabel, fungsi, dan kelas jelas dan bermakna. Struktur kode konsisten dan rapi. Penggunaan komentar yang informatif jika diperlukan (bukan berlebihan). Tidak terlalu kompleks; mengikuti prinsip *KISS* (Keep It Simple).	Mempermudah kolaborasi tim. Mengurangi waktu debugging. Kode lebih cepat dipahami oleh developer baru.

- Kemampuan sistem untuk diperluas dengan fitur baru tanpa harus mengubah kode yang sudah ada secara signifikan.

Ciri-ciri:	Manfaat
Modul kode terpisah dengan tanggung jawab jelas (*Separation of Concerns*). Menghindari duplikasi kode (*Don't Repeat Yourself* – DRY). Kode terdokumentasi dengan baik.	Biaya pemeliharaan jangka panjang lebih rendah. Adaptif terhadap perubahan kebutuhan pengguna. Mudah ditelusuri saat terjadi error.

High extensibility

- Kemampuan sistem untuk diperluas dengan fitur baru tanpa harus mengubah kode yang sudah ada secara signifikan.

Ciri-ciri:	Manfaat
Menggunakan prinsip *Open/Closed* (terbuka untuk ekstensi, tertutup untuk modifikasi). Memanfaatkan antarmuka (*interface*) dan *abstraction*. Desain berbasis plugin atau modul (seperti di microservices).	Sistem tumbuh seiring kebutuhan bisnis. Mengurangi risiko kesalahan saat menambahkan fitur. Memungkinkan pengembangan paralel oleh banyak tim.

High scalability

- Kemampuan sistem untuk menangani peningkatan beban kerja (data, pengguna, proses) dengan penyesuaian minimum.

Ciri-ciri:	Manfaat
Arsitektur mendukung *horizontal scaling* dan *vertical scaling*. Penggunaan desain non-monolitik (misalnya microservices atau event-driven). Optimasi penggunaan memori dan proses latar belakang.	Sistem tetap cepat dan responsif saat beban meningkat. Mendukung pertumbuhan bisnis. Menghindari bottleneck dan kegagalan sistem.

High testability

- Kemudahan dalam menguji sistem, baik secara manual maupun otomatis.

Ciri-ciri:	Manfaat
Kode modular dan loosely coupled. Ketergantungan dapat dimock (Dependency Injection). Unit test, integration test, dan end-to-end test mudah dibuat.	Menjamin fungsionalitas sistem berjalan benar. Mempermudah proses Continuous Integration/Deployment (CI/CD). Mengurangi bug saat refactor atau update.

High reusability

- Kemampuan komponen kode digunakan kembali di bagian lain aplikasi atau di aplikasi berbeda.

Ciri-ciri:	Manfaat
Komponen bersifat generik dan independen. Desain mengikuti prinsip *Single Responsibility*. Modul mudah dipisahkan dan digunakan kembali.	Menghemat waktu pengembangan. Kualitas lebih baik karena komponen teruji. Mengurangi duplikasi logika bisnis.

Elements of a Design Pattern

- A pattern has four essential elements (GoF)
  - Name
    - Describes the pattern
    - Adds to common terminology for facilitating communication (i.e. not just sentence enhancers)
  - Problem
    - Describes when to apply the pattern
    - Answers - What is the pattern trying to solve?

Elements of a Design Pattern

- Solution
  - Describes elements, relationships, responsibilities, and collaborations which make up the design
  - Consequences
    - Results of applying the pattern
    - Benefits and Costs
    - Subjective depending on concrete scenarios

Kategori Pola Desain GoF

GoF membagi 23 pola desain menjadi **tiga kelompok utama**, berdasarkan tujuannya:

Kategori	Deskripsi	Contoh Pola
Creational	Fokus pada cara membuat objek	Singleton, Factory Method, Abstract Factory, Builder, Prototype
Structural	Fokus pada cara menyusun kelas dan objek	Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
Behavioral	Fokus pada cara objek berinteraksi dan berkomunikasi	Observer, Strategy, Command, Iterator, State, Template Method, Mediator, Chain of Responsibility, dll.

Creational Pattern

Pola	Tujuan
Singleton	Menjamin hanya ada satu instance dari suatu kelas.
Factory Method	Menentukan antarmuka untuk membuat objek, tapi membiarkan subclass menentukan kelas apa yang dibuat.
Abstract Factory	Menyediakan antarmuka untuk membuat keluarga objek terkait tanpa menentukan kelas konkretnya.
Builder	Memisahkan proses pembuatan objek kompleks dari representasinya.
Prototype	Membuat objek dengan menyalin instance yang sudah ada.

Structural Patterns

Pola	Tujuan
Adapter	Mengubah antarmuka suatu kelas agar bisa digunakan oleh sistem lain.
Bridge	Memisahkan abstraksi dari implementasinya agar bisa dikembangkan secara independen.
Composite	Menyusun objek menjadi struktur pohon (tree) untuk merepresentasikan hierarki bagian-keseluruhan.
Decorator	Menambahkan fungsionalitas ke objek secara dinamis tanpa mengubah strukturnya.
Facade	Menyediakan antarmuka sederhana untuk sistem yang kompleks.
Flyweight	Menghemat memori dengan berbagi objek yang sama untuk data yang berulang.
Proxy	Menyediakan objek pengganti untuk mengontrol akses ke objek lain.

Behavioral Patterns

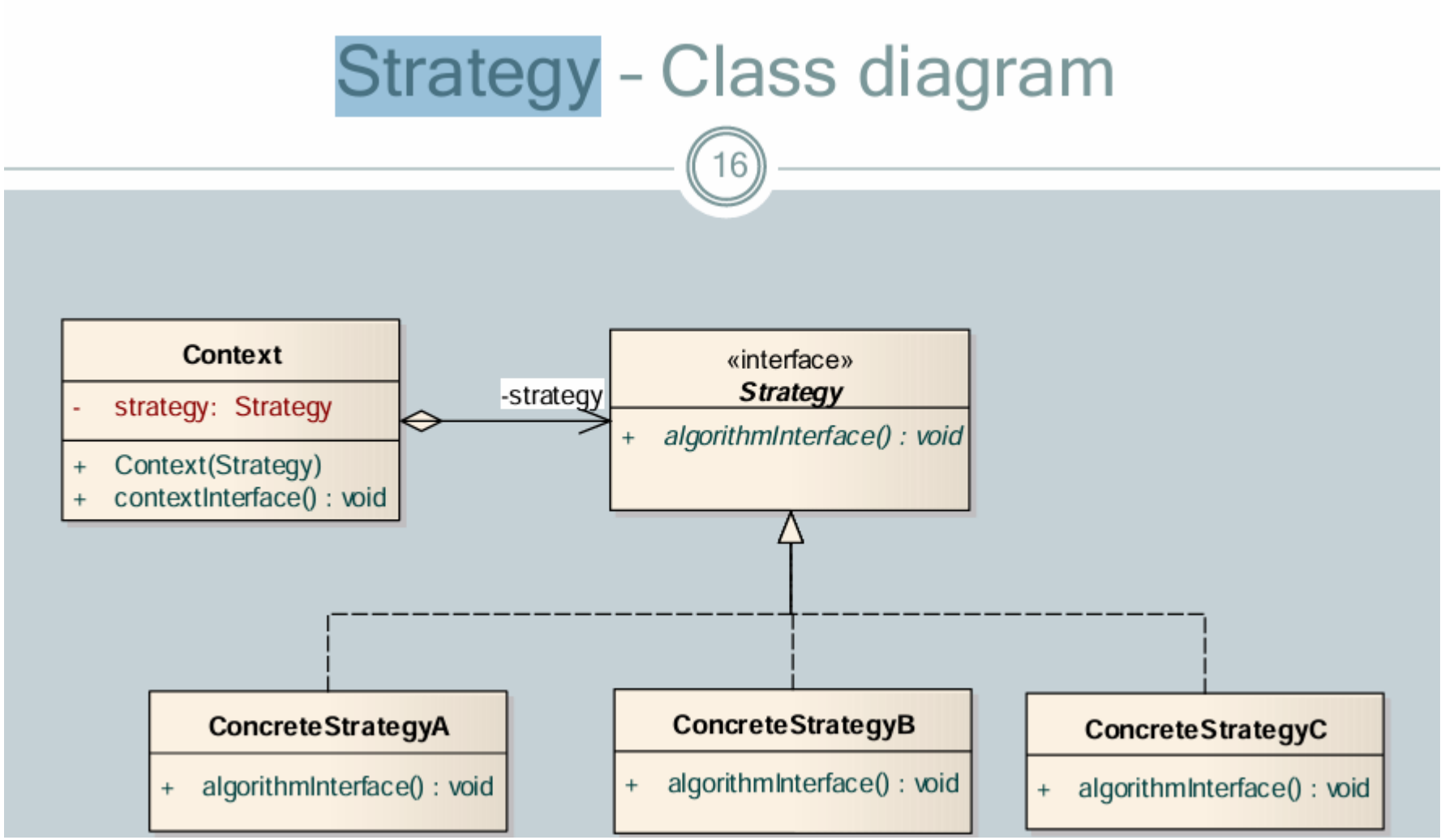
Pola	Tujuan
Observer	Secara otomatis memberi tahu semua objek terkait ketika ada perubahan status.
Strategy	Memilih algoritma dari kumpulan algoritma yang bisa diganti saat runtime.
Command	Mengubah permintaan menjadi objek, memungkinkan antrian atau log permintaan.
Iterator	Memberikan cara untuk mengakses elemen suatu koleksi tanpa mengekspos strukturnya.
State	Mengubah perilaku objek saat status internalnya berubah.
Template Method	Menentukan kerangka algoritma dan membiarkan subclass mengisi detailnya.

Behavioral Patterns

Pola	Tujuan
Mediator	Mengurangi ketergantungan antar objek dengan memperkenalkan objek mediator.
Chain of Responsibility	Memberikan sejumlah objek kesempatan untuk menangani permintaan.
Visitor	Memisahkan algoritma dari objek yang diprosesnya.
Interpreter	Menyediakan cara untuk mengevaluasi kalimat dalam bahasa tertentu.
Memento	Menyimpan dan mengembalikan status objek tanpa melanggar enkapsulasi.

Behavioral Patterns – Strategy

- Strategy Pattern



Kasus

Kasus Riil: *Pengiriman Barang di Aplikasi Ekspedisi*

Sebuah aplikasi ekspedisi mendukung 3 jenis strategi pengiriman:

- **Strategi A**: Pengiriman **Reguler**
- **Strategi B**: Pengiriman **Ekspres**
- **Strategi C**: Pengiriman **Same Day**
- Pengguna dapat memilih metode pengiriman dan sistem akan menghitung biaya berdasarkan strategi tersebut.

1. Strategy Interface: ShippingStrategy

```
public interface ShippingStrategy {
    double calculateShippingCost(double weightInKg);
}
```

2. Concrete Strategies A, B, C

```
public class RegularShipping implements ShippingStrategy {
    @Override
    public double calculateShippingCost(double weightInKg) {
        return 10000 + (weightInKg * 2000);
    }
}

public class ExpressShipping implements ShippingStrategy {
    @Override
    public double calculateShippingCost(double weightInKg) {
        return 20000 + (weightInKg * 3000);
    }
}

public class SameDayShipping implements ShippingStrategy {
    @Override
    public double calculateShippingCost(double weightInKg) {
        return 30000 + (weightInKg * 5000);
    }
```

3. Context Class ShippingContext

```
public class ShippingContext {
    private ShippingStrategy strategy;

    public ShippingContext(ShippingStrategy strategy) {
        this.strategy = strategy;
    }

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

    public double calculateCost(double weightInKg) {
        return strategy.calculateShippingCost(weightInKg);
    }
}
```

4. Main Class

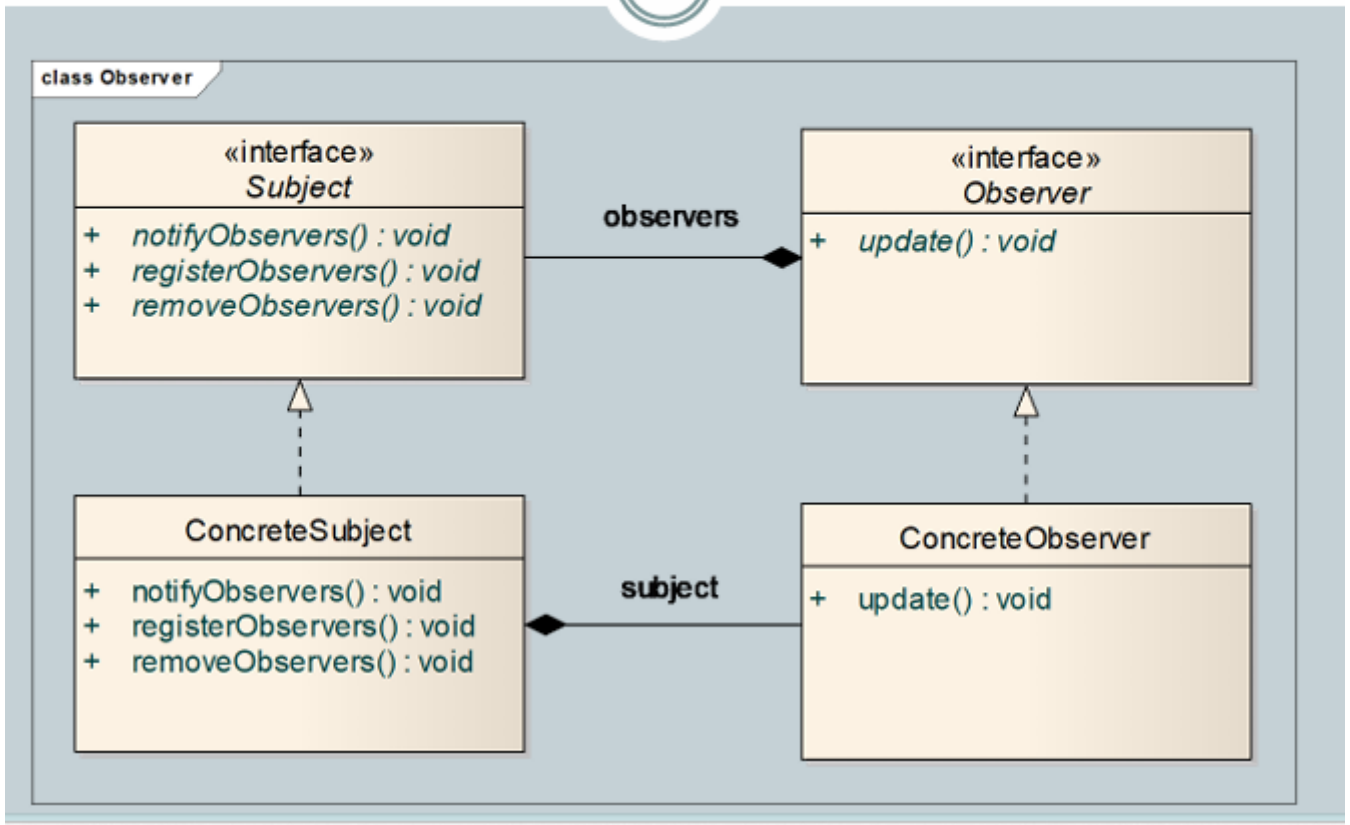
```
public class Main {
    public static void main(String[] args) {
        double packageWeight = 3.5; // kg

        ShippingContext context = new ShippingContext(new RegularShipping());
        System.out.println("Biaya pengiriman Reguler: Rp" + context.calculateCost(packageWeight));

        context.setStrategy(new ExpressShipping());
        System.out.println("Biaya pengiriman Ekspres: Rp" + context.calculateCost(packageWeight));

        context.setStrategy(new SameDayShipping());
        System.out.println("Biaya pengiriman Same Day: Rp" + context.calculateCost(packageWeight));
    }
}
```

Observer - Class diagram



- Kasus Riil: Notifikasi Cuaca
- Sebuah aplikasi cuaca memiliki:
- Subject: WeatherStation — menyediakan data cuaca.
- Observers:
  - Aplikasi pengguna (MobileApp)
  - Layar luar ruangan (OutdoorDisplay)
  - Sistem peringatan dini (WeatherAlertSystem)
  - Ketika suhu berubah, semua observer langsung mendapat notifikasi.

1. Interface Observer

```
public interface Observer {
    void update(float temperature);
}
```

2. Interface Subject

```
public interface Subject {
    void addObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}
```

3.ConcreteSubject WeatherStation

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

public class WeatherStation implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private float temperature;

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

    @Override
    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(temperature);
        }
    }

    public void setTemperature(float newTemperature) {
        System.out.println("WeatherStation: Suhu baru adalah " + newTemperature + "°C");
        this.temperature = newTemperature;
        notifyObservers();
    }
}
```

4.ConcreteObserver

```java
public class MobileApp implements Observer {
    @Override
    public void update(float temperature) {
        System.out.println("MobileApp: Menampilkan suhu di aplikasi: " + temperature + "°C");
    }
}

public class OutdoorDisplay implements Observer {
    @Override
    public void update(float temperature) {
        System.out.println("OutdoorDisplay: Menampilkan suhu luar: " + temperature + "°C");
    }
}
```

```java
public class WeatherAlertSystem implements Observer {
    @Override
    public void update(float temperature) {
        if (temperature > 35) {
            System.out.println("WeatherAlertSystem: ⚠️ Peringatan cuaca panas ekstrem!");
        }
    }
}
```

5. Main

```java
public class Main {
    public static void main(String[] args) {
        WeatherStation station = new WeatherStation();

        Observer app = new MobileApp();
        Observer display = new OutdoorDisplay();
        Observer alertSystem = new WeatherAlertSystem();

        station.addObserver(app);
        station.addObserver(display);
        station.addObserver(alertSystem);

        station.setTemperature(28.5f);
        System.out.println("----");
        station.setTemperature(36.0f);
    }
}
```

```
WeatherStation: Suhu baru adalah 28.5°C
MobileApp: Menampilkan suhu di aplikasi: 28.5°C
OutdoorDisplay: Menampilkan suhu luar: 28.5°C
----
WeatherStation: Suhu baru adalah 36.0°C
MobileApp: Menampilkan suhu di aplikasi: 36.0°C
OutdoorDisplay: Menampilkan suhu luar: 36.0°C
WeatherAlertSystem: ⚠️ Peringatan cuaca panas ekstrem!
```

Praktikum

Behavioural Patterns	Creational Patterns	Structural Patterns
Chain Of Responsibility	Factory Method	Adapter
Command	Abstract Factory	Bridge
Iterator	Builder	Composite
Mediator	Prototype	Decorator
Memento	Singleton	Facade
Observer		Flyweight
State		Proxy
Strategy		
Template		
Visitor		

Praktikum

- Masing-masing Pilih 2 design pattern pada kategori behavioral, cretional dan structural.
- Buatkan source code yang sederhana dengan kasus real yang menerapkan design pattern tersebut
- Buatkan laporan PDF mencakup
  - Name
  - Problem yg menjadi dasar dipilih DP tersebut apa
  - Kapan digunakan
  - Solusi
  - konsekuensi
  - Diagram class