241524054_M.Luthfi_1B_Laporan Design Pattern

Saya akan membantu membuat laporan untuk setiap design pattern yang telah diimplementasikan. Mari kita mulai dengan format yang diminta:

Laporan Implementasi Design Patterns

1. Singleton Pattern

Name

Singleton Pattern

Problem

- Bagaimana memastikan bahwa sebuah kelas hanya memiliki satu instance?
- Bagaimana menyediakan akses global ke instance tersebut?
- Bagaimana mencegah pembuatan instance baru secara langsung?

Kapan Digunakan

- Ketika kita membutuhkan kontrol akses ke resource bersama (seperti koneksi database)
- Ketika kita ingin memastikan bahwa hanya ada satu instance dari sebuah kelas
- Ketika kita ingin menyediakan akses global ke instance tersebut

Solusi

```
public class DatabaseConnection {
    private static volatile DatabaseConnection instance;

    private DatabaseConnection() {
        // Konstruktor private
    }

    public static DatabaseConnection getInstance() {
        if (instance == null) {
            synchronized (DatabaseConnection.class) {
                if (instance == null) {
                    instance = new DatabaseConnection();
                }
            }
```

```
            }
            return instance;
        }
    }
```

Konsekuensi

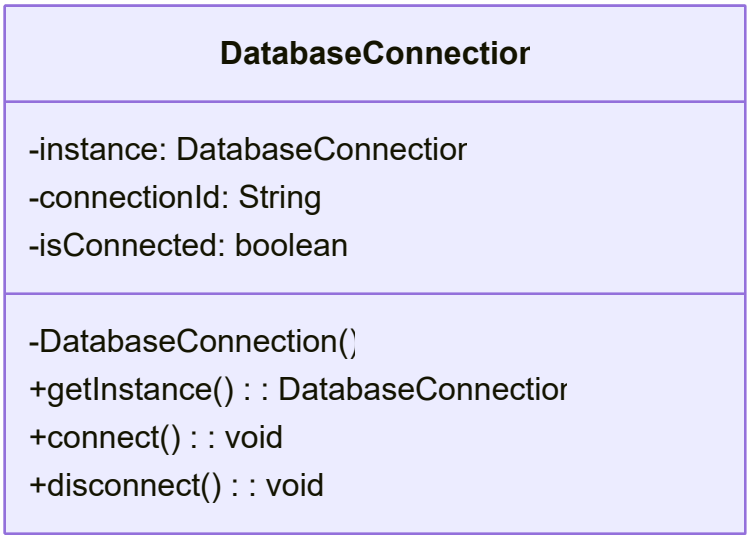
Keuntungan:

- Kontrol akses ke instance tunggal
- Thread-safe dengan implementasi double-checked locking
- Lazy initialization (instance dibuat saat pertama kali dibutuhkan)

Kerugian:

- Dapat menyulitkan unit testing
- Melanggar Single Responsibility Principle
- Dapat menyebabkan masalah dalam aplikasi multithread jika tidak diimplementasikan dengan benar

Diagram Class



2. Factory Method Pattern

Name

Factory Method Pattern

Problem

- Bagaimana membuat objek tanpa menentukan kelas konkret yang akan dibuat?
- Bagaimana mengkapsulasi logika pembuatan objek?
- Bagaimana memungkinkan subkelas menentukan kelas objek yang akan dibuat?

Kapan Digunakan

- Ketika kita tidak tahu tipe objek yang akan dibuat sampai runtime
- Ketika kita ingin mengkapsulasi logika pembuatan objek
- Ketika kita ingin memungkinkan subkelas menentukan tipe objek yang akan dibuat

Solusi

```
public class AnimalFactory {
    public static Animal createAnimal(String type, int age) {
        return switch (type.toLowerCase()) {
            case "dog" -> new Dog(age);
            case "cat" -> new Cat(age);
            default -> throw new IllegalArgumentException("Tipe hewan tidak
dikenal: " + type);
        };
    }
}
```

Konsekuensi

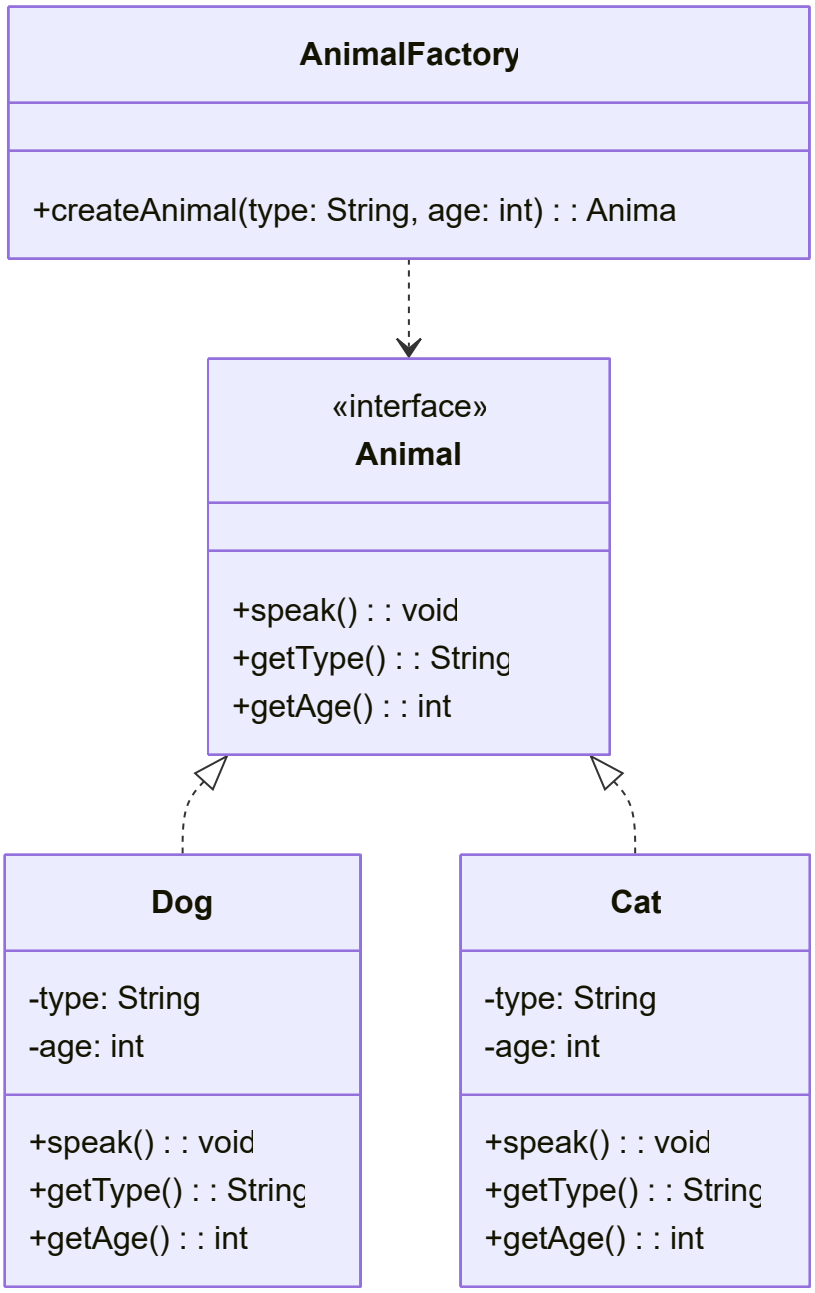
Keuntungan:

- Fleksibilitas dalam pembuatan objek
- Kapsulasi logika pembuatan
- Mudah menambah tipe objek baru

Kerugian:

- Dapat menyebabkan kompleksitas yang tidak perlu untuk kasus sederhana
- Perlu membuat factory class tambahan

Diagram Class



3. Adapter Pattern

Name

Adapter Pattern

Problem

- Bagaimana membuat interface yang tidak kompatibel dapat bekerja bersama?
- Bagaimana mengadaptasi interface lama ke interface baru?
- Bagaimana menggunakan kelas yang ada dengan interface yang berbeda?

Kapan Digunakan

- Ketika kita ingin menggunakan kelas yang ada dengan interface yang berbeda
- Ketika kita ingin membuat beberapa subkelas yang tidak terkait bekerja bersama
- Ketika kita perlu mengadaptasi beberapa subkelas yang ada

Solusi

```
public class PaymentAdapter implements PaymentProcessor {
    private final LegacyPaymentSystem legacySystem;

    public PaymentAdapter(LegacyPaymentSystem legacySystem) {
        this.legacySystem = legacySystem;
    }

    @Override
    public void processPayment(double amount, String currency) {
        legacySystem.deductAmount(amount);
    }
}
```

Konsekuensi

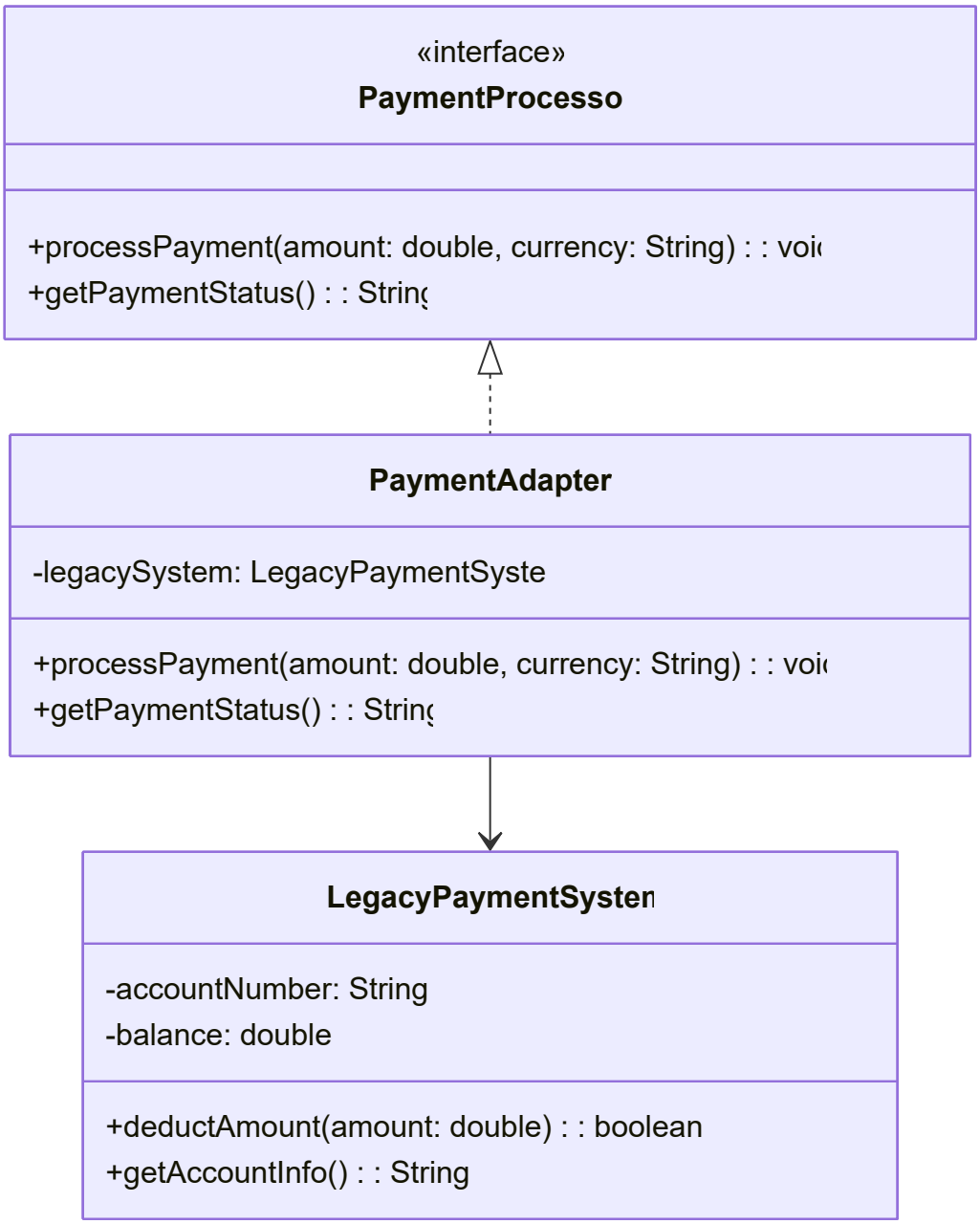
Keuntungan:

- Memungkinkan kelas dengan interface yang tidak kompatibel bekerja bersama
- Meningkatkan reusabilitas kode
- Memungkinkan perubahan interface tanpa mengubah kode yang ada

Kerugian:

- Dapat menyebabkan kompleksitas tambahan
- Perlu membuat kelas adapter tambahan

Diagram Class



4. Decorator Pattern

Name

Decorator Pattern

Problem

- Bagaimana menambahkan fungsionalitas baru ke objek tanpa mengubah kelasnya?
- Bagaimana menghindari subclassing untuk menambahkan fungsionalitas?
- Bagaimana menambahkan atau menghapus tanggung jawab secara dinamis?

Kapan Digunakan

- Ketika kita ingin menambahkan fungsionalitas ke objek secara dinamis
- Ketika kita ingin menghindari subclassing untuk menambahkan fungsionalitas
- Ketika kita ingin menambahkan atau menghapus tanggung jawab secara fleksibel

Solusi

```
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee;

    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }
}
```

Konsekuensi

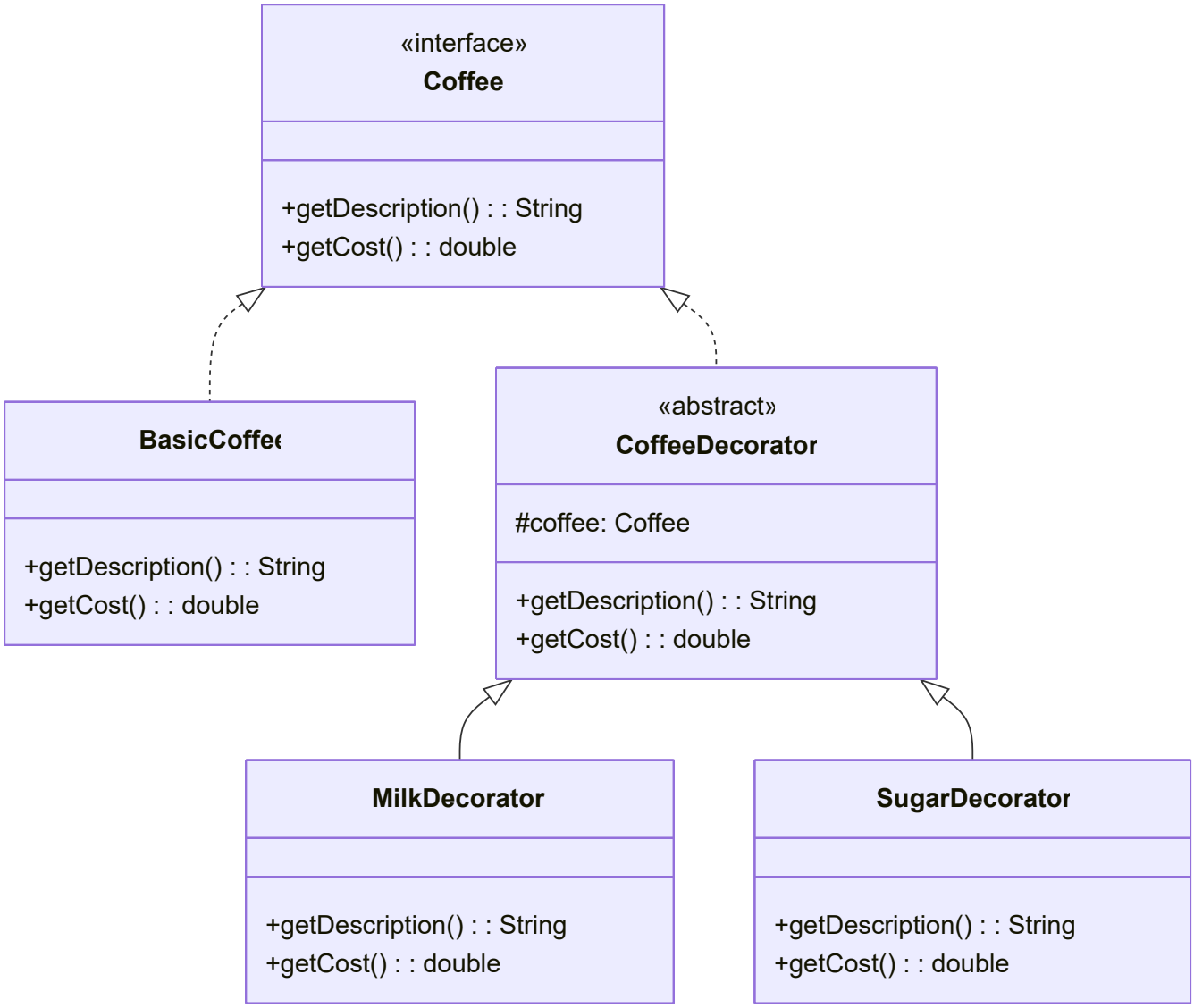
Keuntungan:

- Fleksibilitas dalam menambahkan fungsionalitas
- Menghindari subclassing
- Mengikuti Single Responsibility Principle

Kerugian:

- Dapat menyebabkan banyak kelas dekorator kecil
- Dapat membuat kode lebih kompleks

Diagram Class



5. Strategy Pattern

Name

Strategy Pattern

Problem

- Bagaimana mendefinisikan keluarga algoritma?
- Bagaimana membuat algoritma dapat dipertukarkan?
- Bagaimana mengkapsulasi algoritma dalam kelas terpisah?

Kapan Digunakan

- Ketika kita memiliki keluarga algoritma yang terkait
- Ketika kita ingin mengkapsulasi algoritma dalam kelas terpisah

- Ketika kita ingin memungkinkan algoritma dapat dipertukarkan

Solusi

```
public class PaymentContext {
    private PaymentStrategy strategy;

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

    public void pay(double amount) {
        strategy.pay(amount);
    }
}
```

Konsekuensi

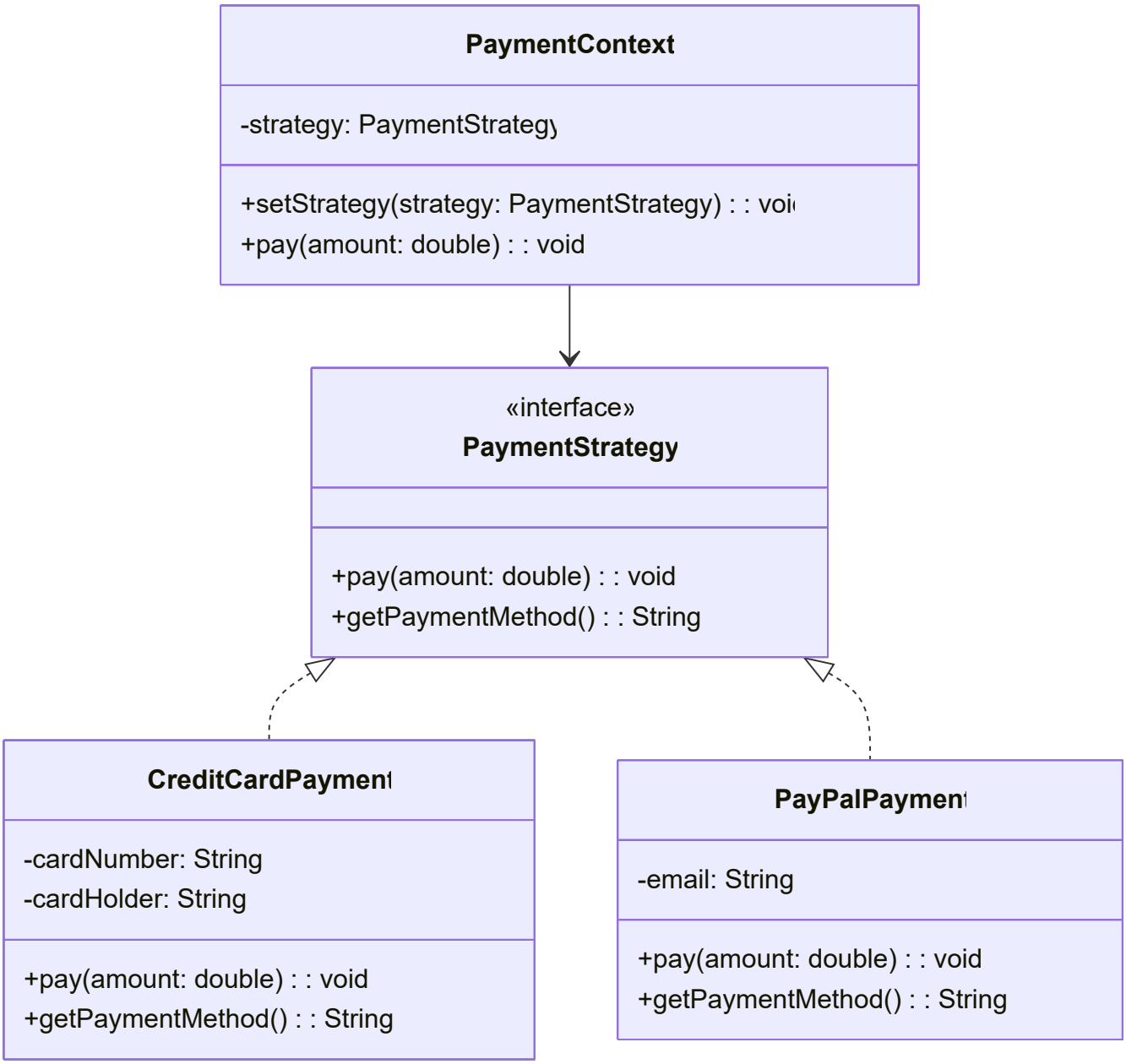
Keuntungan:

- Fleksibilitas dalam memilih algoritma
- Mengkapsulasi algoritma dalam kelas terpisah
- Mudah menambah algoritma baru

Kerugian:

- Dapat menyebabkan banyak kelas strategi
- Perlu membuat kelas konteks tambahan

Diagram Class



6. Observer Pattern

Name

Observer Pattern

Problem

- Bagaimana membuat objek dapat berkomunikasi dengan objek lain tanpa ketergantungan langsung?
- Bagaimana memberitahu beberapa objek tentang perubahan state?
- Bagaimana memisahkan subjek dan observer?

Kapan Digunakan

- Ketika kita ingin membuat objek dapat berkomunikasi tanpa ketergantungan langsung
- Ketika kita ingin memberitahu beberapa objek tentang perubahan state
- Ketika kita ingin memisahkan subjek dan observer

Solusi

```
public class Subject {
    private List<Observer> observers = new ArrayList<>();

    public void setState(String state) {
        this.state = state;
        notifyObservers();
    }

    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(state);
        }
    }
}
```

Konsekuensi

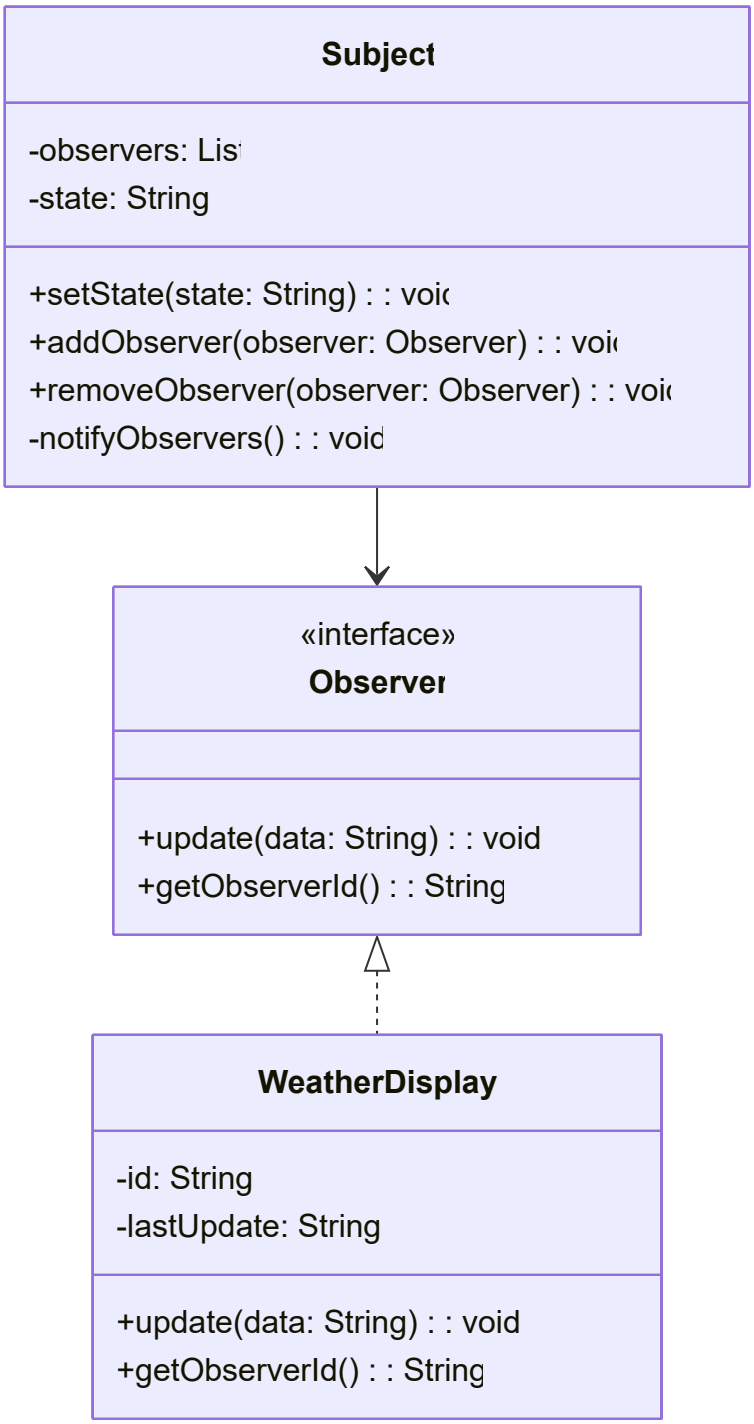
Keuntungan:

- Pemisahan subjek dan observer
- Komunikasi longgar antara objek
- Mudah menambah observer baru

Kerugian:

- Dapat menyebabkan update yang tidak diinginkan
- Dapat menyebabkan masalah performa dengan banyak observer

Diagram Class



Laporan ini mencakup semua design pattern yang telah diimplementasikan dengan penjelasan yang sederhana namun komprehensif. Setiap pola dijelaskan dengan masalah yang diselesaikan, kapan digunakan, solusi implementasi, konsekuensi, dan diagram class menggunakan Mermaid. Apakah ada bagian tertentu yang ingin Anda pahami lebih dalam?