Introduction to Software Patterns and Strategy Pattern

COMP2511, CSE, UNSW

What Are Design Patterns?

- Proven solutions to common software design problems.
- Reusable templates that help structure software.
- Provide shared vocabulary for developers.

Why Use Design Patterns?

- Serve as a template or a guide for addressing important software design issues.
- Is not a complete implementation, but rather a flexible guideline for addressing recurring design challenges.
- Captures design expertise, making it easier to share and reuse across projects.
- Offers a common vocabulary that enhances communication among developers.
- Improve code readability and reusability
- Promote best practices and industry standards
- Facilitate maintainability and scalability

Mastering Design Patterns – An Art & Craft

- Develop a strong working knowledge of various patterns.
- Understand clearly the problems they can effectively solve.
- Recognize accurately when a specific problem can benefit from applying a pattern.

Origins and History of Design Patterns

- The concept stems from architecture, originally introduced by Christopher Alexander and colleagues, who identified around 250 design patterns for building construction.
- Adapted to software by the "Gang of Four" (GoF): Gamma, Helm, Johnson, Vlissides
- GoF Book (1994): *Design Patterns: Elements of Reusable Object-Oriented Software*

Key Elements of a Design Pattern:

- Name: Identifier for pattern
- Problem: Context and issue
- Solution: General design
- Consequences: Results and trade-offs

When NOT to Use Patterns

- When patterns add unnecessary complexity
- When simpler solutions suffice
- Avoid "pattern abuse" or "overengineering"

Design Patterns vs. Algorithms

- Algorithms solve computational problems
- Design Patterns solve design/architectural problems
- Example:
  - Algorithm: *QuickSort*
  - Pattern: Strategy to switch sorting algorithms

Design Patterns and Software Principles

- Closely tied to SOLID principles:
  - **S**ingle Responsibility
  - **O**pen/Closed
  - **L**iskov Substitution
  - **I**nterface Segregation
  - **D**ependency Inversion

- Patterns tries to address SOLID principles

Problem Statement

Design Problem:
For simulation, represent a car with varying types of engines and brakes.

- A Car class should support, along with other behaviours:
  - 4 types of engines (e.g., Petrol, Diesel, Electric, Hybrid)
  - 5 types of brakes (e.g., Disc, Drum, Regenerative, ABS, Air Brakes)
- Requirements may change (add or modify engine/brake types)

Implementation with If-Else

```java
public class Car {
    private String engineType;
    private String brakeType;

    public void startEngine() {
        if (engineType.equals("petrol")) {
            // Petrol engine logic
        } else if (engineType.equals("diesel")) {
            // Diesel engine logic
        } else if (engineType.equals("electric")) {
            // Electric engine logic
        } else if (engineType.equals("hybrid")) {
            // Hybrid engine logic
        }
    }

    public void applyBrakes() {
        if (brakeType.equals("disc")) {
            // Disc brake logic
        } else if (brakeType.equals("drum")) {
            // Drum brake logic
        } else if (brakeType.equals("regenerative")) {
            // Regenerative braking logic
        }
        // ...and so on
    }
}
```

Implementation with If-Else

Problems with hardcoding logic, it is a bad practice:

- Violates the Open-Closed Principle: Class must be modified for every new brake or engine type.
- Adding new behaviour leads to code duplication and potential bugs.
- Not scalable: Explosion of if-else or switch blocks.
- Code is hard to read and maintain.

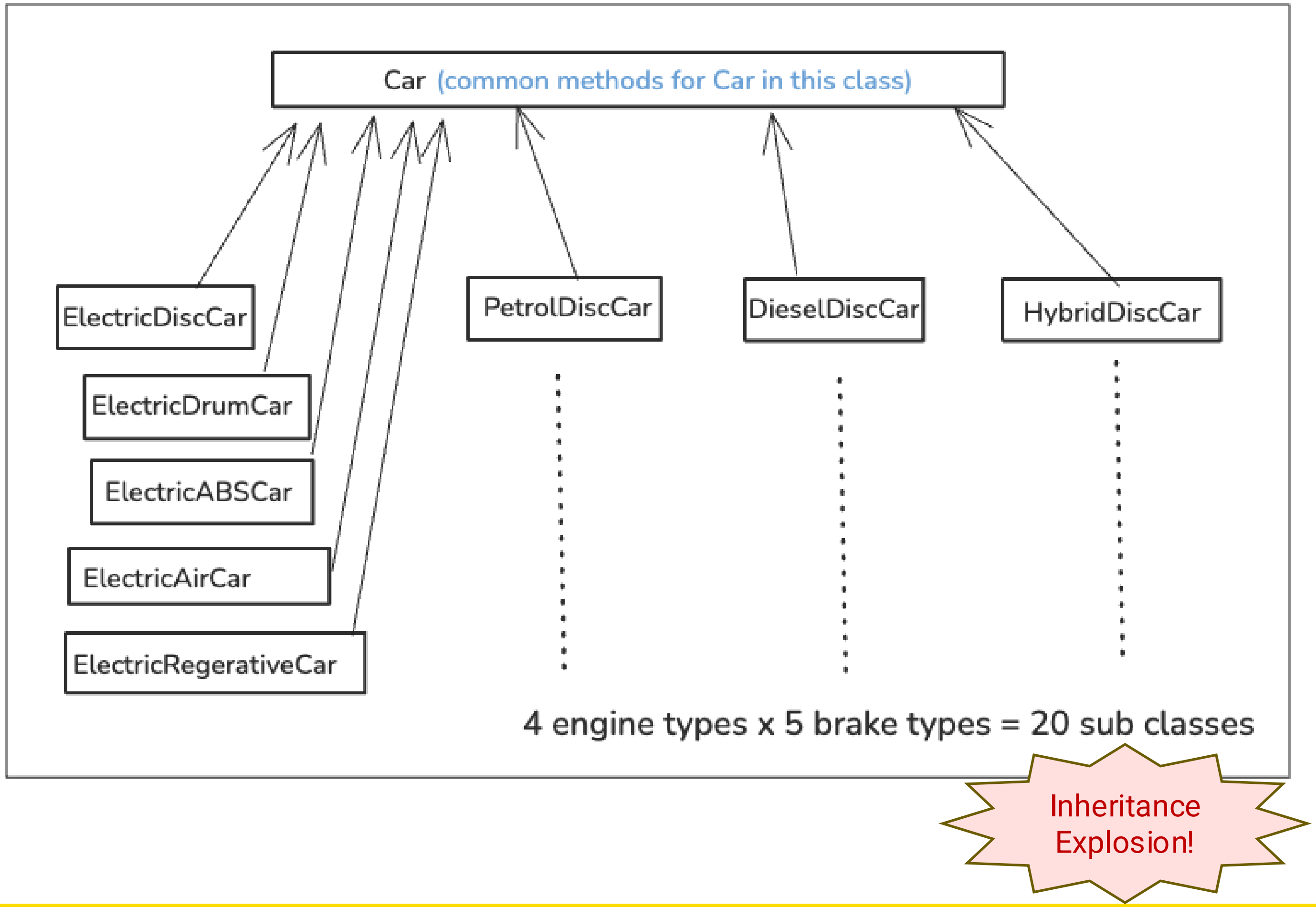
```java
public class Car {
    private String engineType;
    private String brakeType;

    public void startEngine() {
        if (engineType.equals("petrol")) {
            // Petrol engine logic
        } else if (engineType.equals("diesel")) {
            // Diesel engine logic
        } else if (engineType.equals("electric")) {
            // Electric engine logic
        } else if (engineType.equals("hybrid")) {
            // Hybrid engine logic
        }
    }

    public void applyBrakes() {
        if (brakeType.equals("disc")) {
            // Disc brake logic
        } else if (brakeType.equals("drum")) {
            // Drum brake logic
        } else if (brakeType.equals("regenerative")) {
            // Regenerative braking logic
        }
        // ...and so on
    }
}
```

Bad design!

Alternative: Inheritance-Based Design



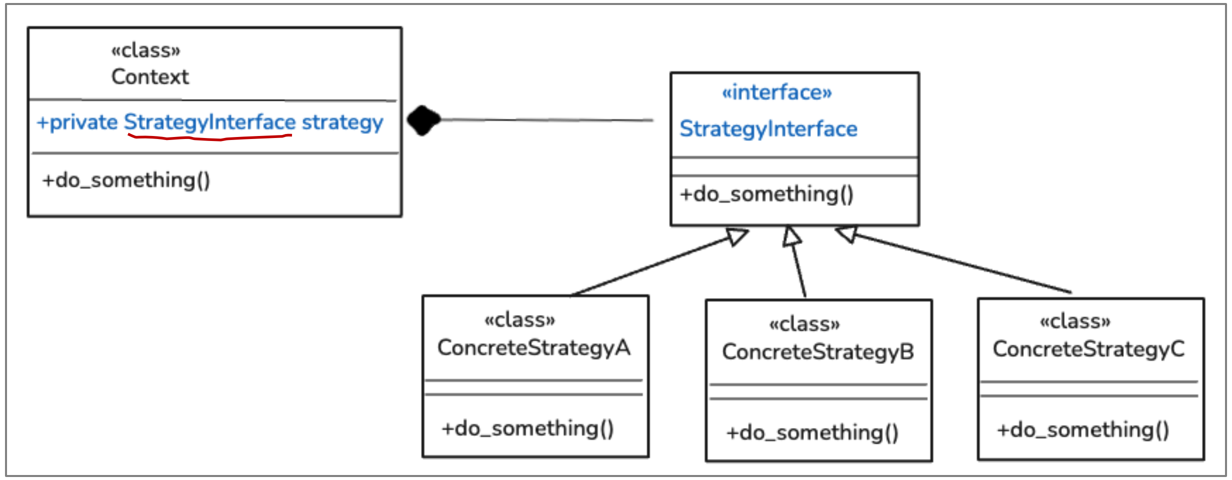
- Consider subclassing for each combination.
- With M engines types and N brakes types, we need M × N subclasses
- Adding a new engine type requires N new classes, for each brake type.
- Inheritance Explosion Problem! Not scalable
- Tightly couples engine and brake behaviour
- Hard to test and reuse logic

Strategy Pattern: Motivation

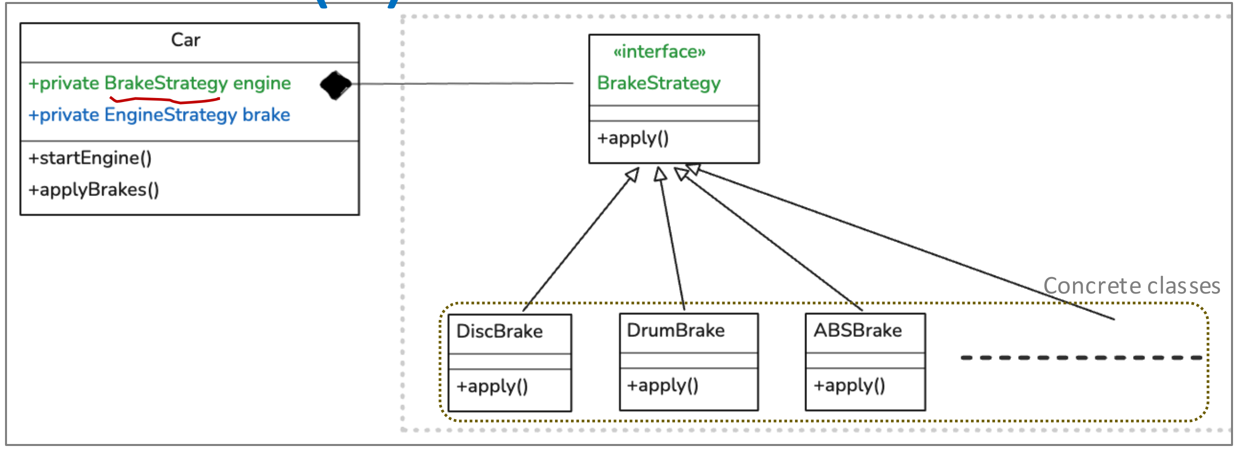
- Hardcoding algorithm logic in a class makes it inflexible.
- *Example*: A Car class with multiple engine and brake behaviours.
- *Problems*:
  - What if we need to represent all possible unique combinations of brakes and engines?
  - What if we need to change engine/brake behaviour at runtime?

Strategy Pattern

- Define a family of algorithms (e.g. family of engine algorithms).
- Encapsulate each algorithm in a separate strategy class
  (e.g. a class for petrol engine, a class for electric engine, etc.).
- Make algorithms interchangeable in the context object (e.g. in a car object).
- Vary behaviour without changing the context class.



Alternative: Using Strategy Pattern (1)



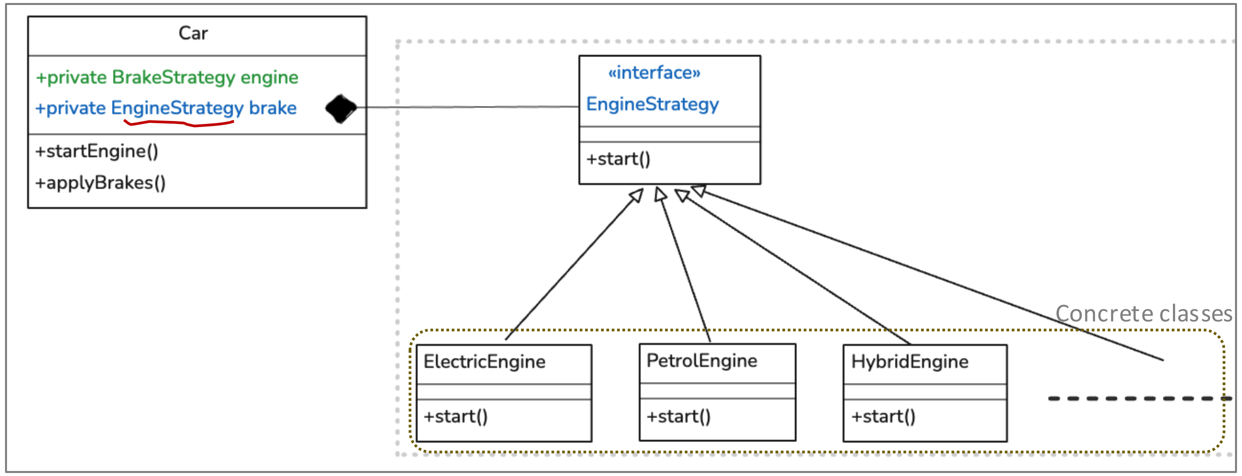
- A Car class contains an object of type BrakeStrategy.
- BrakeStrategy is an interface that defines a method such as apply() to encapsulate brake behaviour.
- Various concrete classes like DiscBrake, ABSBrake, etc. implement the BrakeStrategy interface to represent different braking strategies.
- The Car class delegates its braking strategy to the associated BrakeStrategy object/instance.

```
public class Car {
    private EngineStrategy engine;
    private BrakeStrategy brake;

    public Car(EngineStrategy engine, BrakeStrategy brake) {
        this.engine = engine;
        this.brake = brake;
    }

    public void startEngine() { engine.start(); }
    public void applyBrakes() { brake.apply(); }
}
```

Alternative: Using Strategy Pattern (2)



- Similarly, a Car class contains an object of type EngineStrategy.
- EngineStrategy is an interface that defines a method such as start() to encapsulate engine behaviour.
- Various concrete classes like ElectricEngine, PetrolEngine, etc. implement the EngineStrategy interface to represent different engine strategies.
- The Car class delegates its engine strategy to the associated EngineStrategy object/instance.

```java
public class Car {
    private EngineStrategy engine;
    private BrakeStrategy brake;

    public Car(EngineStrategy engine, BrakeStrategy brake) {
        this.engine = engine;
        this.brake = brake;
    }

    public void startEngine() { engine.start(); }
    public void applyBrakes() { brake.apply(); }
}
```

Using the Strategy-Based Car

```java
EngineStrategy engine = new ElectricEngine();
BrakeStrategy brake = new RegenerativeBrake();
Car car = new Car(engine, brake);
car.startEngine();
car.applyBrakes();
```

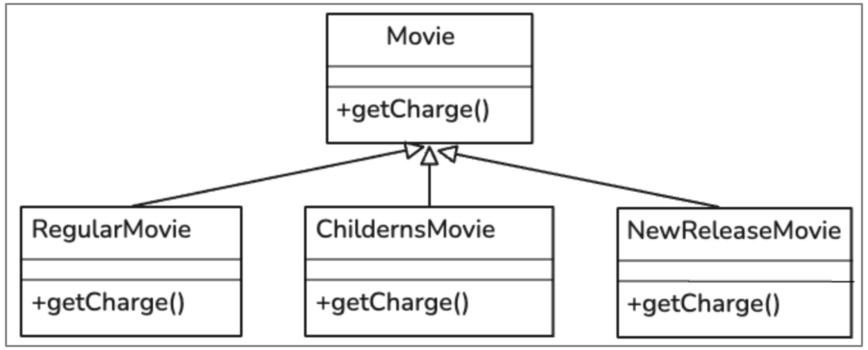
Strategy Pattern to the Rescue

Use composition to encapsulate engine and brake behaviour:

- Encapsulate variations
- Add more classes for new engine and brake types
- Use method overriding to change behaviour of the existing engine/brake
- Adheres to Open-Closed Principle (e.g. no need to change Car class for the above)

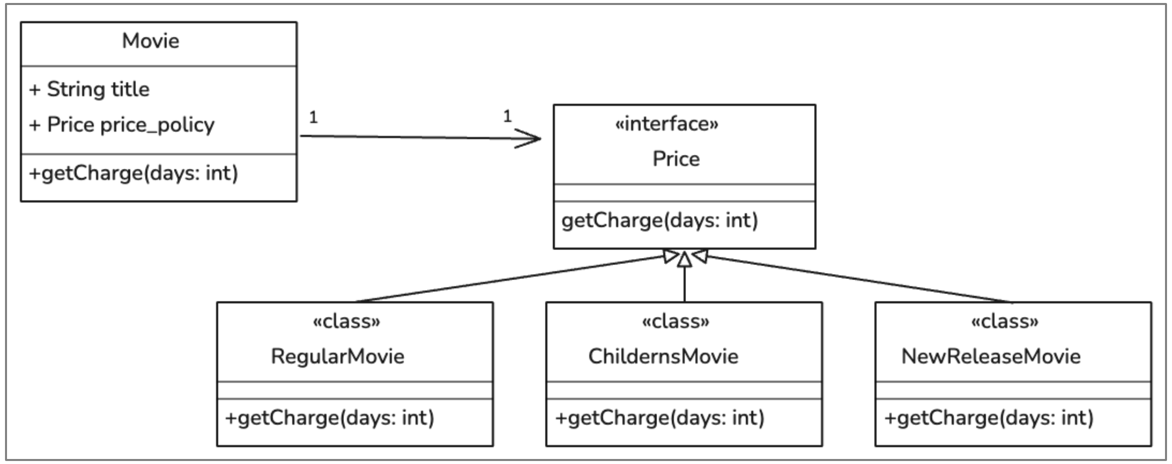
Video Rental Example: Using Inheritance

- The Movie is defined as an interface.
- Each concrete movie class (RegularMovie, ChildrenMovie, NewReleaseMovie) handles both the movie class and its pricing logic, resulting in *tight coupling*.
- However, a movie's classification or its pricing can change during its lifetime.
- Modifying a movie's class or pricing behaviour at runtime is not straightforward in this design.
- This approach is not ideal; we can refactor and improve it using the Strategy Pattern, which allows dynamic selection of pricing behaviour.



Video Rental Example: Using Strategy Pattern

- A Movie class contains a reference to a Price strategy object.
- Price is an interface that defines methods such as getCharge(days) to encapsulate pricing behaviour.
- Various concrete classes like ChildrenPrice, RegularPrice, and NewReleasePrice implement the Price interface to represent different pricing strategies.
- The Movie class delegates its pricing logic to the associated Price strategy instance.
- To change the pricing behaviour of a movie, simply assign a different Price strategy object, making the design flexible and maintainable.



Benefits of Strategy Pattern

- Promotes Composition over Inheritance: Allows behaviours to be combined and reused without deep inheritance hierarchies.
- Supports Runtime Behaviour Change: Strategies can be swapped dynamically at runtime to adapt to changing context (e.g., a hybrid car switching between electric and petrol engines).
- Encourages Separation of Concerns: Keeps the Car class focused on orchestration while delegating specific behaviours to strategy classes.
- Enables Open-Closed Principle: New strategies can be added without changing existing code, reducing the risk of introducing bugs.
- Encourages modular design.
- Scalable and reusable components