

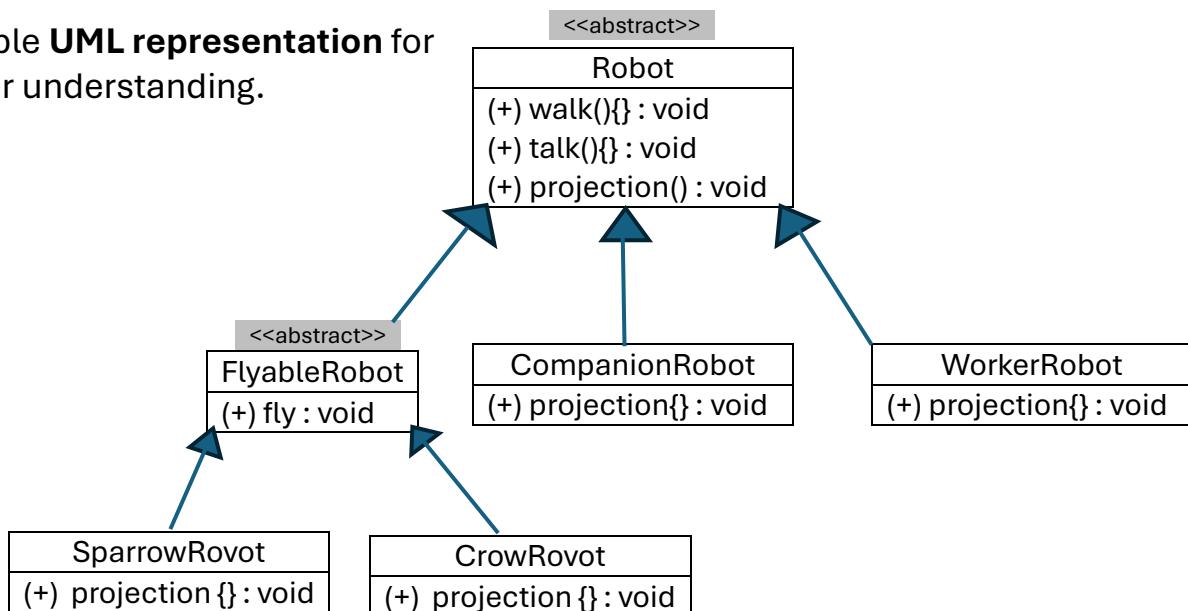
# Strategy Design Pattern

**Design patterns** are formalized best practices that programmers can use to **solve common problems** when **designing** an **application** or **system**. They are **not** directly **executable code** but rather **descriptions** of **how to solve a problem**, which can be implemented in various ways depending on the specific context.

Let's begin with an **example** where we are developing an **application** to simulate **various types of robots**.

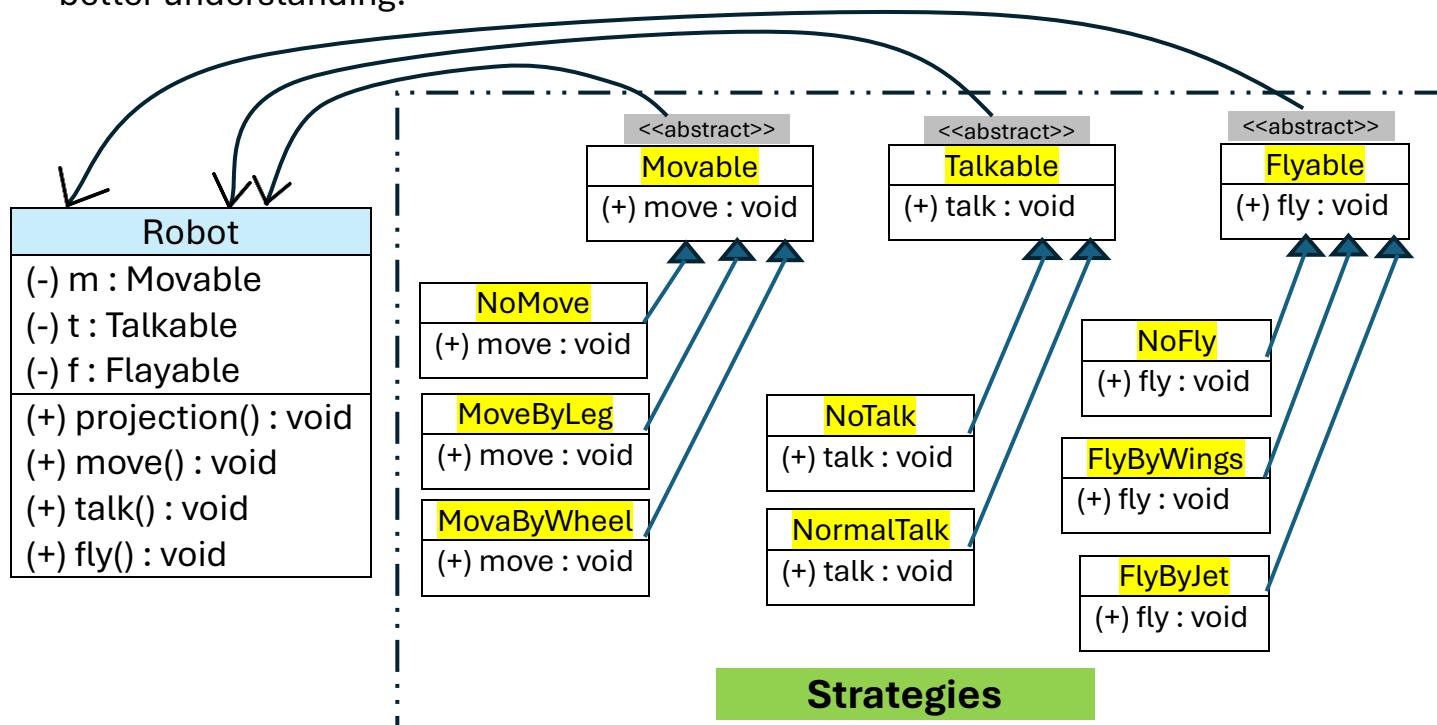
- We start by designing an **abstract class** named **Robot**, which **implements** the **common functionalities** such as **walk()** and **talk()**, and **declares** an **abstract method** **projection()**. Each specific robot type will provide its **own** implementation of the **projection method**.
- Initially, we have two types of robots: **CompanionRobot** and **WorkerRobot**. Both robots **share** the ability to **talk** and **walk**, so the current design works well.
- However, imagine a new robot called **SparrowRobot** is introduced, which has the ability to **fly**. Following that, more **flyable robots** like **CrowRobot** are developed. At this point, we face a **design decision**:
  - Should we **add** a **fly()** method to the **base Robot class**?
  - Or should we **create a new interface** like **FlyableRobot** that **extends Robot**?
- Now, suppose another robot is invented that **flies** using **jets** instead of **wings**. This means we now have **two categories** of **flying behavior**: **fly by wings** and **fly by jet**. To support this, we would either need to **rewrite** significant portions of the code or restructure the class hierarchy altogether.
- A similar **issue** arises with **walking behavior**. Some robots may **walk using legs**, while others might **move using wheels**. If we use **interfaces** alone to represent these behaviors, we risk **violating** the **Open/Closed Principle (OCP)** — our classes are **not closed** for **modification** and **require frequent changes** as **new behaviors emerge**.
- This scenario clearly illustrates the need for a **more flexible** and **extensible design** — one that can **handle evolving behaviors** without **constantly altering existing code**.

- Sample **UML representation** for better understanding.



## Problems with inheritance:

- Code reusability **X**.
  - To add new feature **a lot of changes were required** **✓**.
  - **open-close principle (OCP)** **X**.
- To address these issues, I will use the **Strategy Design Pattern**, which states:  
**“Define a family of algorithms (or strategies), encapsulate each one, and make them interchangeable so that they can vary independently at runtime.”**
  - I will create three separate interfaces — **Movable**, **Talkable**, and **Flyable** — and use them as **composed objects** within the Robot class.
  - By doing so, we can **dynamically assign** or **change** behaviours related to movement, communication, and flying without modifying the existing robot code.
  - Sample **UML representation** for better understanding.



- By **using constructor injection**, we can now dynamically assign any valid combination of behaviours at runtime without modifying existing code.
- This approach fully follows the **Open-Closed Principle (OCP)** — the system is **open for extension but closed for modification**.

### 💡 Benefits

- **code reusability** and **flexibility**. **✓**
- **Open/Closed Principle (OCP)**. **✓**
- **Bulky inheritance tree logic**. **X**
- Makes behaviour **interchangeable** at **runtime**. **✓**

## Robot

```
public abstract class Robot {  
    private final Movable m;  
    private final Talkable t;  
    private final Flyable f;  
    //constructor injection  
    public Robot (Movable m, Talkable t, Flyable f) {  
        this.m = m;  
        this.t = t;  
        this.f = f;  
    }  
    public void move() { m.move(); }  
    public void talk() { t.talk(); }  
    public void fly() { f.fly(); }  
    public abstract void projection();  
}
```

## Worker Robot

```
public class WorkerRobot extends Robot {  
    public WorkerRobot(Movable m, Talkable t, Flyable f) {  
        super(m, t, f);  
    }  
    @Override  
    public void projection() {  
        System.out.println(""  
            Worker robot : Power | 100%  
                Brain | 20%  
            "");  
    }  
}
```

## CompanionRobot

```
public class CompanionRobot extends Robot {  
    public CompanionRobot(Movable m, Talkable t, Flyable f) {  
        super(m, t, f);  
    }  
    @Override  
    public void projection() {  
        System.out.println(""  
            Companion robot : Power | 30%  
                Brain | 100%  
            "");  
    }  
}
```

## ParrotRobot

```
public class ParrotRobot extends Robot {  
    public ParrotRobot(Movable m, Talkable t, Flyable f) {  
        super(m, t, f);  
    }  
    @Override  
    public void projection() {  
        System.out.println(""  
            Parrot robot : Power | 30%  
                Brain | 10%  
                Speed | 40 km/h  
                Color | Green  
            "");  
    }  
}
```

## Talkable interface & Talking strategies

```
public interface Talkable {  
    void talk();  
}  
public class NoTalk implements Talkable {  
    @Override  
    public void talk() {  
        System.out.println("Can't Talk");  
    }  
}  
public class NormalTalk implements Talkable {  
    @Override  
    public void talk() {  
        System.out.println("Talks Normally");  
    }  
}
```

## Movable interface & Moving strategies

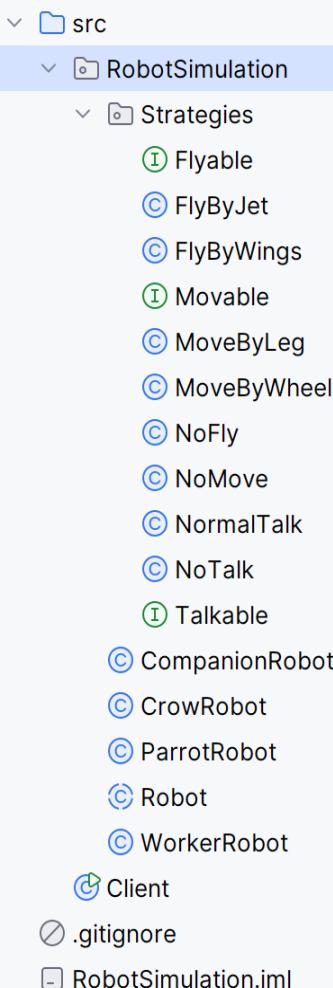
```
public interface Movable {  
    void move();  
}  
public class NoMove implements Movable {  
    @Override  
    public void move() {  
        System.out.println("Can't Move");  
    }  
}  
public class MoveByLeg implements Movable {  
    @Override  
    public void move() {  
        System.out.println("Move using Legs");  
    }  
}  
public class MoveByWheel implements Movable {  
    @Override  
    public void move() {  
        System.out.println("Move using Wheels");  
    }  
}
```

## Flyable interface & Flying strategies

```
public interface Flyable {  
    void fly();  
}  
public class NoFly implements Flyable {  
    @Override  
    public void fly() {  
        System.out.println("Can't Fly");  
    }  
}  
public class FlyByWings implements Flyable {  
    @Override  
    public void fly() {  
        System.out.println("Fly using Wings");  
    }  
}  
public class FlyByJet implements Flyable {  
    @Override  
    public void fly() {  
        System.out.println("Fly using Jet");  
    }  
}
```

## Clint Code

```
public class Client {  
    public static void main(String[] args) {  
        Robot companion = new CompanionRobot(  
            new MoveByLeg(),  
            new NormalTalk(),  
            new NoFly());  
        Robot worker = new WorkerRobot(  
            new MoveByWheel(),  
            new NoTalk(),  
            new NoFly());  
        Robot parrot = new ParrotRobot(  
            new MoveByLeg(),  
            new NormalTalk(),  
            new FlyByWings());  
        Robot crow = new CrowRobot(  
            new MoveByLeg(),  
            new NoTalk(),  
            new FlyByWings());  
  
        companion.projection();  
        worker.projection();  
        parrot.projection();  
        crow.projection();  
  
        companion.move();  
        worker.move();  
        parrot.talk();  
        crow.fly();  
    }  
}
```



## CrowRobot

```
public class CrowRobot extends Robot {  
    public CrowRobot(Movable m, Talkable t, Flyable f) {  
        super(m, t, f);  
    }  
    @Override  
    public void projection() {  
        System.out.println(""  
            + "Crow robot : Power | 40%"  
            + "Brain | 20%"  
            + "Speed | 45 km/h"  
            + "Color | Black"  
            + "");  
    }  
}
```

## -----Output of Client Code-----

Companion robot : Power | 30%  
Brain | 100%

Worker robot : Power | 100%  
Brain | 20%

Parrot robot : Power | 30%  
Brain | 10%  
Speed | 40 km/h  
Color | Green

Crow robot : Power | 40%  
Brain | 20%  
Speed | 45 km/h  
Color | Black

Move using Legs  
Move using Wheels  
Talks Normally  
Fly using Wings

## Conclusion:

- Encapsulate what varies & keep it separate from what remains same.
- Solution to inheritance is not more inheritance.
- Composition should be favoured over inheritance.
- Don't repeat yourself (DRY).