May
21-05-2025
Wednesday

Day-10

Lecture-8

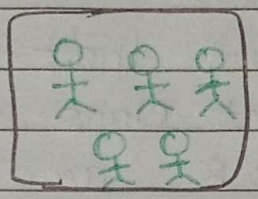
Strategy Design Pattern

* Design Pattern

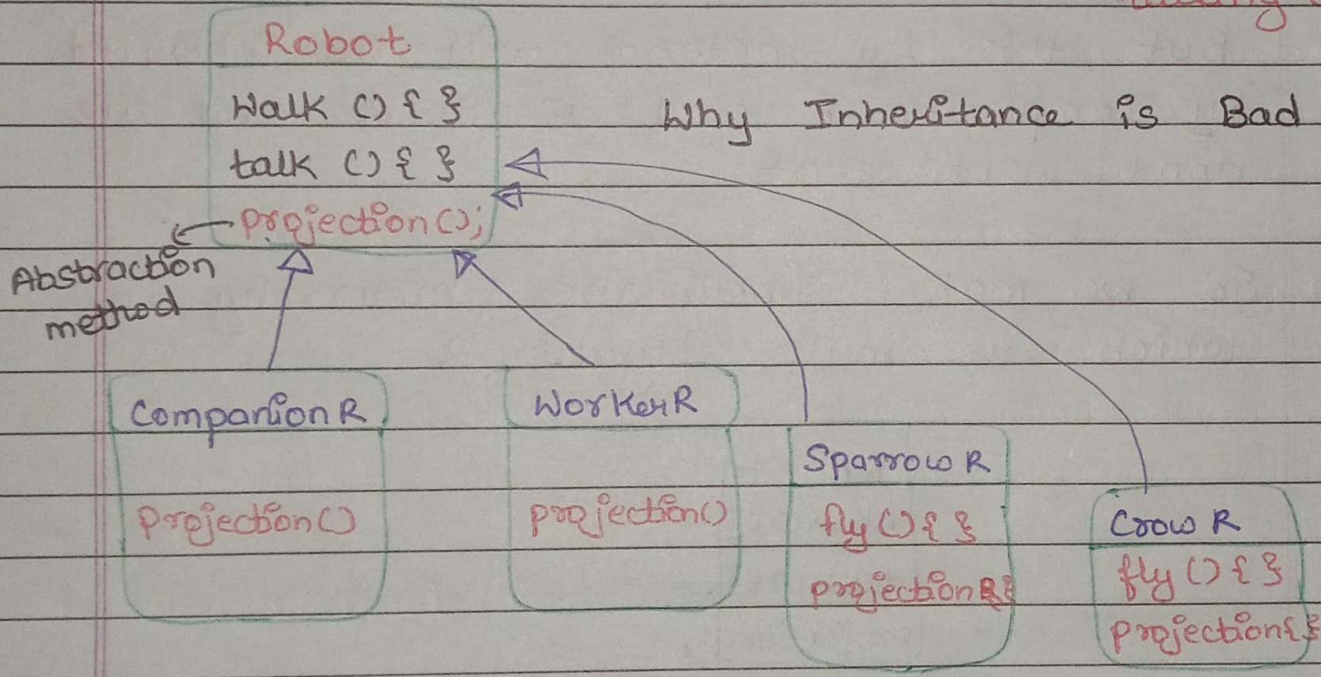
It is like reusable solutions to common problems that you might face while building software systems.

* Strategy Design Pattern

Application →



We have to stimulate robot like walking, talking etc.



→ As we know inheritance inherits method from parent class and projection is abstract class so child class define at their own

→ Next in future some sparrow R come which can fly but Robot class don't have that method so we made it in child class.

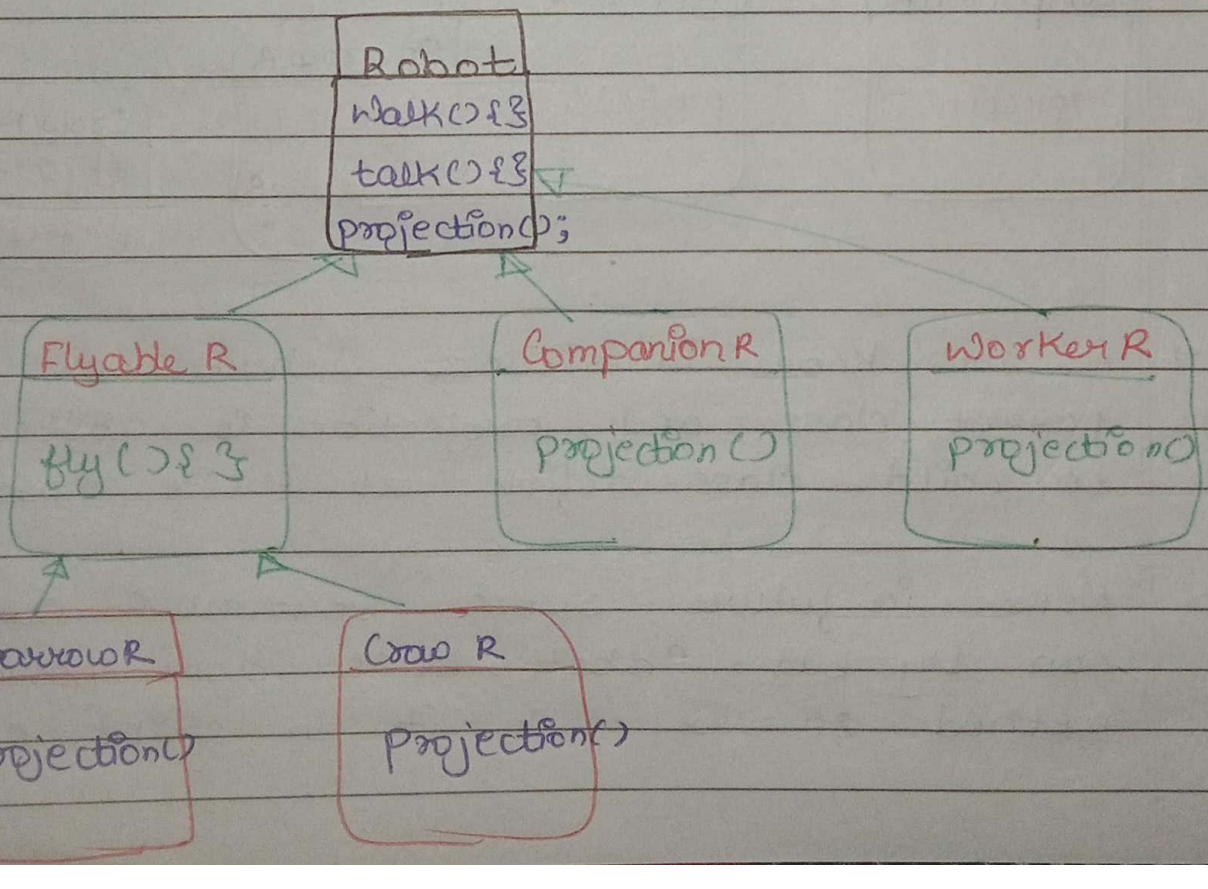
→ But Again another robot come which can fly so we have to copy and paste the code from SparrowR to that class.

→ So this breaking a basic principle DRY (Do not Repeat yourself)

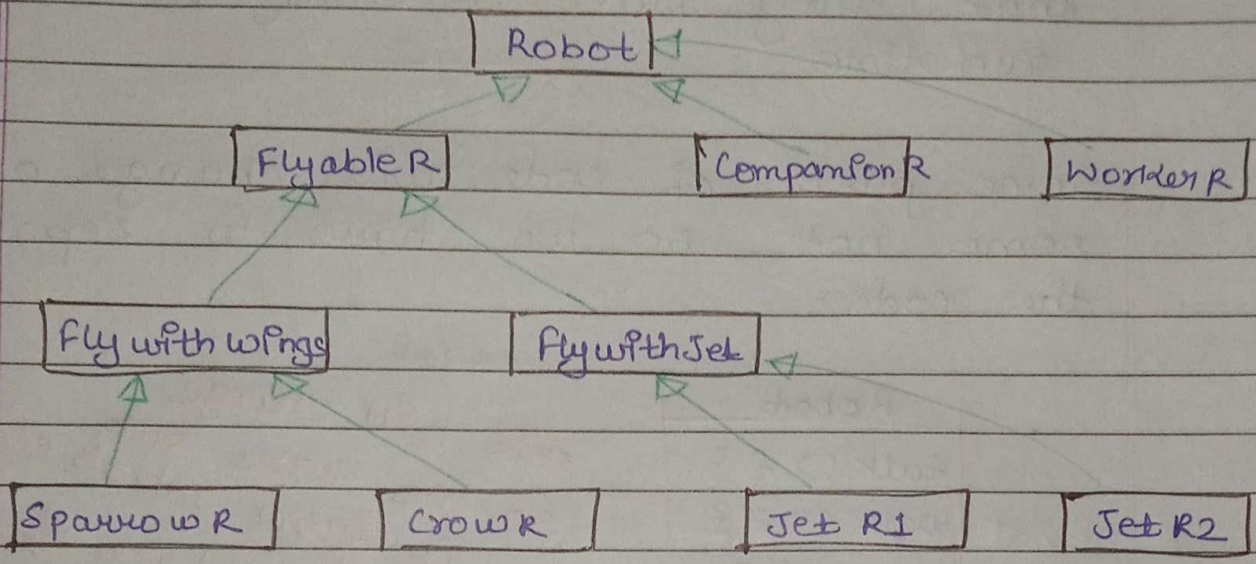
→ We think that if we put fly method in parent class then it will easy for SparrowR and CrowR

→ But this fly method will be inherited by CompanionR and WorkerR and we don't want that

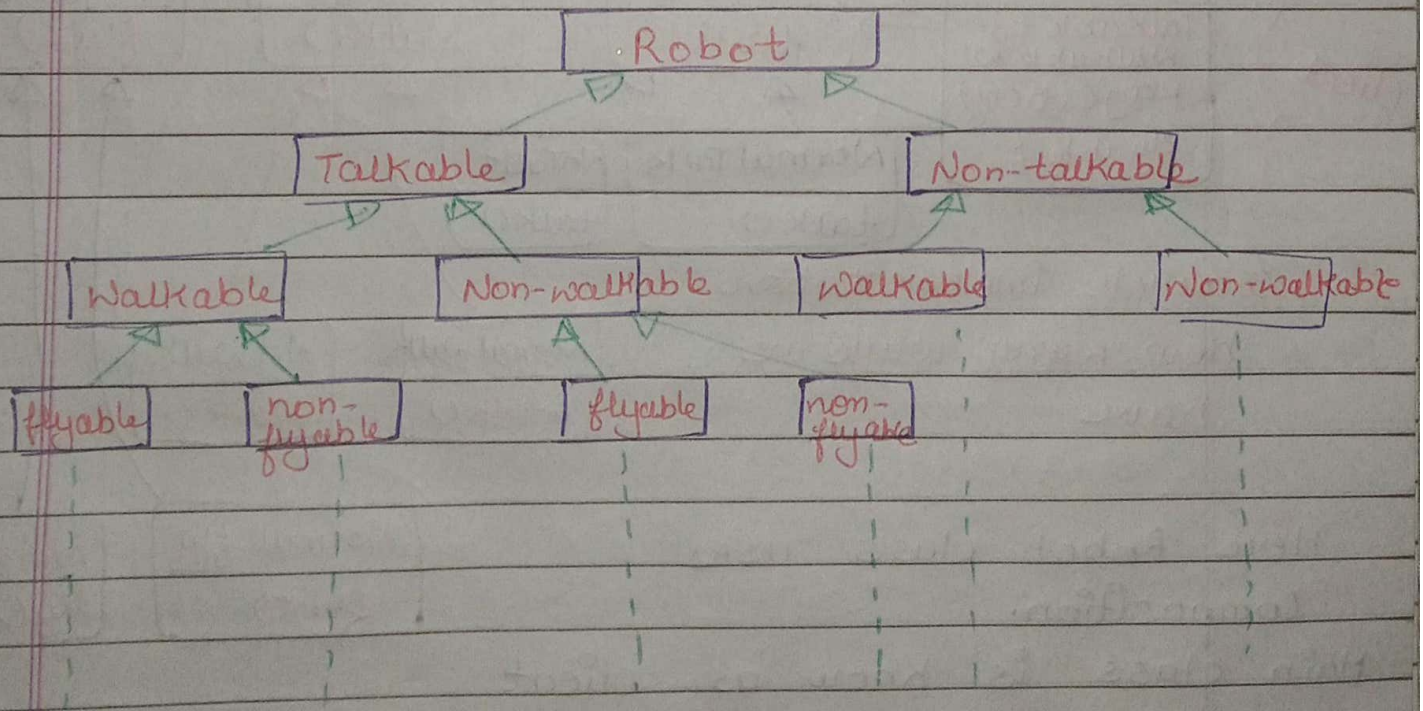
→ So we make inheritance hierarchy in which we make flyableR



→ But what if another type of robots come who fly with jet then we again have to make another class with fly with Jet.



→ Problem will arises like non walking (), non flying, non talking then the hierarchy become complicated.



Problems with Inheritance

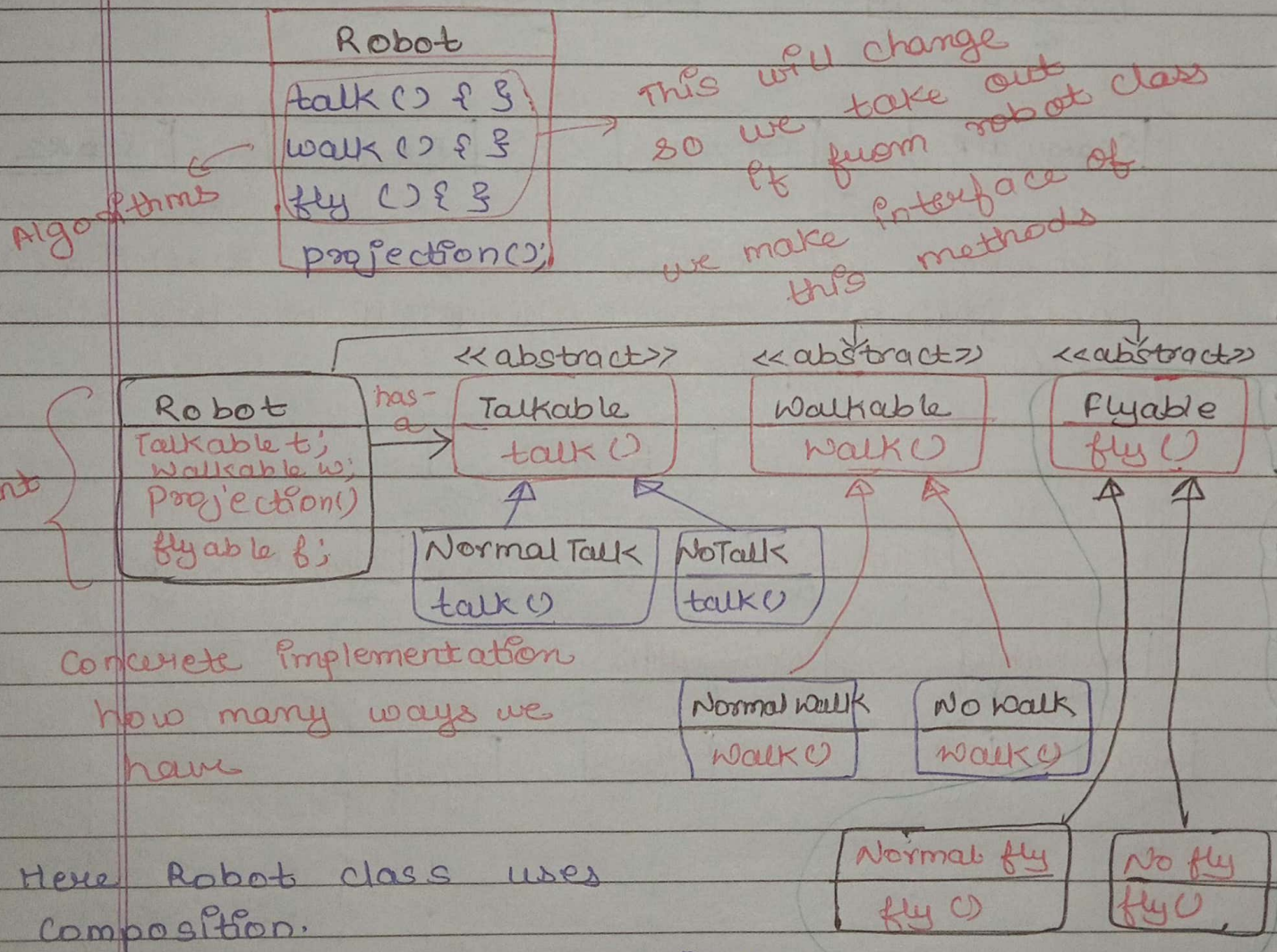
→ Code Reuse → To add new features lot of changes are required

→ Breaking OCP

Defination of Design Pattern

Defines a family of algorithm, put them into seperate classes so that they can be changed at run time

Some part of code will change or some not so we have to separate the code.

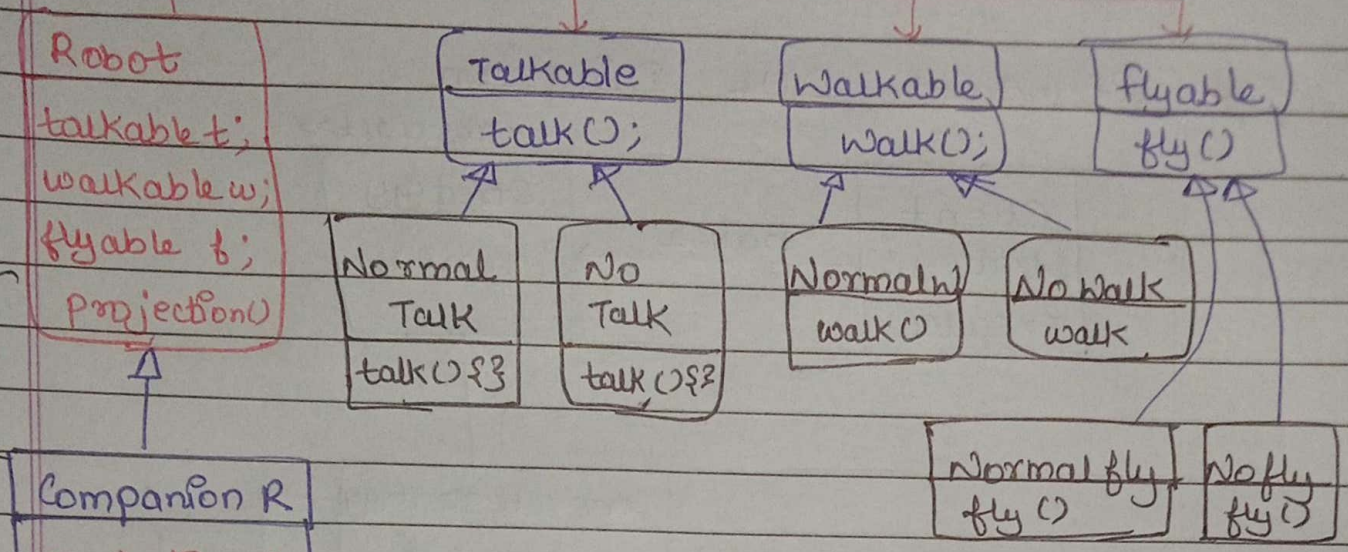


Here Robot class uses Composition.

Main class is know as client

Algorithms are known as strategies

act as dumb object it doing nothing it delegated the work



Robot * robot = new CompanionR (
new NormalTalk();
new normalWalk();
new NonflyableC();
)

Here we Can create any method

Strategy Design pattern say favour composition over inheritance

Here we are using inheritance. So we can replace it by making projectable abstract class

Projectable
projection();

we can add this in above diagram and then we don't need child class like Companion Robot

Robot * r = new Robot → to make a robot
(talking
walking
flying
projection
)

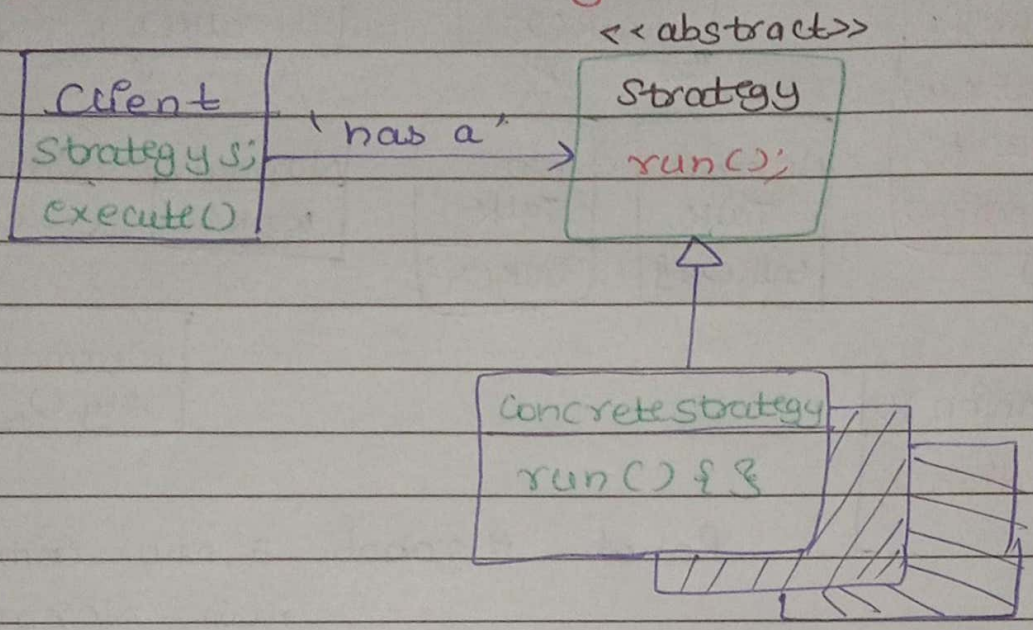
```cpp
#include <iostream>
using namespace std;

// --- Strategy Interface for Walk ---
class WalkableRobot {
public:
    virtual void walk() = 0;
    virtual ~WalkableRobot() {}
};

// --- Concrete Strategies for walk ---
class NormalWalk : public WalkableRobot {
public:
    void walk() override {
        cout << "Walking normally..." << endl;
    }
};

class NoWalk : public WalkableRobot {
public:
    void walk() override {
        cout << "Cannot walk." << endl;
    }
};


```

```cpp
// --- Strategy Interface for Talk ---
class TalkableRobot {
public:
    virtual void talk() = 0;
    virtual ~TalkableRobot() {}
};

// --- Concrete Strategies for Talk ---
class NormalTalk : public TalkableRobot {
public:
    void talk() override {
        cout << "Talking normally..." << endl;
    }
};

class NoTalk : public TalkableRobot {
public:
    void talk() override {
        cout << "Cannot talk." << endl;
    }
};

// --- Strategy Interface for Fly ---
class FlyableRobot {
public:
    virtual void fly() = 0;
    virtual ~FlyableRobot() {}
};

class NormalFly : public FlyableRobot {
public:
    void fly() override {
        cout << "Flying normally..." << endl;
    }
};
```

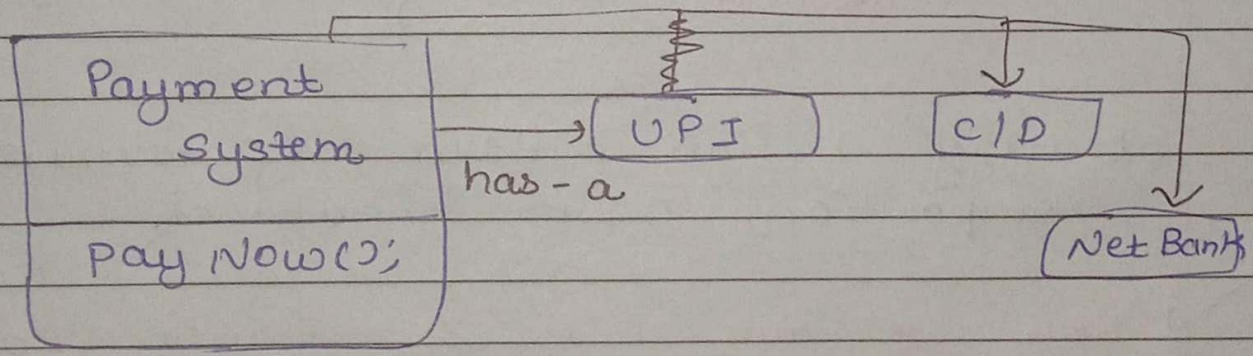
```
62
63  class NoFly : public FlyableRobot {
64  public:
65      void fly() override {
66          cout << "Cannot fly." << endl;
67      }
68  };
69
70  // --- Robot Base Class ---
71  class Robot {
72  protected:
73      WalkableRobot* walkBehavior;
74      TalkableRobot* talkBehavior;
75      FlyableRobot* flyBehavior;
76
77  public:
78      Robot(WalkableRobot* w, TalkableRobot* t, FlyableRobot* f) {
79          this->walkBehavior = w;
80          this->talkBehavior = t;
81          this->flyBehavior = f;
82      }
83
84      void walk() {
85          walkBehavior->walk();
86      }
87      void talk() {
88          talkBehavior->talk();
89      }
90      void fly() {
91          flyBehavior->fly();
92      }
93
94      virtual void projection() = 0; // Abstract method for subclasses
95  };
96
97  // --- Concrete Robot Types ---
98  class CompanionRobot : public Robot {
99  public:
100     CompanionRobot(WalkableRobot* w, TalkableRobot* t, FlyableRobot* f)
101         : Robot(w, t, f) {}
102
103     void projection() override {
104         cout << "Displaying friendly companion features..." << endl;
105     }
106 };
```

```
};

class WorkerRobot : public Robot {
public:
    WorkerRobot(WalkableRobot* w, TalkableRobot* t, FlyableRobot* f)
        : Robot(w, t, f) {}

    void projection() override {
        cout << "Displaying worker efficiency stats..." << endl;
    }
};

// --- Main Function ---
int main() {
    Robot *robot1 = new CompanionRobot(new NormalWalk(), new NormalTalk(), new NoFly());
    robot1->walk();
    robot1->talk();
    robot1->fly();
    robot1->projection();

    cout << "--------------------" << endl;

    Robot *robot2 = new WorkerRobot(new NoWalk(), new NoTalk(), new NormalFly());
    robot2->walk();
    robot2->talk();
    robot2->fly();
    robot2->projection();

    return 0;
}
```

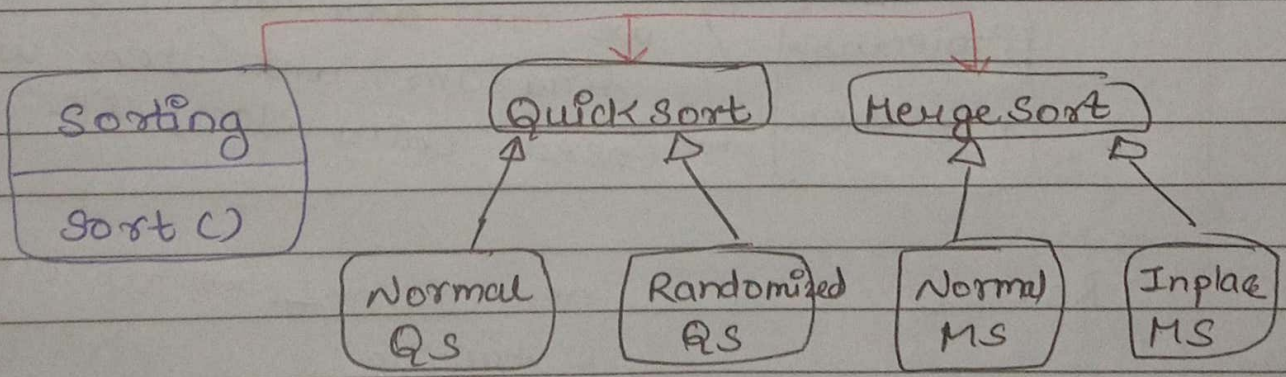
* Standard UML Diagram



* Real- Life Example



Payment system has different kind of strategies



* Conclusion

→ Encapsulate what varies and Keep it separate from what remains same.

→ Solution to inheritance is not more inheritance

→ Composition should be favoured over inheritance

→ Code the interface and not to concretion

→ Do Not Repeat Yourself.