DESIGN PATTERNS

Creational :

1. Singleton
2. Factory Method
3. Abstract Factory
4. Builder
5. Prototype.

Structural :

1. Adapter (& Wrapper)
2. Composite
3. Proxy.
4. Flyweight
5. Facade
6. Bridge
7. Decorator.

Behavioral :

1. Observer
2. Strategy
3. Command
4. Iterator
5. State
6. Visitor
7. Mediator.
8. Memento
9. Chain of Responsibility
10. Template Method.
11. Interpreter

Compound Pattern
<=>
Patterns of Patterns.

INTRODUCTION TO DESIGN PATTERNS

"Do not want to reinvent the wheel"

Design Patterns :-

* Best practices addressing recurring solutions.
* Avoid Reinventing the Wheel.
* Improve Code Readability and Maintainability
* Facilitate Communication.
* Promote Best Practices
* Adaptability and Scalability.
* Enhance Efficiency

Example :-

```
public class Aircraft {
    private String type;
    public Aircraft (String type) {
        this.type = type;
    }
}
```

⇒ Adding additional properties after sometime:

```
public class Aircraft {
    private String type;
    private String color;
    public Aircraft (String type) {
        this.type = type;
    }
    public Aircraft (String type, String color) {
        this.type = type;
        this.color = color;
    }
}
```

⇒ Adding few more properties

Starting from one field and increasing to many fields, the constructor arguments increase and look like telescope.

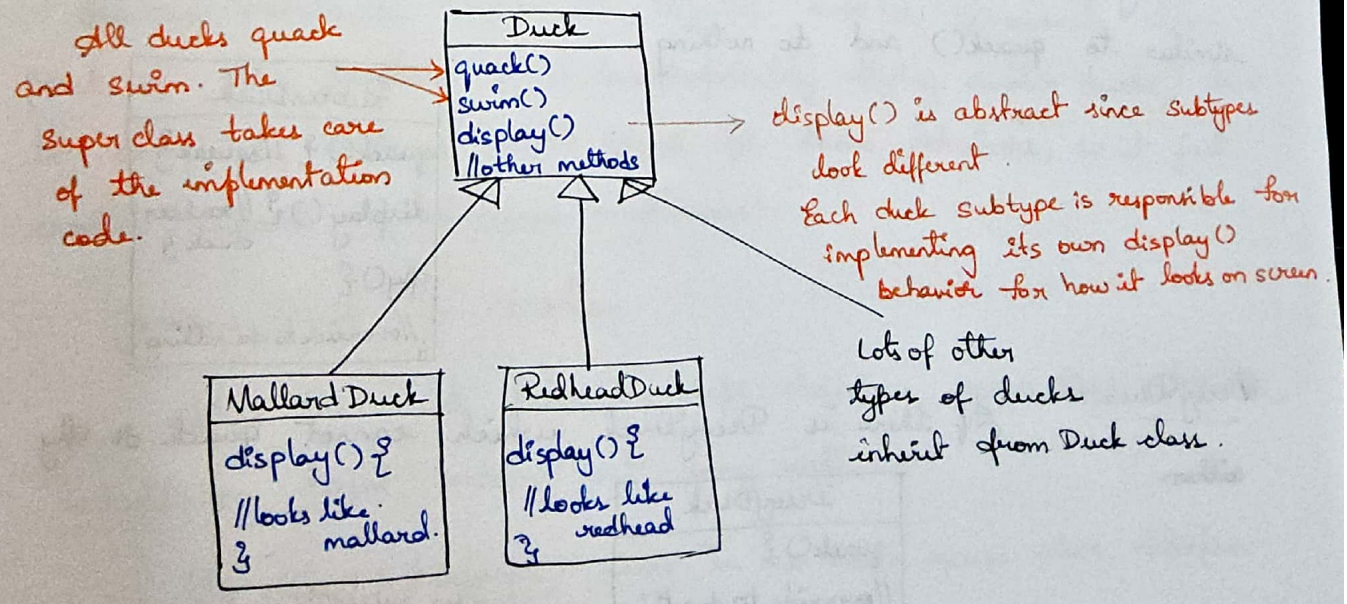
```
Aircraft(String type)
Aircraft(String type, String color)
Aircraft(String type, String color, String prop3)
Aircraft(String type, String color, String prop3, String prop4)
```

The telescopic pattern is called an anti-pattern: how NOT to do things.

The approach to solve this increasing number of variables is to use Builder Pattern.

Introduction to Design Patterns

① Simple SimUDuck app: This is a duck pond simulation game. It shows variety of ducks in pond "swimming" and "quacking".

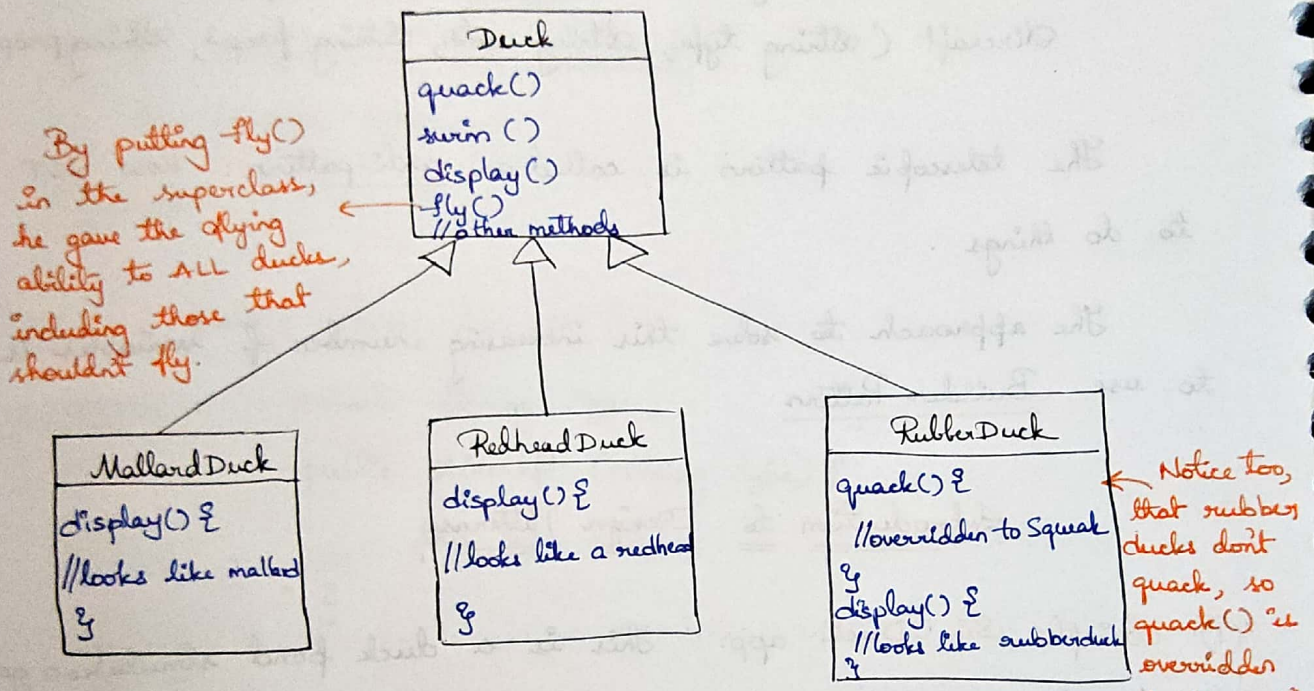


② But now we need the ducks to FLY:

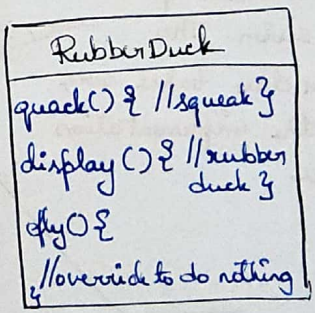
we need flying ducks, hence we add fly() method to Duck class and all subtypes inherit it.

③ But there are rubber duckies that inherit Duck class:

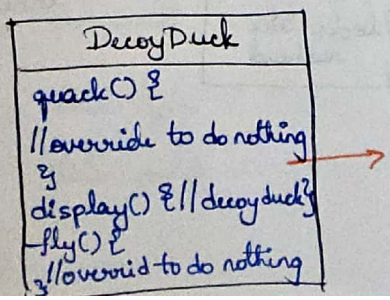
We used inheritance for the purpose of reuse but it hasn't turned out so well when it comes to maintenance.



Using inheritance, we can also override fly() method in RubberDuck similar to quack() and do nothing.

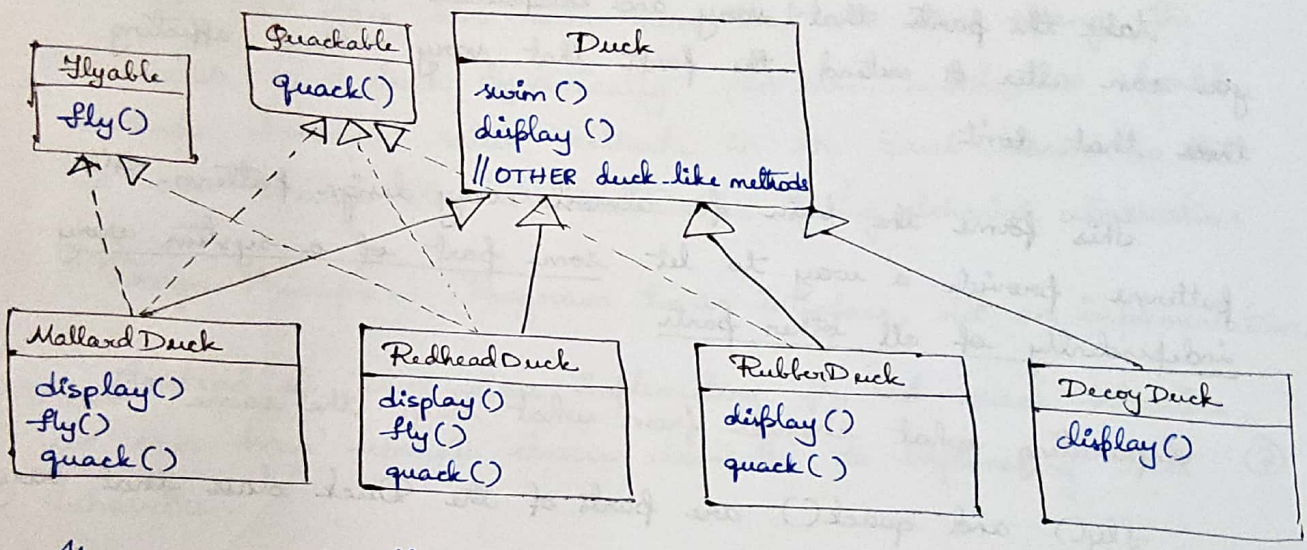


DecoyDuck:- If there is DecoyDuck which cannot quack or fly either.



④ How about interface?

Since not all ducks can fly & quack, we can create Flyable interface with a fly() method and Quackable interface with a quack() method and only let new subclasses that can fly & quack implement these interfaces.



If you override flying behavior from an interface, then what if a small change in flying behavior required. We cannot update small change in many subclasses.

Having subclasses implement Flyable and/or Quackable solves part of the problem (no inappropriately flying rubber ducks) but it completely destroys code reuse for those behaviors, so it just creates different maintenance nightmare.

⑤ Zeroing in on the problem.

Since the duck behavior keeps changing across the subclasses, inheritance hasn't worked out very well.

By using interfaces, there is no code reuse since interface has no implementation.

In either case, whenever you need to modify a behaviour, you're often forced to track down and change it in all the different subclasses where that behaviour is defined.

Design Principle :- Identify the aspects of your application that vary and separate them from what stays the same.

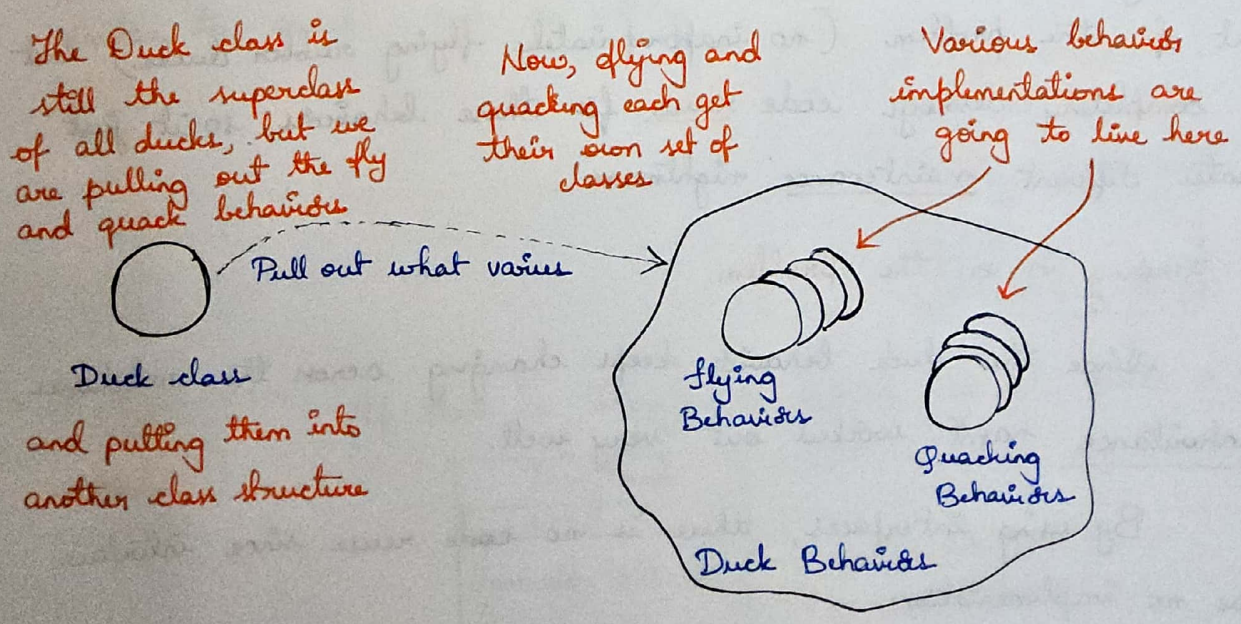
take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't.

This forms the basis for almost every design pattern. All patterns provide a way to let some part of a system vary independently of all other parts.

(6) Separating what changes from what stays the same.

fly() and quack() are parts of the Duck class that vary across ducks.

To separate these behaviours from the Duck class, we'll pull both methods out of the Duck class and create a new set of classes to represent each behavior.



⑦ Designing the Duck Behaviors

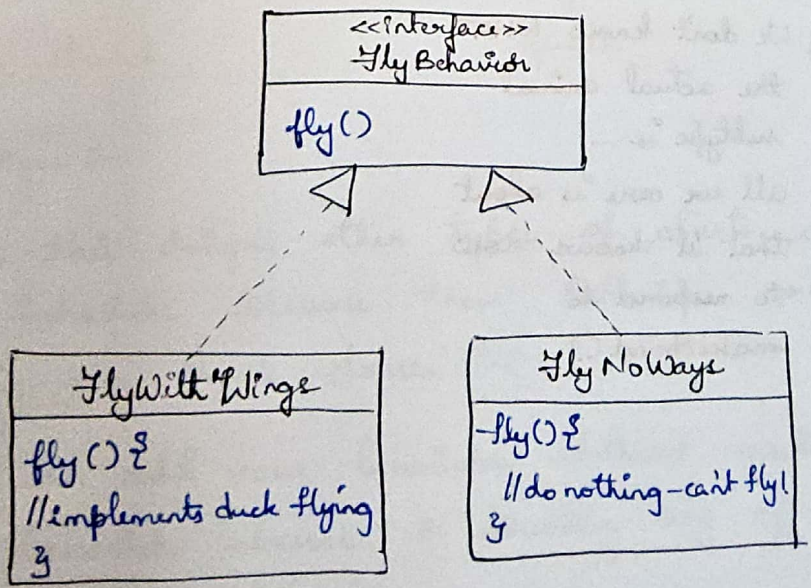
* Idea is to keep things flexible.
* We want to assign behaviors to the instances of Duck.
* For example, instantiate new MallardDuck instance and initialize it with a specific type of flying behavior.
* We can also make sure that we can change the behavior of a duck dynamically. In other words, we should include behavior setter methods in the Duck classes so that we can change the MallardDuck's flying behavior at runtime.

Design Principle:- Program to an interface, not an implementation.

Instead of Duck class implementing fly and quack behaviors, we can have separate classes dedicated to implementing these behaviors.

It's the behavior class, rather than Duck class, that will implement the behavior interface.

That way, Duck classes won't need to know any of the implementation details for their own behaviors.



Why interface for FlyBehavior? As this is about polymorphism, you can do same thing with an abstract superclass.

Ans- "Program to an interface" means

"Program to a supertype".

(or)

"the declared type of the variables should be a supertype, usually an abstract class or interface, so that the objects assigned to those variables can be of any concrete implementation of the supertype, which means the class declaring them doesn't have to know about the actual object types".

Programming to an implementation

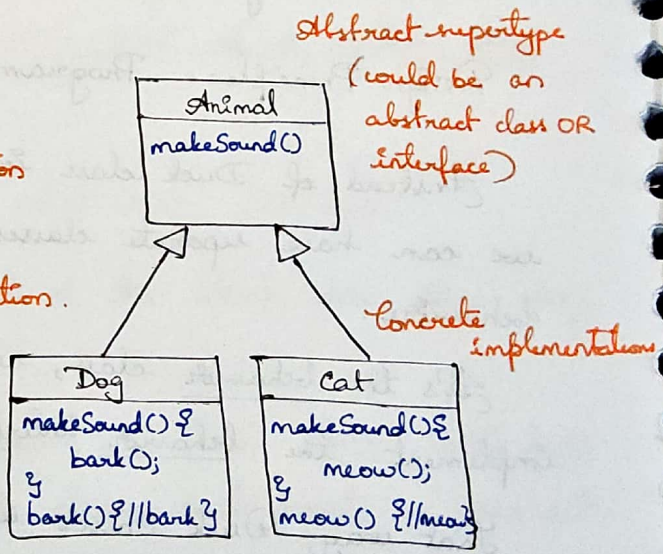
```
Dog d = new Dog();
d.bark();
```

Declaring "d" as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation.

Programming to an interface

```
Animal a = new Dog();
a.makeSound();
```

We know it's a Dog, but we can now use the animal reference polymorphically.

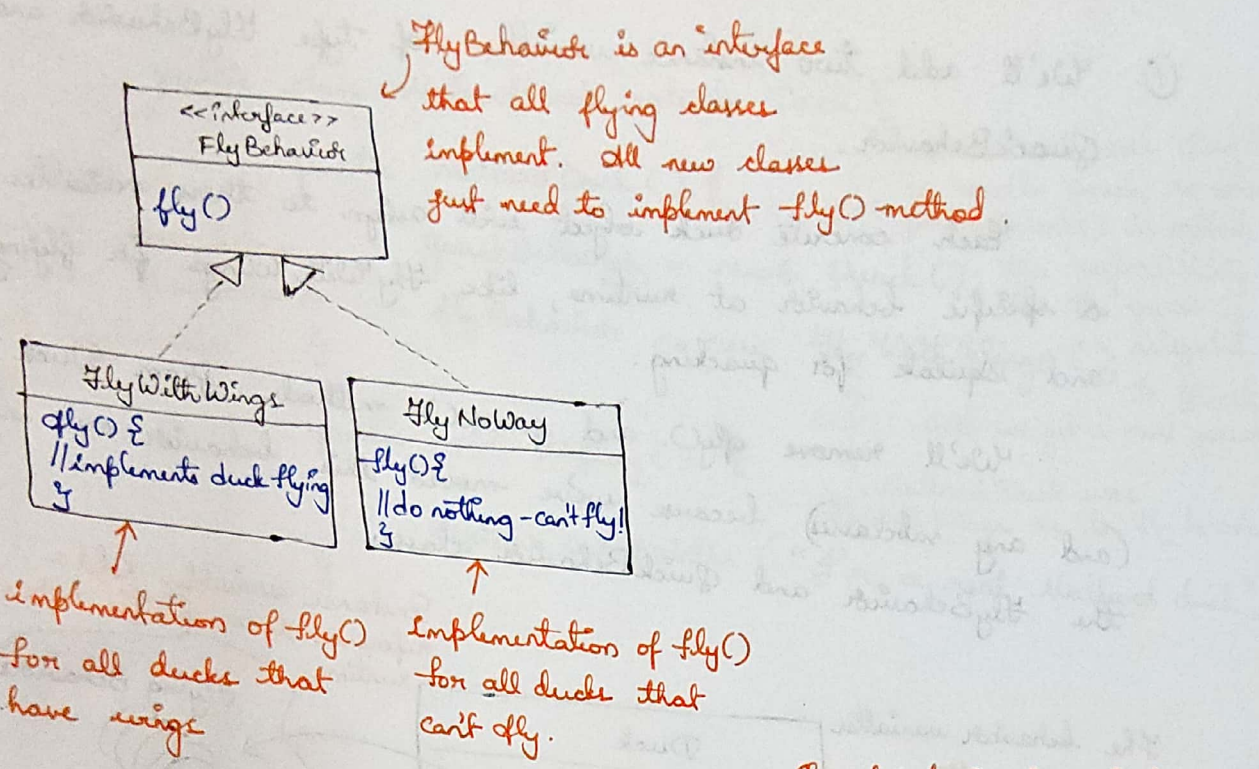


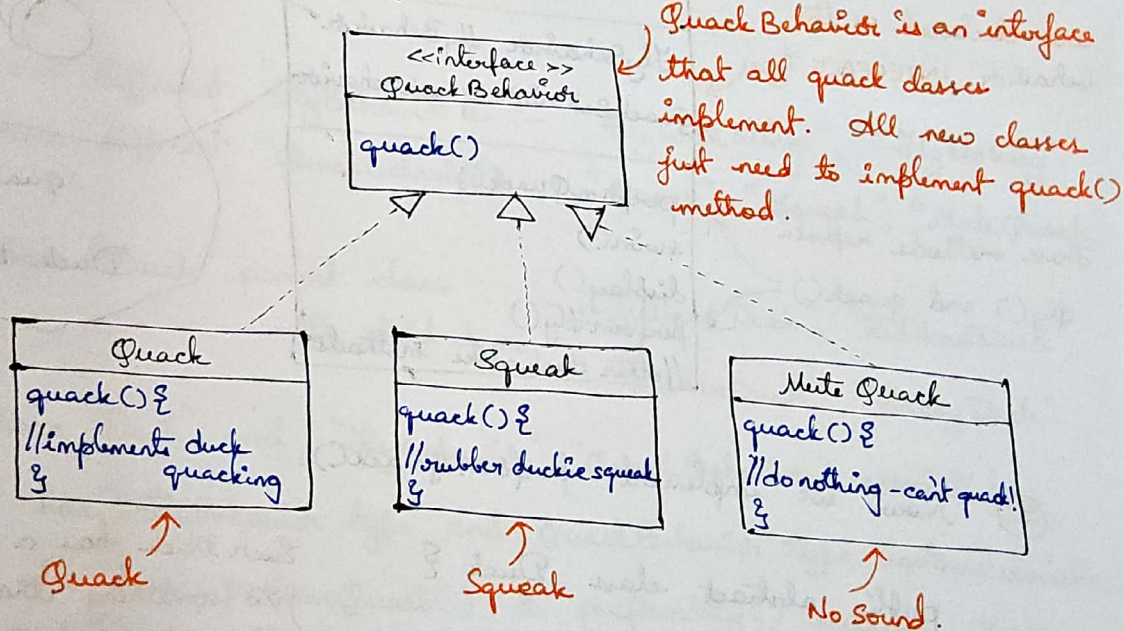
Assign concrete implementation object at runtime instead of hardcoding the instantiation.

```
a = getAnimal();
a.makeSound();
```

We don't know WHAT the actual animal subtype is ... all we care is about that it knows how to respond to makeSound().

⑧ Implementing the Duck Behaviors.





Advantages :-

With this design, other types of objects can re-use our fly and quack behaviors because these behaviors are no longer hidden away in our Duck classes.

We can add new behaviors without modifying any of our existing behavior classes or touching any of the Duck classes that use flying behaviors.

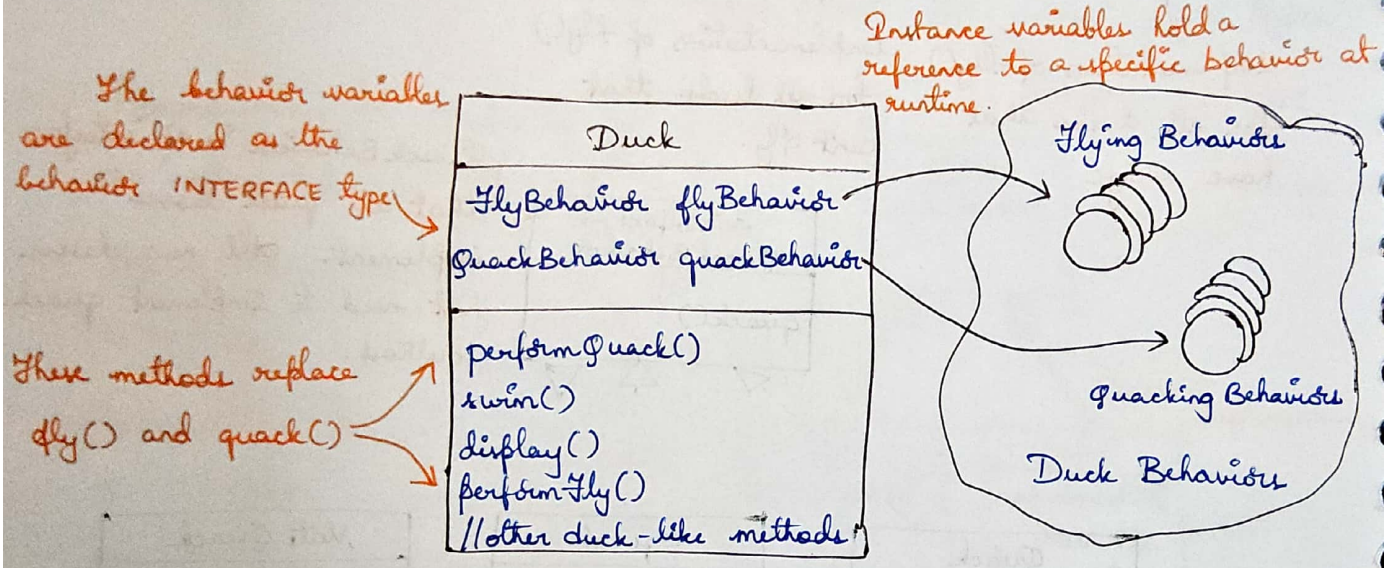
We get the benefit of REUSE without using inheritance.

⑨ Integrating the Duck Behaviors

① We'll add two instance variables of type FlyBehavior and QuackBehavior.

Each concrete duck object will assign to those variables a specific behavior at runtime, like FlyWithWings for flying and Squeak for quacking.

We'll remove fly() and quack() methods from Duck class (and any subclasses) because we've moved this behavior out into the FlyBehavior and QuickBehavior classes.



② Now we implement performQuack():

```
public abstract class Duck {
    QuackBehavior quackBehavior;
    //more
    public void performQuack() {
        quackBehavior.quack();
    }
}
```

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.

③ How to set flyBehavior and quackBehavior instance variables

```
public class MallardDuck extends Duck {
    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }
    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}
```

MallardDuck inherit instance variables "quackBehavior" and "flyBehavior" from Duck.

MallardDuck uses Quack to handle quack, so when performQuack() is called, the responsibility for the quack is delegated to Quack and we get a real quack

MallardDuck uses FlyWithWings as its fly behavior

Summary till now :-

* We have different "FlyBehavior"s – "FlyWithWings", "FlyNoWay".
* We have different "QuackBehavior"s – "Quack", "Squeak", "MuteQuack".
* We have "Duck" parent class with child classes – "MallardDuck", "RedHeadDuck", "RubberDuck", "DecoyDuck".
* We have "Duck" and "FlyBehavior", "QuackBehavior" associated ⇒
* Duck has FlyBehavior type and QuackBehavior type instance variables.
* Duck also has performQuack() & performFly() methods that delegate quack and fly behaviors (as flyBehavior.fly() & quackBehavior.quack()).
* MallardDuck (an inherited child of Duck) has a constructor that instantiates fly and quack behavior instance variables of parent Duck class.

This means that each type of Duck has its own fly and quack behaviors. Thus, the behaviors can be flexibly updated.

```
MallardDuck() {
    quackBehavior = new Quack();
    flyBehavior = new FlyWithWings();
}
```

⇒ whereas,

```
RubberDuck() {
    quackBehavior = new Squeak();
    flyBehavior = new FlyNoWay();
}
```

When someone asks you to have another Duck type with MuteQuack and FlyNoWay features, you can easily update as:

```
DecoyDuck() {
    quackBehavior = new MuteQuack();
    flyBehavior = new FlyNoWay();
}
```

* But, as we discussed, we should NOT program to an implementation. In the constructor, we are poorly doing this job. The instantiation of Fly and Quack Behaviors are being done inside the concrete class.
* But, we can correct this easily so that we can flexibly apply behaviors at "runtime".
* As of now, we achieved flexibility to some extent by letting the concrete class decide what type of behavior object it wants.
* Time to test the code in GitHub. (v1 to v4)

(10) Setting behavior dynamically :-

Let's say you have a new type of Duck - ModelDuck. You want to set behavior at runtime like below:

```
model.performFly();  // Flying...
set model's flyBehavior as "rocket fly power"
model.performFly();  // Rocket Flying...
```

In order to achieve different behaviors at run times, let the parent Duck introduce setFlyBehavior() and setQuackBehavior() methods that will change behavior at runtime.

(1)

```
public class ModelDuck extends Duck {
    public ModelDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new Quack();
    }                                   } default
    public void display() {
        System.out.println("I'm a model duck");
    }
}
```

(2) Duck class: add below two methods

```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}
public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```

(3) new FlyBehavior type (FlyRocketPowered.java)

```
public class FlyRocketPowered implements FlyBehavior {
    public void fly() {
        System.out.println("flying with rocket");
    }
}
```

(4) Now perform dynamic Behavior on ModelDuck

```
Duck model = new ModelDuck();
model.performFly();
model.setFlyBehavior(new FlyRocketPowered());
model.performFly();
```

o/p:

```
Flying...
Flying with rocket...
```

* Time to test the code in GitHub. (v5)

(11) The Big Picture

Instead of looking at duck behaviors as a "set of behaviors", we can start looking at them as "family of algorithms".

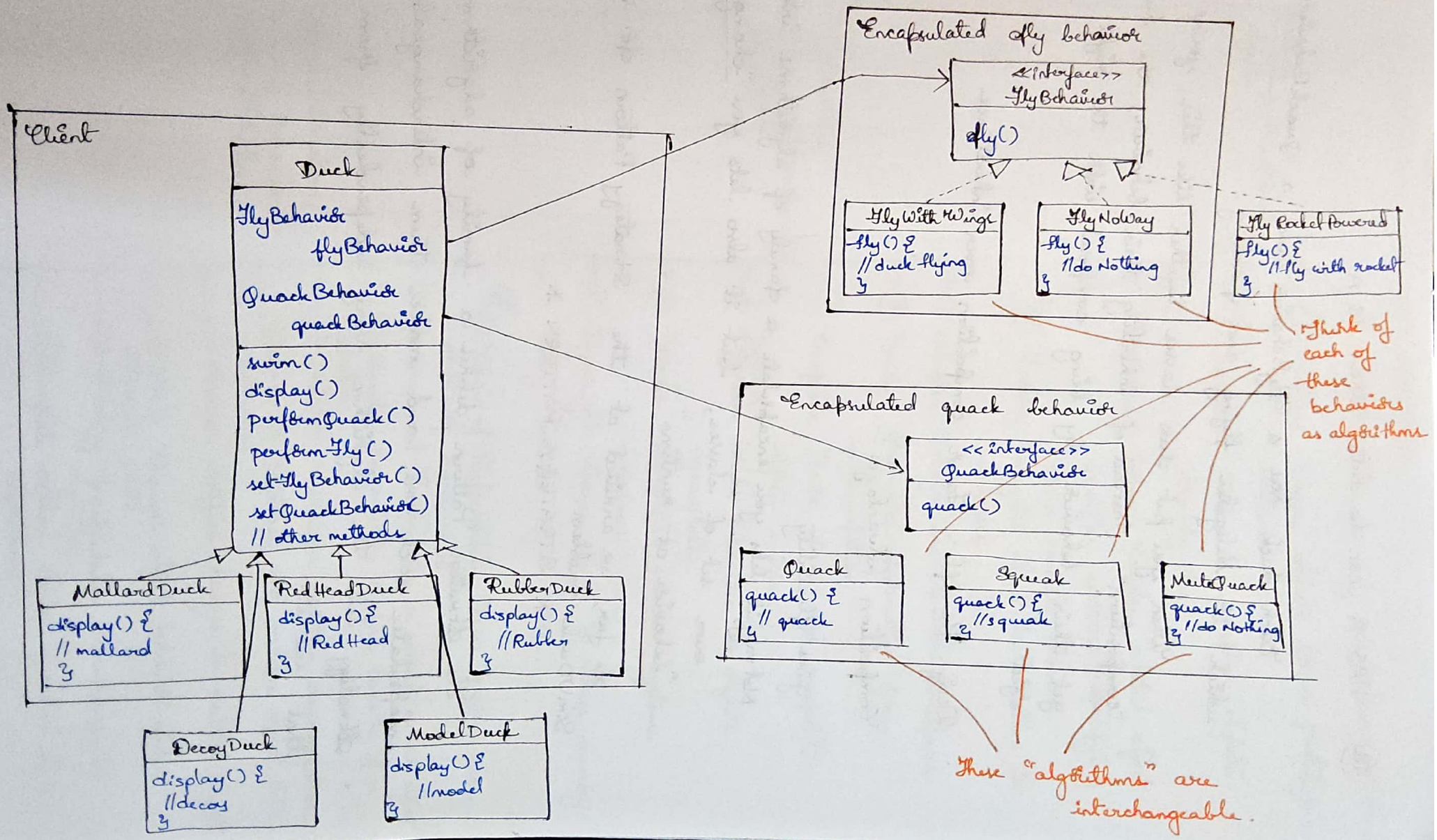
In SimUDesign,

algorithms ⇒ things a duck would do (different ways of flying & quacking)

In StateSalesTax,

algorithms ⇒ ways to compute sales tax by state.

Be mindful about IS-A (—▷), HAS-A (——▷), implements (- - -▷)



(12) HAS-A can be better than IS-A.

Each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking.

When you put two classes together like this you're using "composition". Instead of inheriting their behavior, the ducks get their behavior by being composed with the right behavior object.

Design Principle: Favor composition over inheritance.

Composition advantages:-

gives flexibility

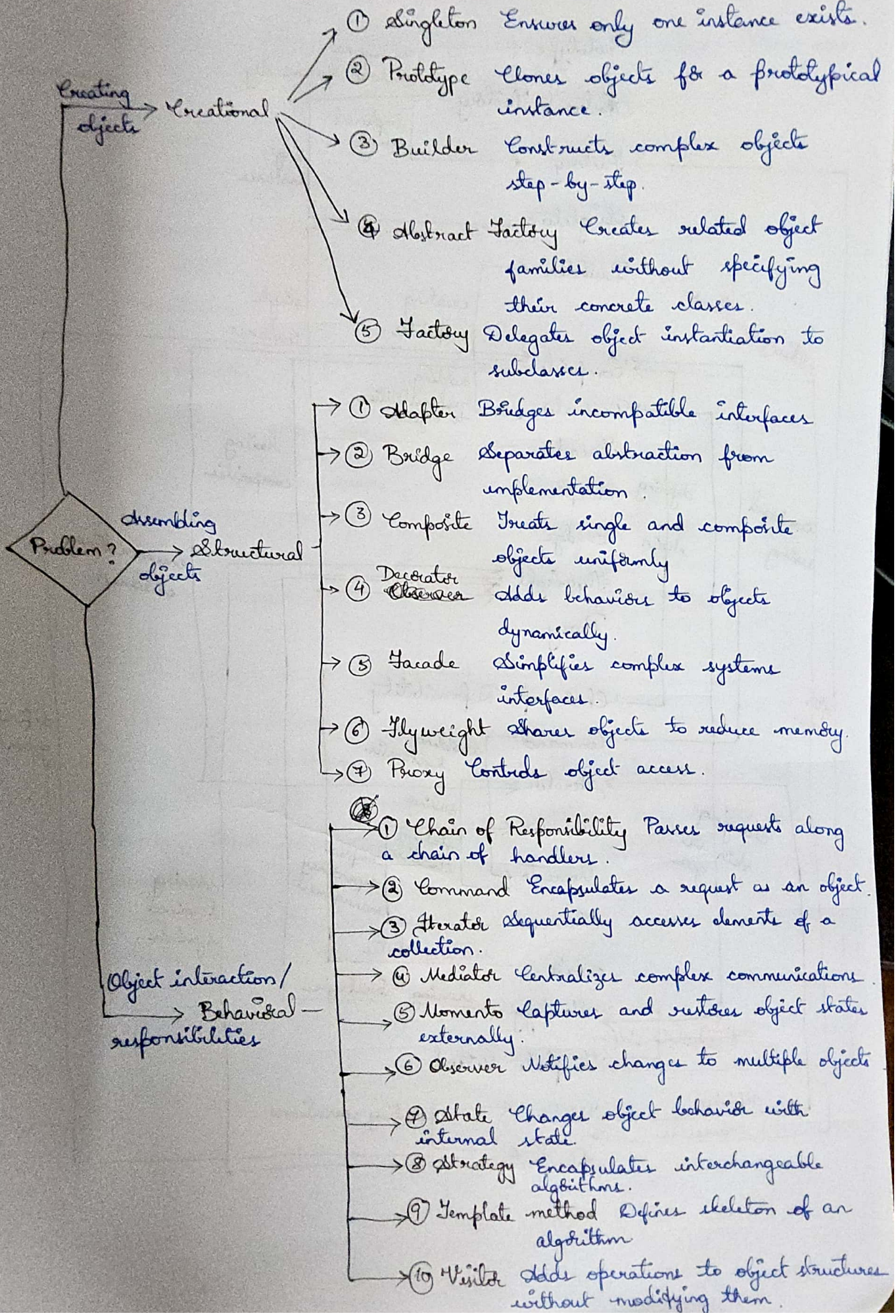
Not only it lets you encapsulate a family of algorithms into their own set of classes, but it also lets you "change behavior at runtime".

So far, we arrived at the Strategy Pattern for solving SimUDuck problem.

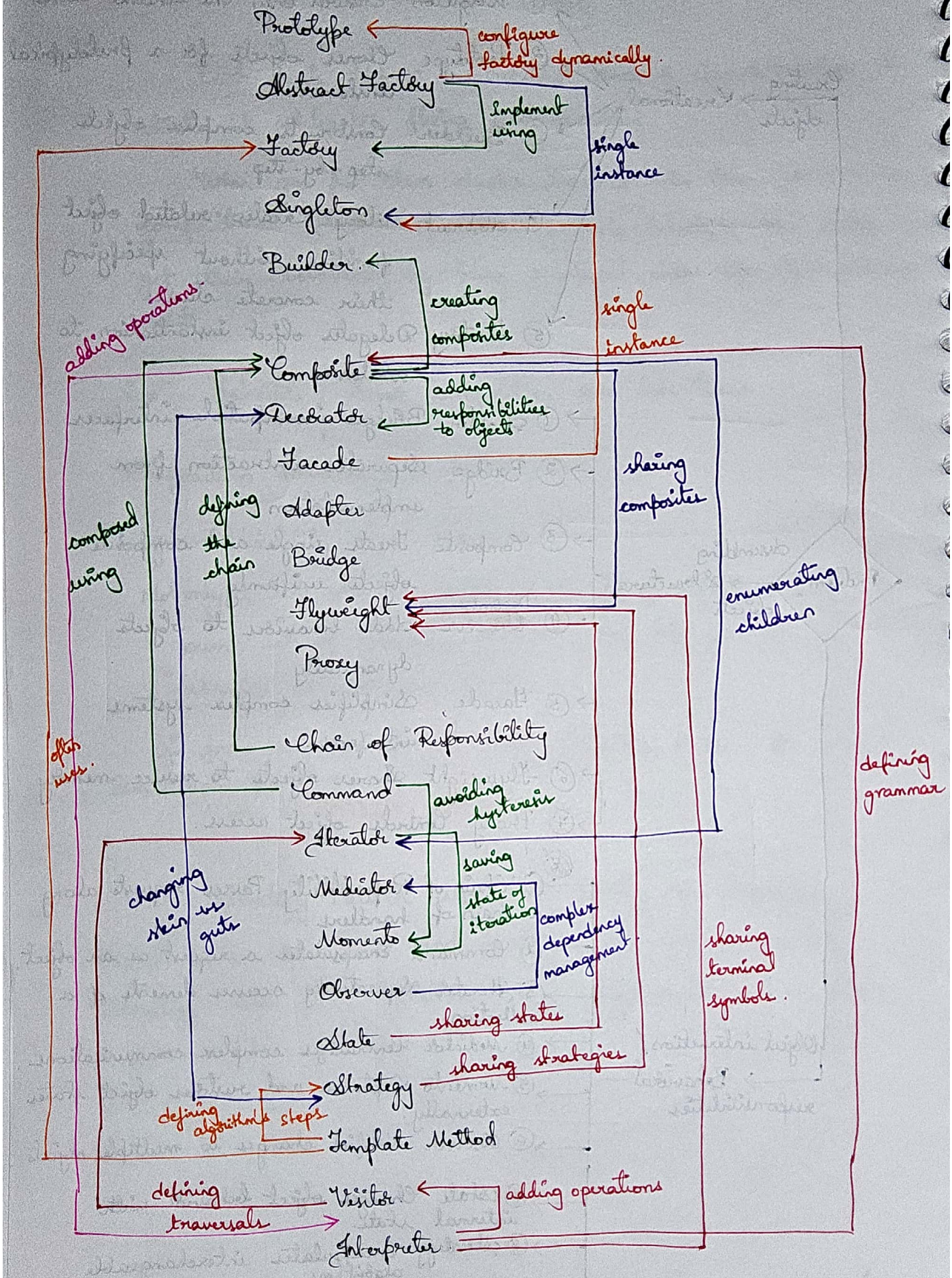
* STRATEGY PATTERN *

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

OVERVIEW OF EACH PATTERN



DESIGN PATTERN RELATIONSHIPS – BY GANG OF FOUR



To design a reusable, extensible and maintainable code; below are necessary:

OO Basics

Abstraction
Encapsulation
Polymorphism
Inheritance

"be very good at OO Basics"

OO Principles

Encapsulate what varies

Favor composition over inheritance

Program to interfaces, not implementations.

"So far we followed these principles. These build a foundation to design patterns. There can be more.."

OO Patterns

Strategy - defines a family of algorithms, encapsulates each one, and makes them interchangable. Strategy lets the algorithm vary independently from clients that use it.

"Think about how each pattern rely on OO basics and principles"