

CSE 3216: Software Design Patterns

Introduction

What is Design Pattern

- Design patterns **represent the best practices** used by experienced object-oriented software developers
- Design patterns are **solutions to general problems** that software developers faced during software development.
- These solutions were obtained **by trial and error** by numerous software developers over quite a substantial period of time.

Types of Design Patterns

- **Creational Design Patterns:** These design patterns provide a way to **create objects while hiding the creation logic**, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding **which objects need to be created** for a given use case.
- **Structural Design Patterns:** These design patterns concern **class and object composition**. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
- **Behavioral Design Patterns:** These design patterns are specifically concerned with **communication** between objects.

Creational Design Patterns

1. Factory (Hide creation logic, refer using interface)
2. Abstract Factory (Factory of factories)
3. Singleton (Single Object Creation)
4. Builder (Builds the final object step by step)
5. Prototype (Creates prototype of an object)

Structural Design Patterns

- 6. Adapter (Works as a bridge between two incompatible interfaces)
- 7. Bridge (Works as a bridge between abstraction and implementation classes)
- 8. Filter (To filter a set of objects)
- 9. Composite (to treat a group of objects in a similar fashion)
- 10. Decorator (decorate existing object with new functionalities, not changing the signature class)
- 11. Facade (Hides complexities of the system and gives interface)
- 12. Flyweight (Reduces number of objects to decrease memory and improve runtime)
- 13. Proxy (Class representing behaviour of another class)

Behavioral Design Patterns

- 14. Chain of responsibility (Creates a chain of receiver object for a request)
- 15. Command (Data driven, request wrapped under an object as command)
- 16. Interpreter (To evaluate expression, grammar)
- 17. Iterator (Accessing a set of sequential objects not knowing underlying representation)
- 18. Mediator (To reduce communication complexity)
- 19. Memento (Persistence quality of backtrack)
- 20. Observer (In One-to-many relationships, an object is changed, the dependent objects need to be notified)
- 21. State (To handle different states)

Behavioral Design Patterns

22.Null Object (To check null)

23.Strategy (Objects representing strategies, context object changes behavior based on the strategy/object)

24.Template (Template abstract class with its different implementation by extended classes)

25.Visitor (Visitor class, changes behaviour of an element class)

Architectural Design Patterns

- 26. Model(Data), View(Visuals), Controller(Like between model and view)
- 27. Business Delegate (To separate business and presentation codes)
- 28. Composite (Objects are represented as graphs and their inbetween connection is maintained)
- 29. Data Access Object (To separate low and high level data maintenance)
- 30. Front Controller (Single server based system)
- 31. Intercepting Filter (To conduct some pre-processing or post processing)
- 32. Service Locator (To locate services)
- 33. Transfer Object (How to transfer object with multiple attributes)

Basics of Object Oriented Principles

- Abstraction
 - To hide what is happening underneath
 - Public Interface, private implementation
- Inheritance
 - Abstracting a class and its definition, extend it to use its functionality
 - To increase reusability
- Encapsulation
 - Setting public, private to have access control
- Polymorphism
 - Dynamic runtime behaviour of different functionalities in different objects
 - Follow dependency inversion

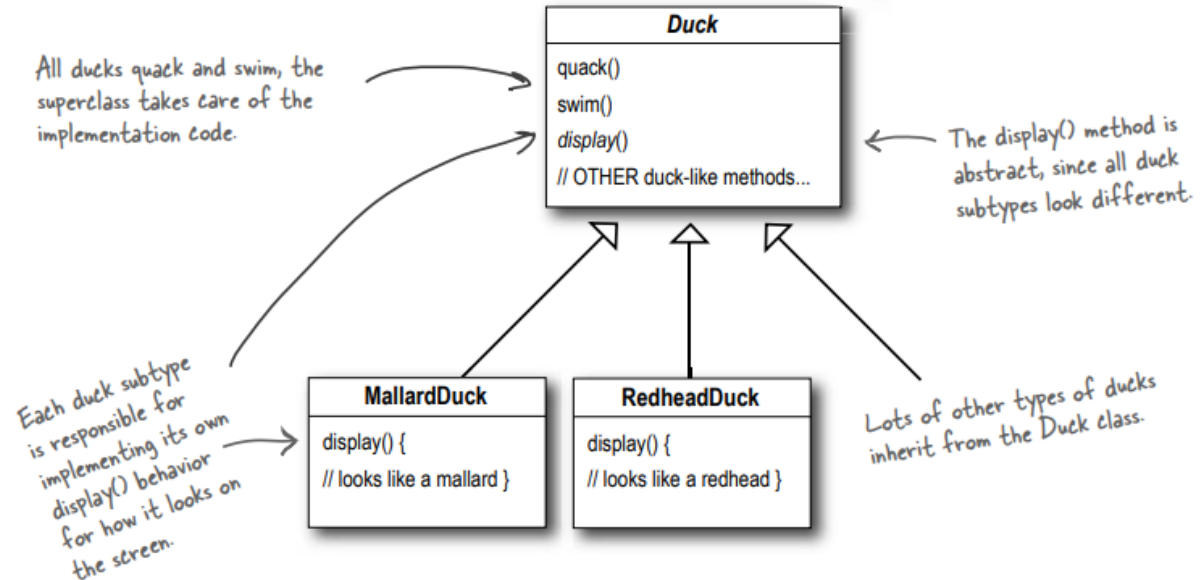
Chapter 1. Intro to Design Patterns (Strategy Pattern)

It started with a simple SimUDuck app

Joe works for a company that makes a highly successful duck pond simulation game, *SimUDuck*. The game can show a large variety of duck species swimming and making quacking sounds. The initial designers of the system used standard OO techniques and created one Duck superclass from which all other duck types inherit.

Arrow means (Extends)

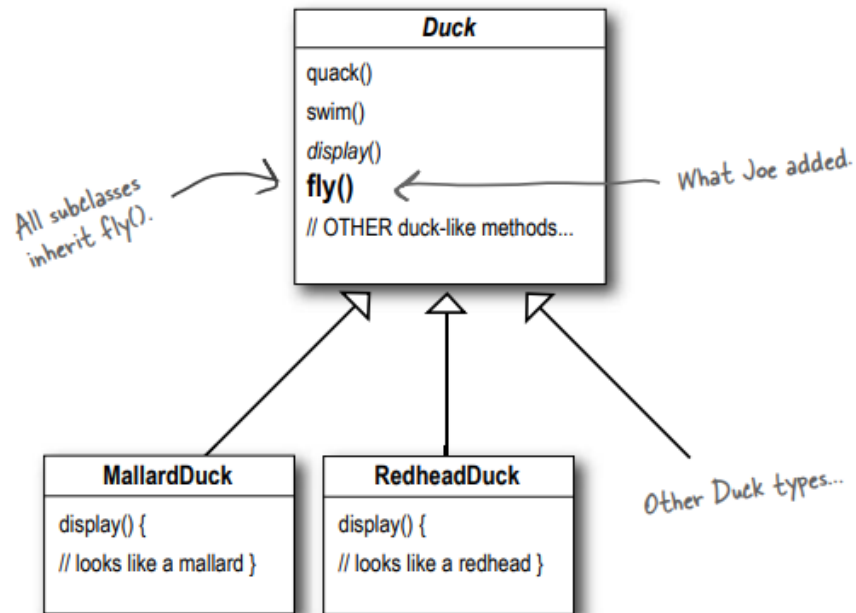
Solution focusing on reusability (Applying inheritance)

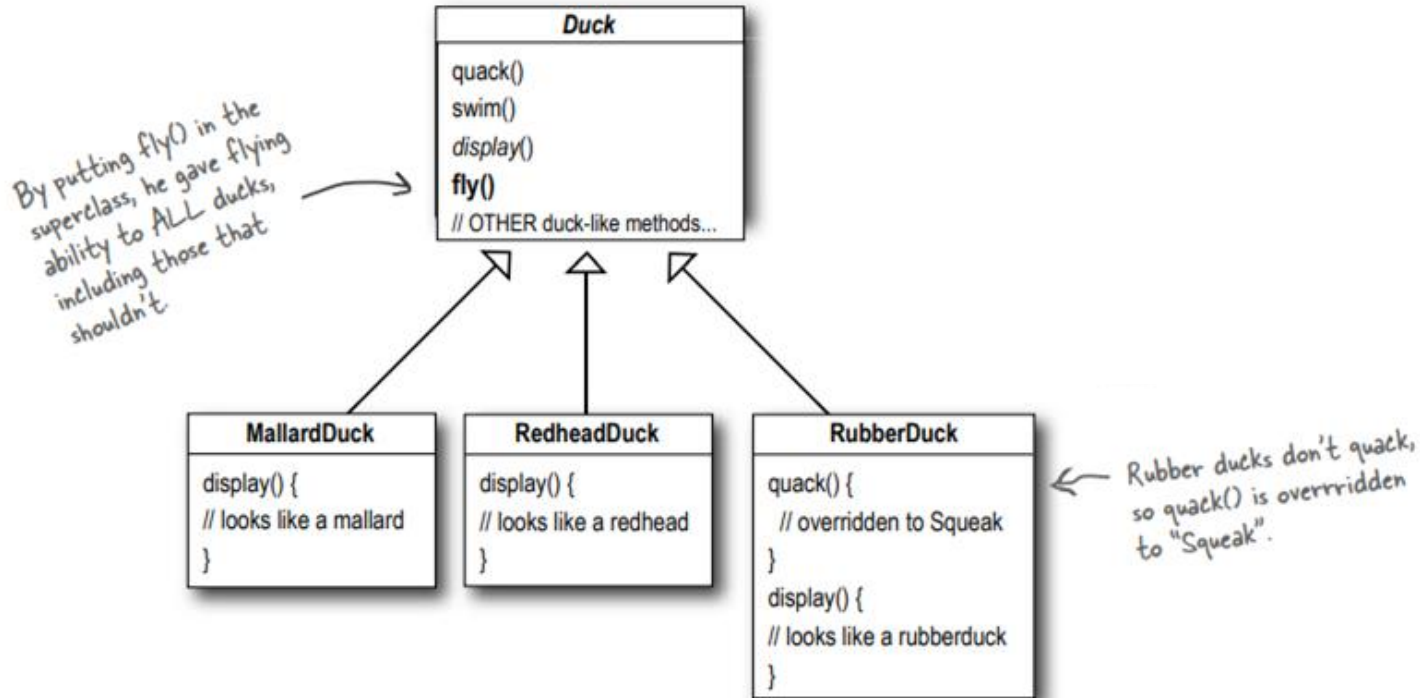


But now we need the ducks to FLY

New Challenge Appears !

The executives decided that flying ducks is just what the simulator needs to blow away the other duck sim competitors. And of course Joe's manager told them it'll be no problem for Joe to just whip something up in a week. "After all", said Joe's boss, "he's an OO programmer... *how hard can it be?*"





But it creates a problem.

Joe thinks about inheritance...

I could always just override the fly() method in rubber duck, the way I am with the quack() method...



```
RubberDuck  
quack() { // squeak}  
display() { // rubber duck }  
fly() {  
    // override to do nothing  
}
```

A possible solution

But then what happens when we add wooden decoy ducks to the program? They aren't supposed to fly or quack...



```
DecoyDuck  
quack() {  
    // override to do nothing  
}  
  
display() { // decoy duck }  
  
fly() {  
    // override to do nothing  
}
```

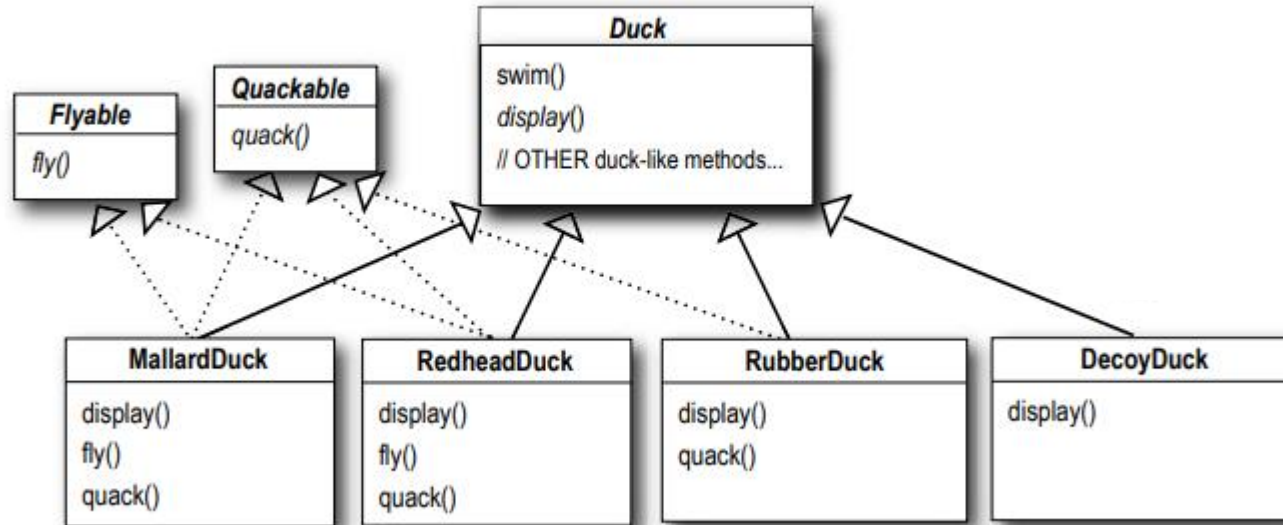
Here's another class in the hierarchy; notice that like RubberDuck, it doesn't fly, but it also doesn't quack.

But the issue persists.

Solution using Interface

Issues:

- > Lost Reusability, need to implement each interface each time
- > Brings difficulties in code maintenance



First Design Principle



Design Principle

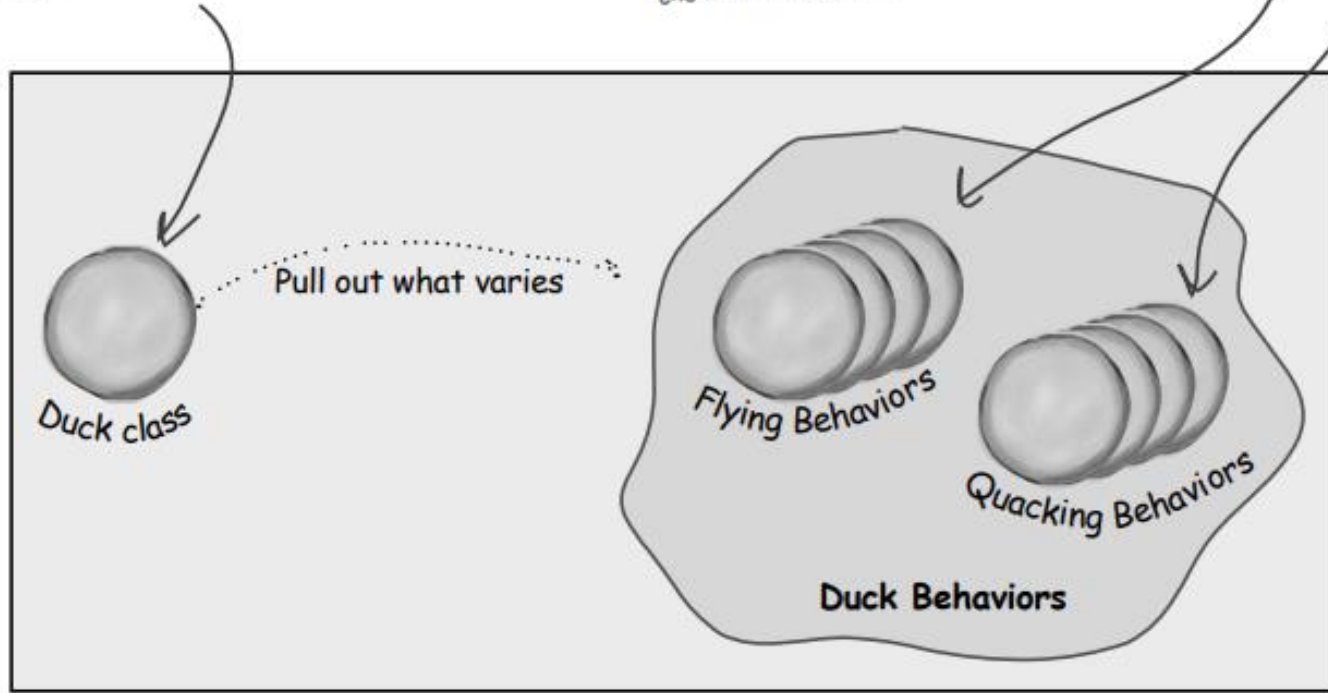
Identify the aspects of your application that vary and separate them from what stays the same.

Here's another way to think about this principle: ***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.***

The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.

Now flying and quacking each get their own set of classes.

Various behavior implementations are going to live here.



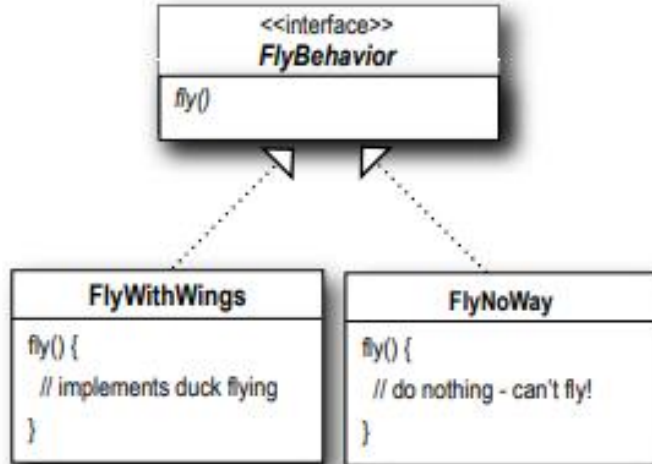
Putting flying and quacking to different classes

Second Design Principle



Design Principle

Program to an interface, not an implementation.



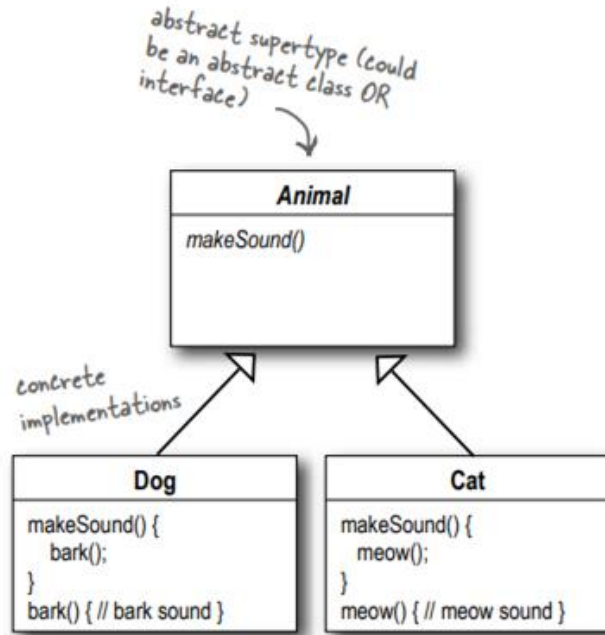
From now on, the Duck behaviors will live in a separate class—a class that implements a particular behavior interface.

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

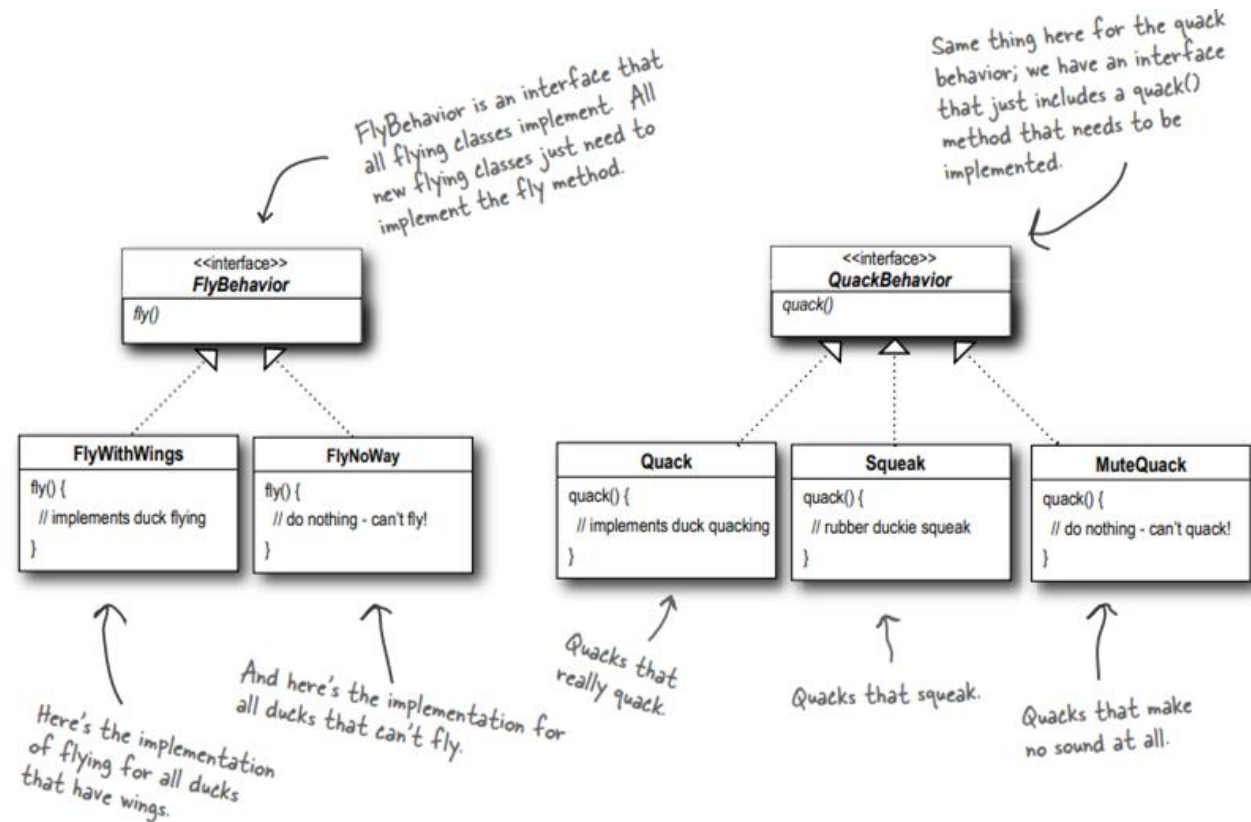
Helps to add new behaviour on the fly

Program to an interface == Program to a supertype

To implement the supertype, we can either use **Interface** or **Abstract classes**



Implementing the duck behaviour



The Core Advantage/Benefit

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

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

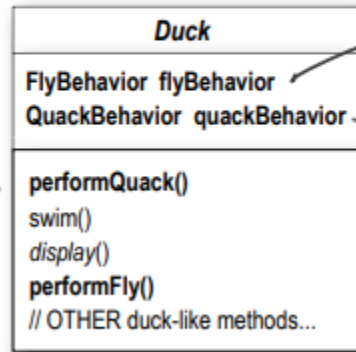
← So we get the benefit of REUSE without all the baggage that comes along with inheritance.

Integrating the duck behaviour

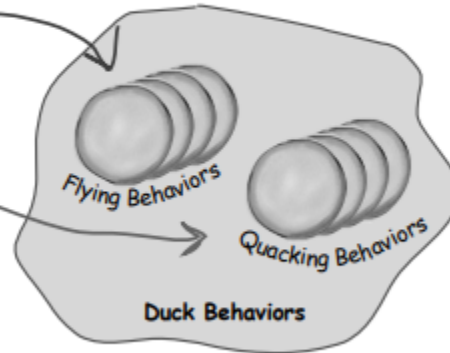
1. First we'll add two instance variables

The behavior variables are declared as the behavior INTERFACE type.

These methods replace fly() and quack().



Instance variables hold a reference to a specific behavior at runtime.



2 Now we implement performQuack():

```
public 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 the *flyBehavior* and *quackBehavior* instance variables are set

```
public class MallardDuck extends Duck {  
  
    public MallardDuck() {  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
}
```

Remember, *MallardDuck* inherits the *quackBehavior* and *flyBehavior* instance variables from class *Duck*.

A *MallardDuck* uses the *Quack* class to handle its quack, so when *performQuack* is called, the responsibility for the quack is delegated to the *Quack* object and we get a real quack.

And it uses *FlyWithWings* as its *FlyBehavior* type.

```
    public void display() {  
        System.out.println("I'm a real Mallard duck");  
    }  
}
```


Testing the Duck code - 1

```
public abstract class Duck {
```

```
    FlyBehavior flyBehavior;  
    QuackBehavior quackBehavior;  
    public Duck() {  
    }
```

← Declare two reference variables
for the behavior interface types.
All duck subclasses (in the same
package) inherit these.

```
    public abstract void display();
```

```
    public void performFly() {  
        flyBehavior.fly();  
    }
```

← Delegate to the behavior class.

```
    public void performQuack() {  
        quackBehavior.quack();  
    }
```

```
    public void swim() {  
        System.out.println("All ducks float, even decoys!");  
    }
```

```
}
```

Testing the Duck code - 2

```
public interface FlyBehavior {  
    public void fly();  
}
```

The interface that all flying
behavior classes implement.

```
public class FlyWithWings implements FlyBehavior {  
    public void fly() {  
        System.out.println("I'm flying!!");  
    }  
}
```

Flying behavior implementation
for ducks that DO fly...

```
public class FlyNoWay implements FlyBehavior {  
    public void fly() {  
        System.out.println("I can't fly");  
    }  
}
```

Flying behavior implementation
for ducks that do NOT fly (like
rubber ducks and decoy ducks).

Testing the Duck code - 3

```
public interface QuackBehavior {  
    public void quack();  
}
```

```
public class Quack implements QuackBehavior {  
    public void quack() {  
        System.out.println("Quack");  
    }  
}
```

```
public class MuteQuack implements QuackBehavior {  
    public void quack() {  
        System.out.println("<< Silence >>");  
    }  
}
```

```
public class Squeak implements QuackBehavior {  
    public void quack() {  
        System.out.println("Squeak");  
    }  
}
```

Testing the Duck code - 4

```
public class MiniDuckSimulator {  
    public static void main(String[] args) {  
        Duck mallard = new MallardDuck();  
        mallard.performQuack();  
        mallard.performFly();  
    }  
}
```

This calls the *MallardDuck*'s inherited *performQuack()* method, which then delegates to the object's *QuackBehavior* (i.e. calls *quack()* on the duck's inherited *quackBehavior* reference).

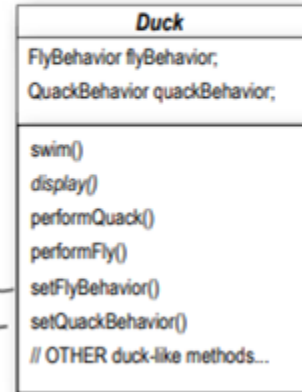
Then we do the same thing with *MallardDuck*'s inherited *performFly()* method.

Setting Behaviour Dynamically - 1

We can call these methods anytime we want to change the behavior of a duck on the fly.

❶ Add two new methods to the Duck class:

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



2 Make a new Duck type (ModelDuck.java).

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

Our model duck begins life grounded...
without a way to fly.

3 Make a new FlyBehavior type (FlyRocketPowered.java).

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

That's okay, we're creating a
rocket powered flying behavior.



4 Change the test class (MiniDuckSimulator.java), add the ModelDuck, and make the ModelDuck rocket-enabled.


```
public class MiniDuckSimulator {  
    public static void main(String[] args) {  
        Duck mallard = new MallardDuck();  
        mallard.performQuack();  
        mallard.performFly();  
  
        Duck model = new ModelDuck();  
        model.performFly();  
        model.setFlyBehavior(new FlyRocketPowered());  
        model.performFly();  
    }  
}
```

If it worked, the model duck dynamically changed its flying behavior! You can't do THAT if the implementation lives inside the duck class.

5 Run it!

```
File Edit Window Help Yabadabadoo  
%java MiniDuckSimulator  
Quack  
I'm flying!!  
I can't fly  
I'm flying with a rocket
```

before



The first call to `performFly()` delegates to the `flyBehavior` object set in the `ModelDuck`'s constructor, which is a `FlyNoWay` instance.

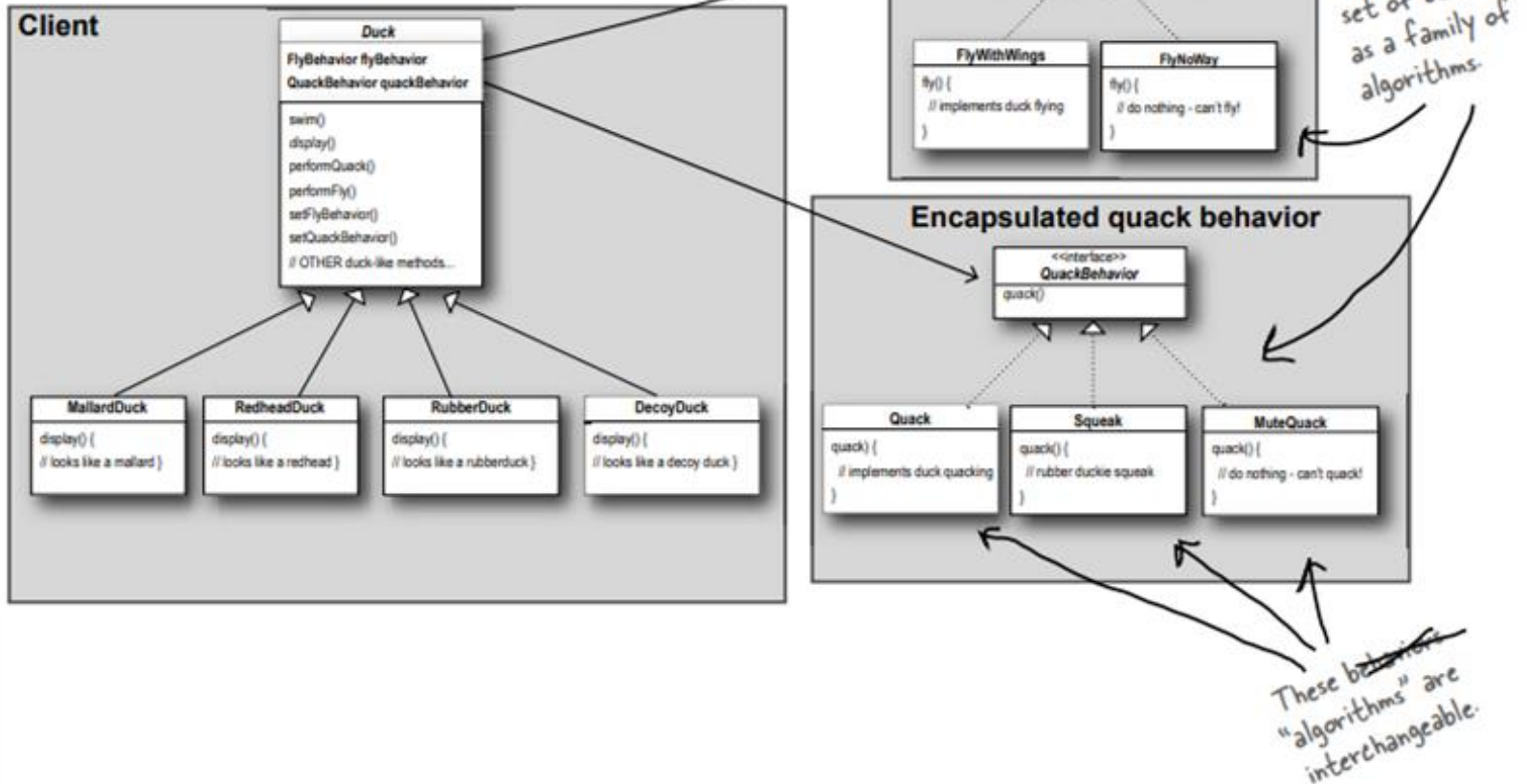
This invokes the model's inherited behavior setter method, and...voila! The model suddenly has rocket-powered flying capability!

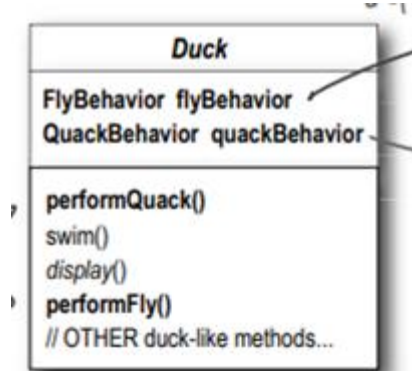
after



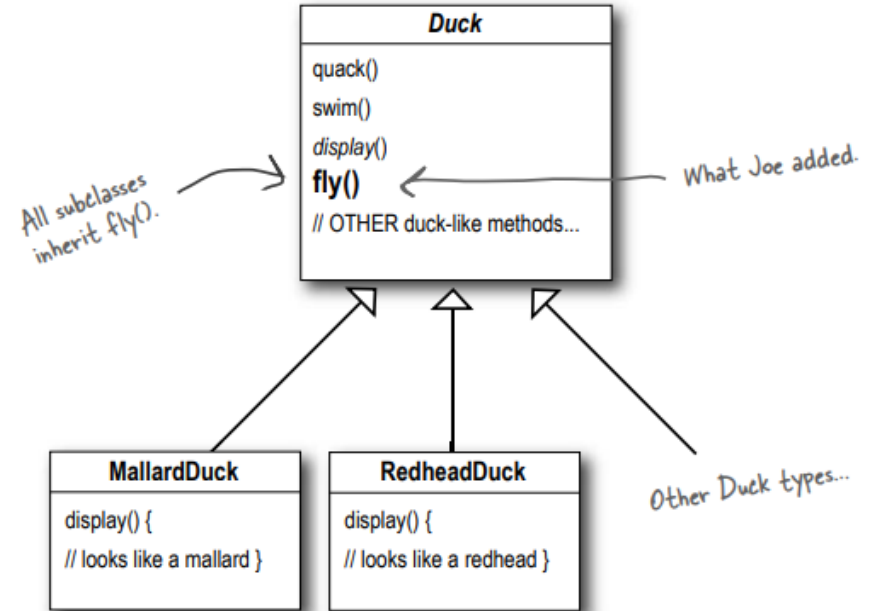
Client makes use of an encapsulated family of algorithms for both flying and quacking.

The Big Picture





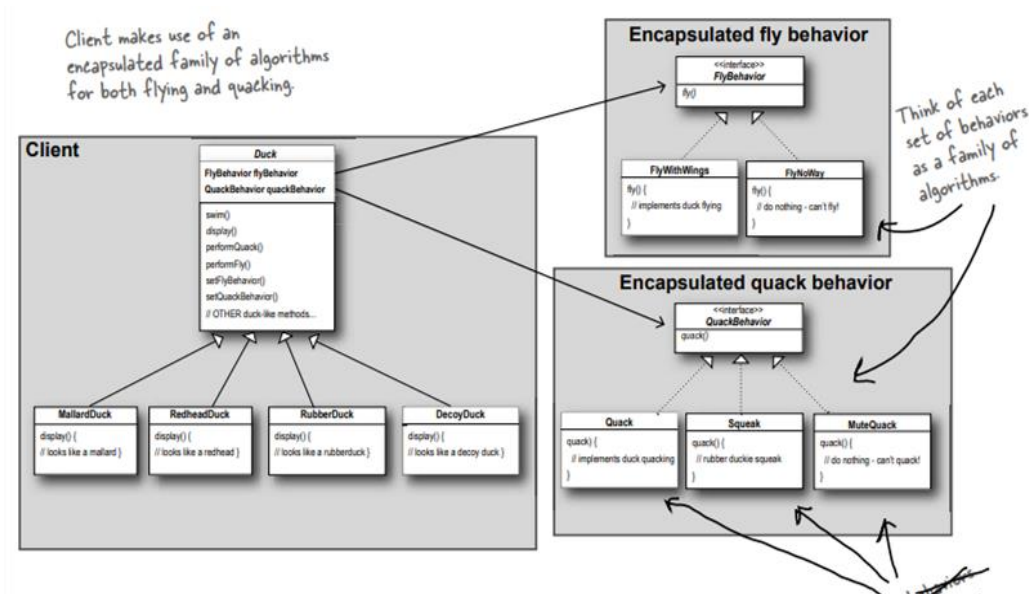
Has A



Is A

Main Advantage: Changing Behaviour in Runtime

Formal Definition of the Design Pattern

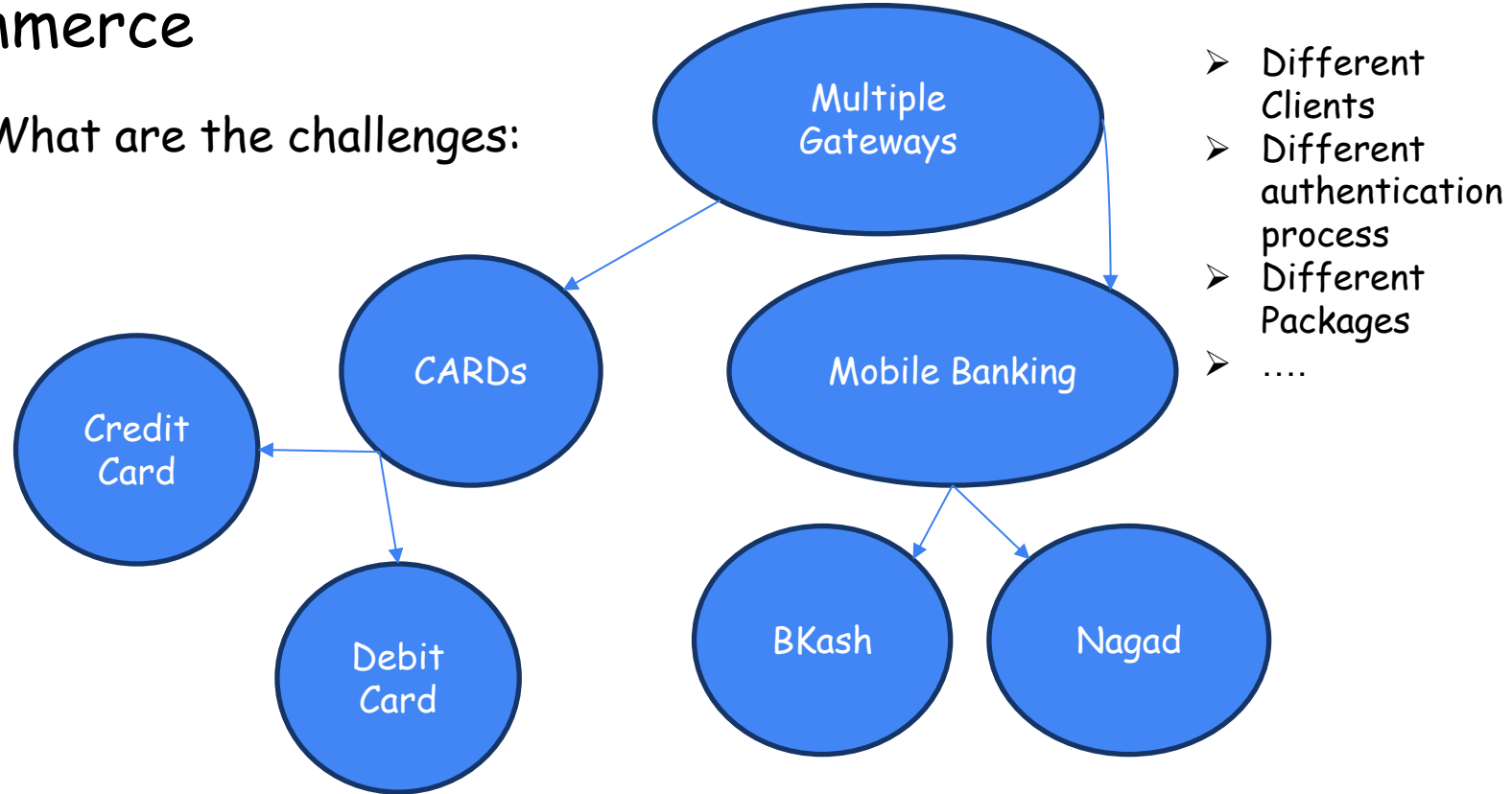


The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Design Pattern, Helps?

Let's say we are going to build a gateway for E-Commerce

- What are the challenges:



Storage vs Communication among modules

- Database/Storage: Is used to preserve the data only
- But **Database connection** ? [Might require Singleton Pattern for single access point to reduce memory usage]
- How **Functions should communicate** ? [Objects -> Application of Abstraction], e.g: Bkash System, Nagad System everything under MobileClient System. *Objects with their hidden logic will propagate the feature of an application.*
- Does any **feature corresponds to the specifics of a design pattern**? If does, apply it.
- Consider these logics as **Algorithms**: That would vary as per the context.

Thank You