STRATEGY DESSIGN PATTERN

WELCOME TO OUR GROUP

Hao Trinh The

10421017

Huy Vo Vuong Bao

10421021

TABLE OF CONTENT

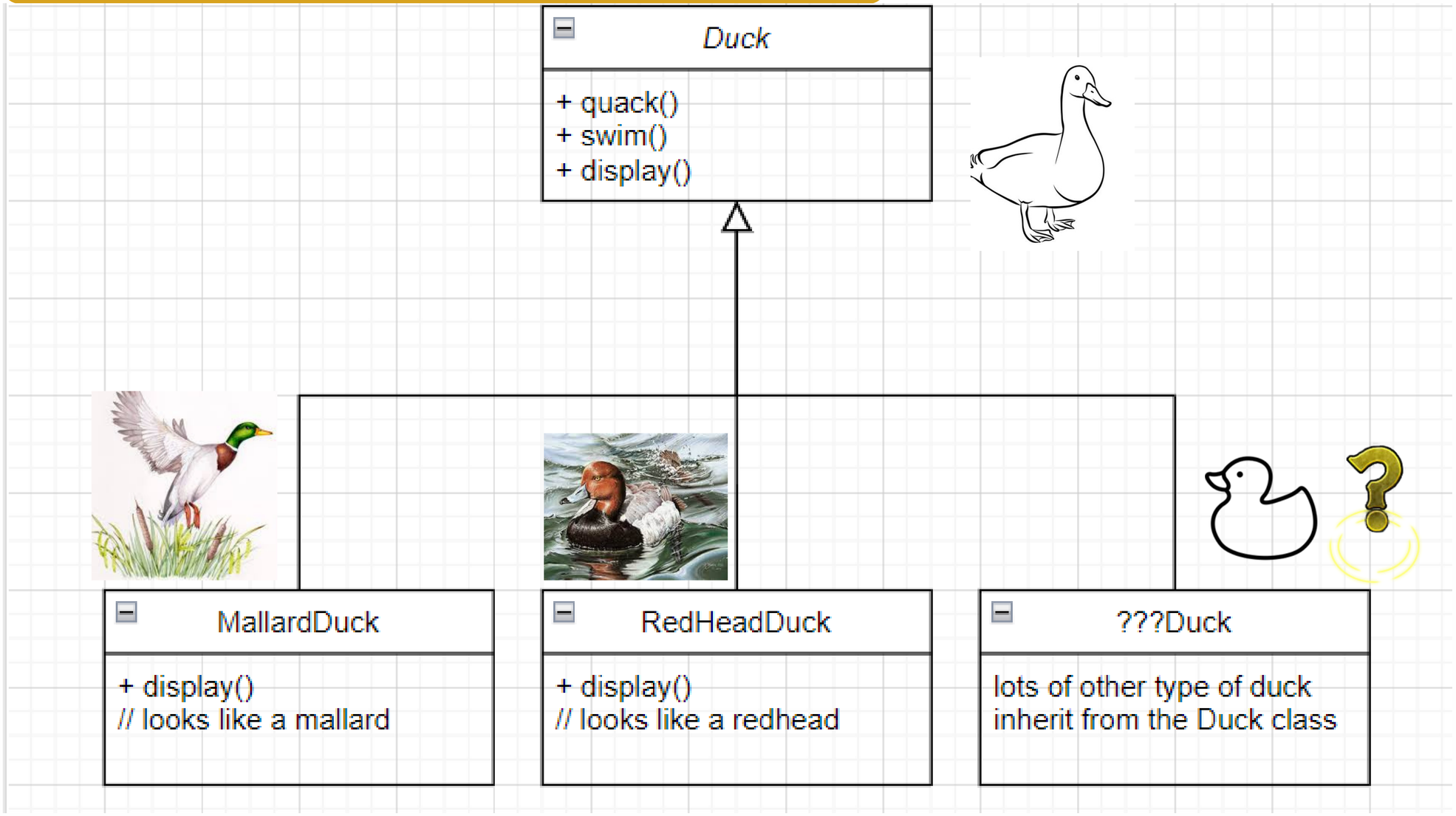
SimUDuck Application

A SIMPLE SIM U DUCK APP

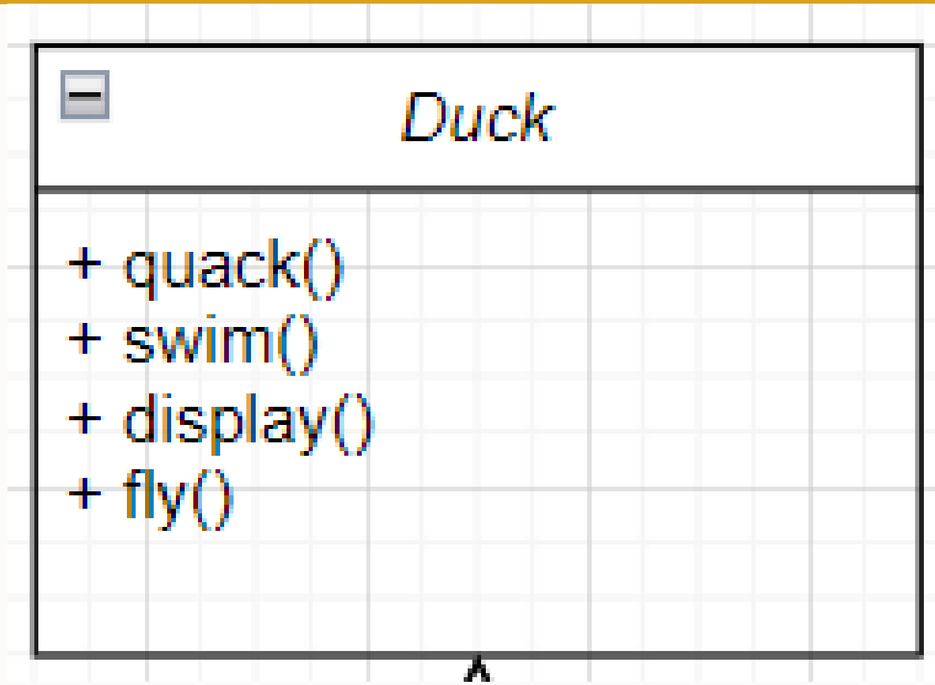
Display a wide range of duck species.

- Duck behaviors
- Using fundamental OO principles, one Duck superclass will be created, from which all other duck types will be inherited.

SIM U DUCK DIAGRAM

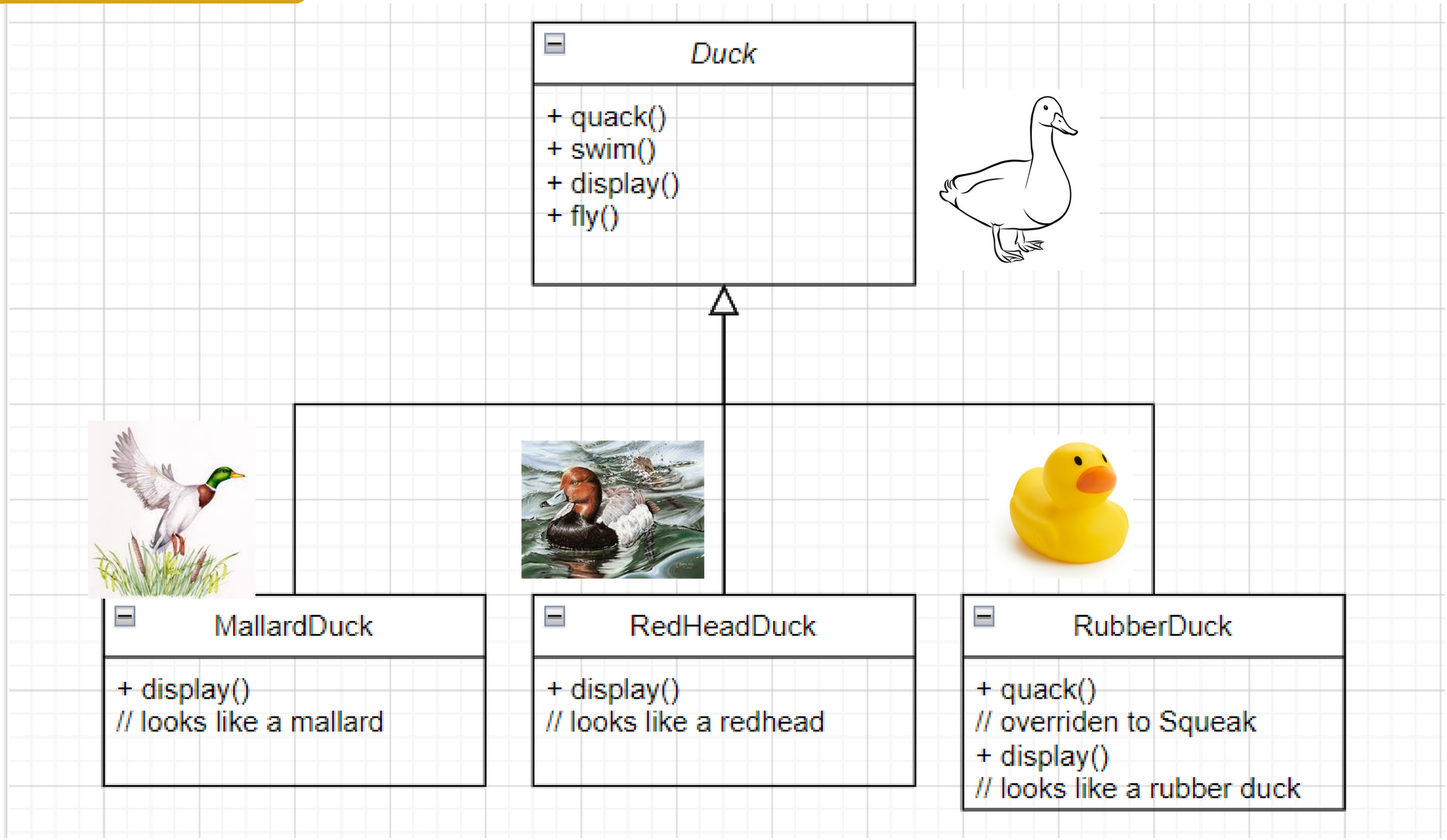


NOW WE NEED THE DUCKS TO FLY




My Boss
Now I need the ducks to fly
Me
So I just need to add a fly() method in the Duck classes, right?
My Boss
Me

NOT ALL SUBCLASSES OF DUCK CAN FLY



Arised Problem

It doesn't turn out well when it comes to maintenance.

INHERITANCE OVERRIDING

RubberDuck

+ quack()
// overriden to Squeak
 	
+ fly()
// overridden to do nothing
 	
+ display()
// looks like a rubber duck

DecoyDuck

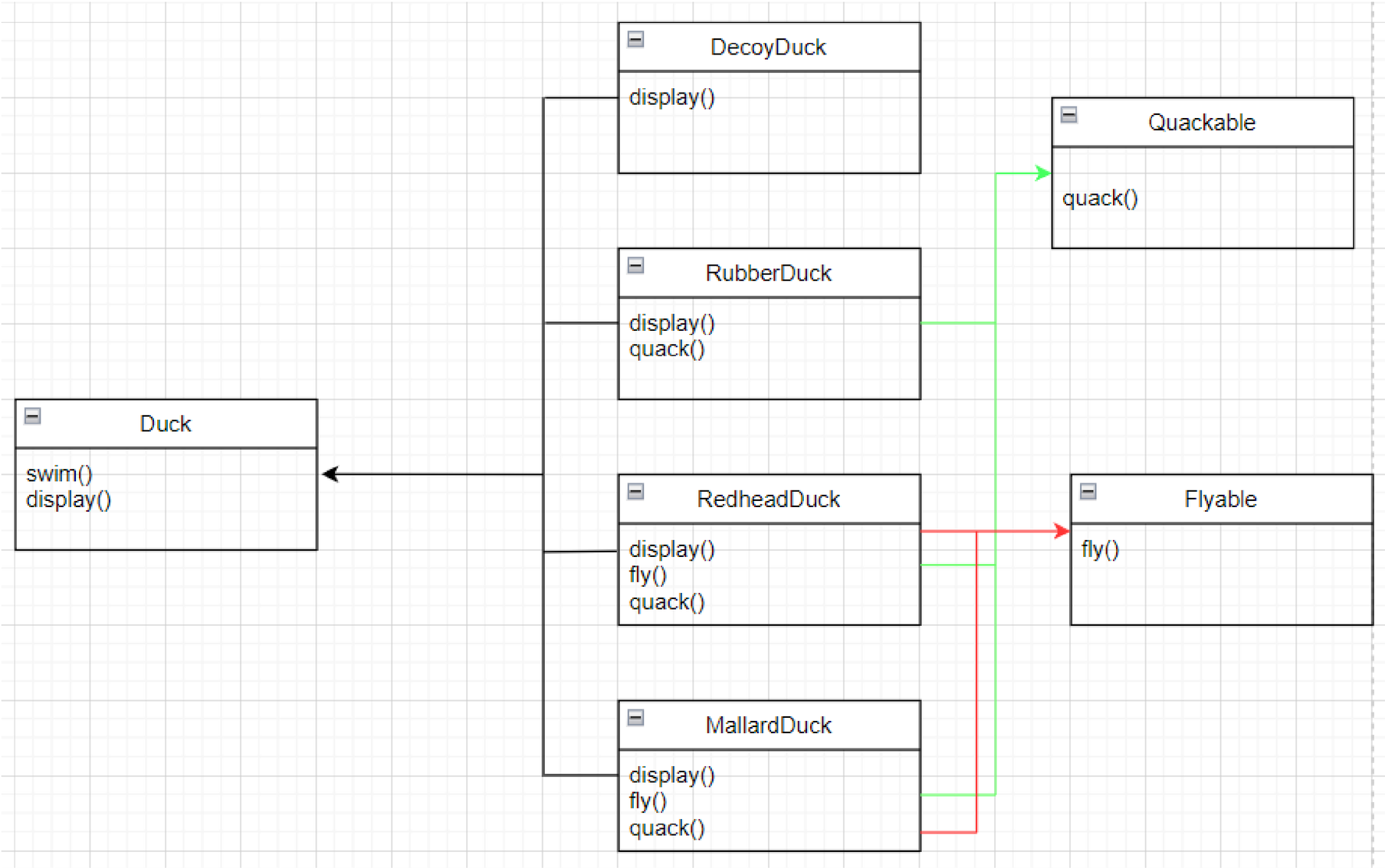
+ quack()
// overriden to do nothing
 	
+ fly()
// overriden to do nothing
 	
+ display()
// looks like a rubber duck

So we need to override fly() a& quack() for ever new Duck subclass? Is that good solution????

HOW ABOUT USING INTERFACE?

- Take **fly()** out of the Duck superclass, and make a **Flyable()** interface with a fly() method.
- Only the ducks that are supposed to fly will implement that interface and have **fly()** method.
- The same thing happens with **Quackable()**

WHAT DO YOU THINK ABOUT THIS DIAGRAM?



IT'S A NIGHTMARE!

DUPLICATED CODE!

- Consider the **100** flying Duck subclasses when you may need to slightly alter your flying style.
- When we need to **alter a piece** of software, how can we do it with the **least amount of damage** on the existing code?

Strategy Design Pattern

DEFINITION

Strategy design pattern is one of the behavioral design patterns. Strategy pattern is used when we have multiple algorithms for a specific task and the client decides the actual implementation to be used at runtime

CHANGE

One constant in software development

ELIMINATE ALL THE PROBLEMS

Problem

The behaviors of the ducks are always changing, and not all subclasses should have those traits.

Solution

Separate the elements of your application that change from those that don't by identifying what changes.

Result

More flexibility in your systems and fewer unwanted implications from code modifications.

1

Seperating what changes from what stays the same

2

Design the Duck behaviors

3

Implementing the Duck behaviors

4

Integrating the Duck behavior

5

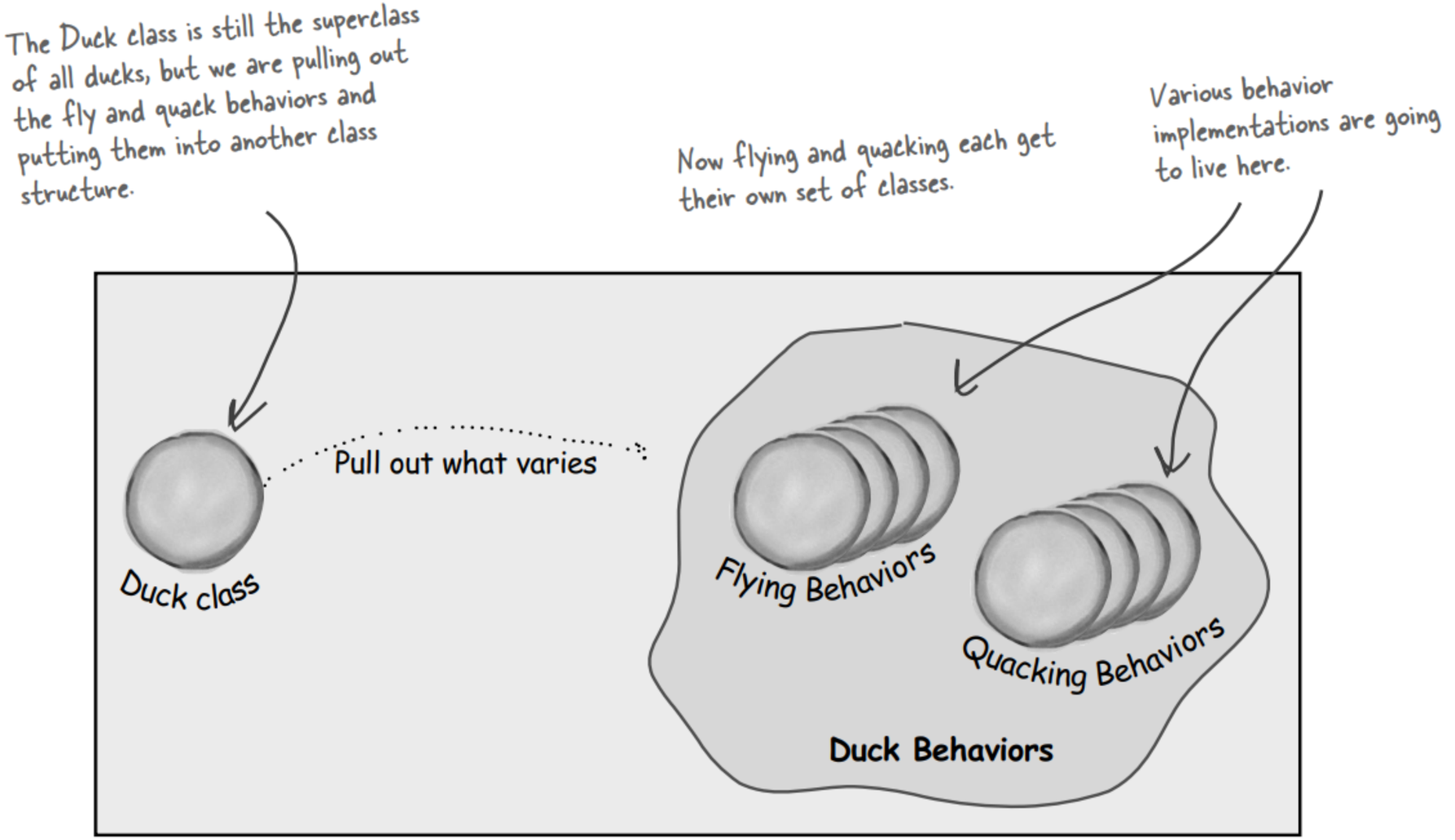
Testing the Duck code

6

Setting behaviors dynamically

THE PROCESS

1.SEPARATE WHAT CHANGES FROM THE REST



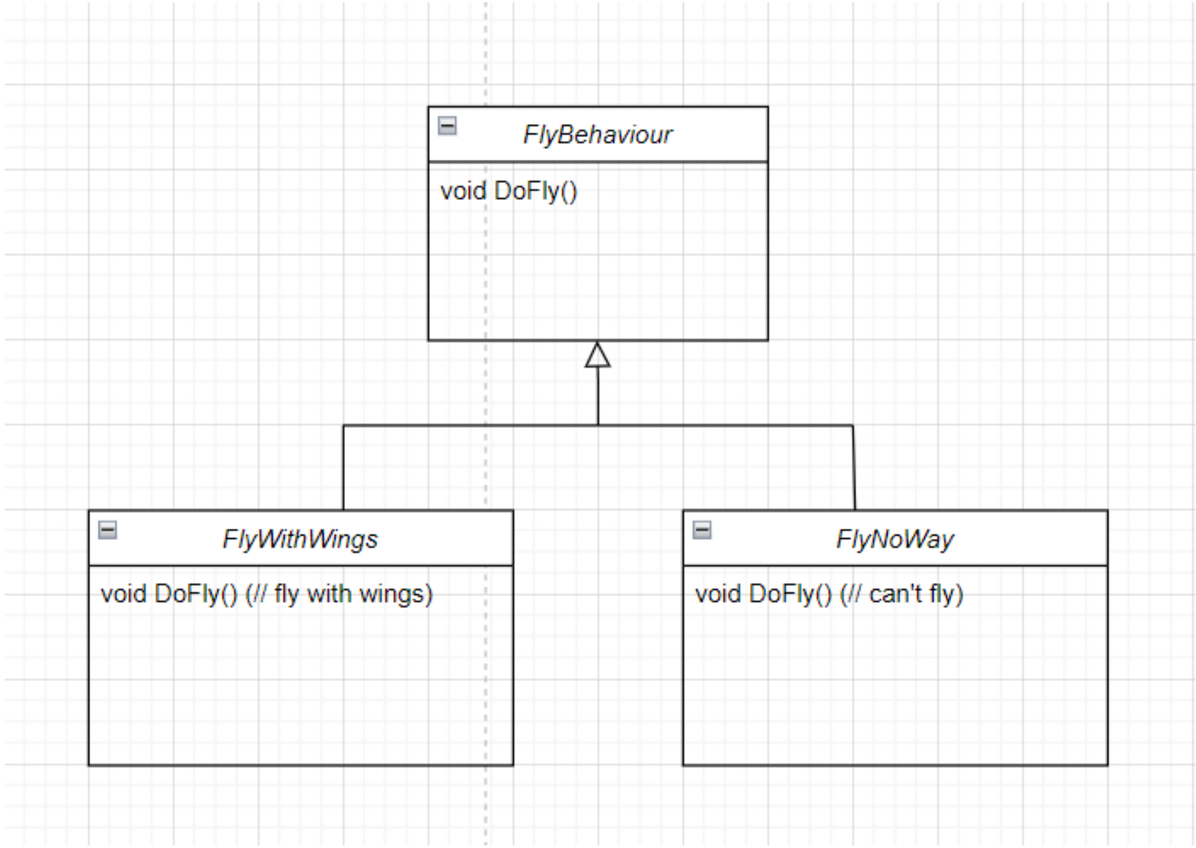
2.DESIGNING THE DUCK BEHAVIORS

Use an interface to represent each behavior.

Each implementation of a behavior will implement one of those interfaces

Design Principle

Program to an interface, not an implementation



LET'S USE THE INTERFACE AGAIN

```
// interface for fly methods
class FlyBehavior {
public:
    virtual void DoFly(){};
};
```

This time all duck classes will implement the **FlyBehaviour** interface.

The interface will contain the DoFly() prototype.

3.IMPLEMENTING THE DUCK BEHAVIORS

The **DoFly()** function will be implemented by classes tha implement the interface.

Benefits

- These behaviours can be **REUSED**.
- No need of modifying existing behaviour classes or Duck classes **when adding new behaviors**.

```
class FlyWithWings : public FlyBehavior {
public:
    void DoFly() {
        cout << "I'm flying with wings!" << endl;
    }
};

class FlyNoWay : public FlyBehavior {
public:
    void DoFly() override{
        cout << "I can't fly!" << endl;
    }
};
```

3.IMPLEMENTING THE DUCK BEHAVIORS

All we care is it knows how to perform actions

Integrating the Duck Behaviour

- Create a pointer for each behaviour interface type variable.
- **PerformFly()** and **PerformQuack()** will carry out the action of the behaviour.

```
class Duck {
protected:
    string name;
public:
    QuackBehavior *quackBehavior;
    FlyBehavior *flyBehavior;

    string getName() {
        return name;
    }

    void PerformFly() {
        flyBehavior->DoFly();
    }

    void PerformQuack() {
        quackBehavior->DoQuack();
    }

    virtual void Display() = 0;
};
```

MORE INTEGRATIONS

Mallard Duck will fly with wings and quack

Decoy Duck can do nothing

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

    void Display(){
        cout<<"I'm a Mallard Duck!"<<endl;
    }
};
class DecoyDuck : public Duck{
public:
    DecoyDuck(){
        quackBehavior = new CannotQuack;
        flyBehavior = new FlyNoWay;
    }

    void Display(){
        cout<<"I'm a Decoy Duck!"<<endl;
    }
};
```

TIME TO CREATE SOME DUCKS!

```
int main(int argc, char *argv[]){
    Duck* mallard = new MallardDuck();
    mallard->Display();
    mallard->PerformFly();
    mallard->PerformQuack();
    cout << endl;

    Duck* decoy = new DecoyDuck();
    decoy->Display();
    decoy->PerformFly();
    decoy->PerformQuack();
    cout << endl;
}
```

```
I'm a Mallard Duck!
I'm flying with wings!
Quack!

I'm a Decoy Duck!
I can't fly!
I can't quack!
```

NOW, WE SET THE BEHAVIOR DYNAMICALLY

- In this duck class, we can add the following method:

```
void SetFlyBehavior(FlyBehavior *flybehavior) {
    flyBehavior = flybehavior;
}
```

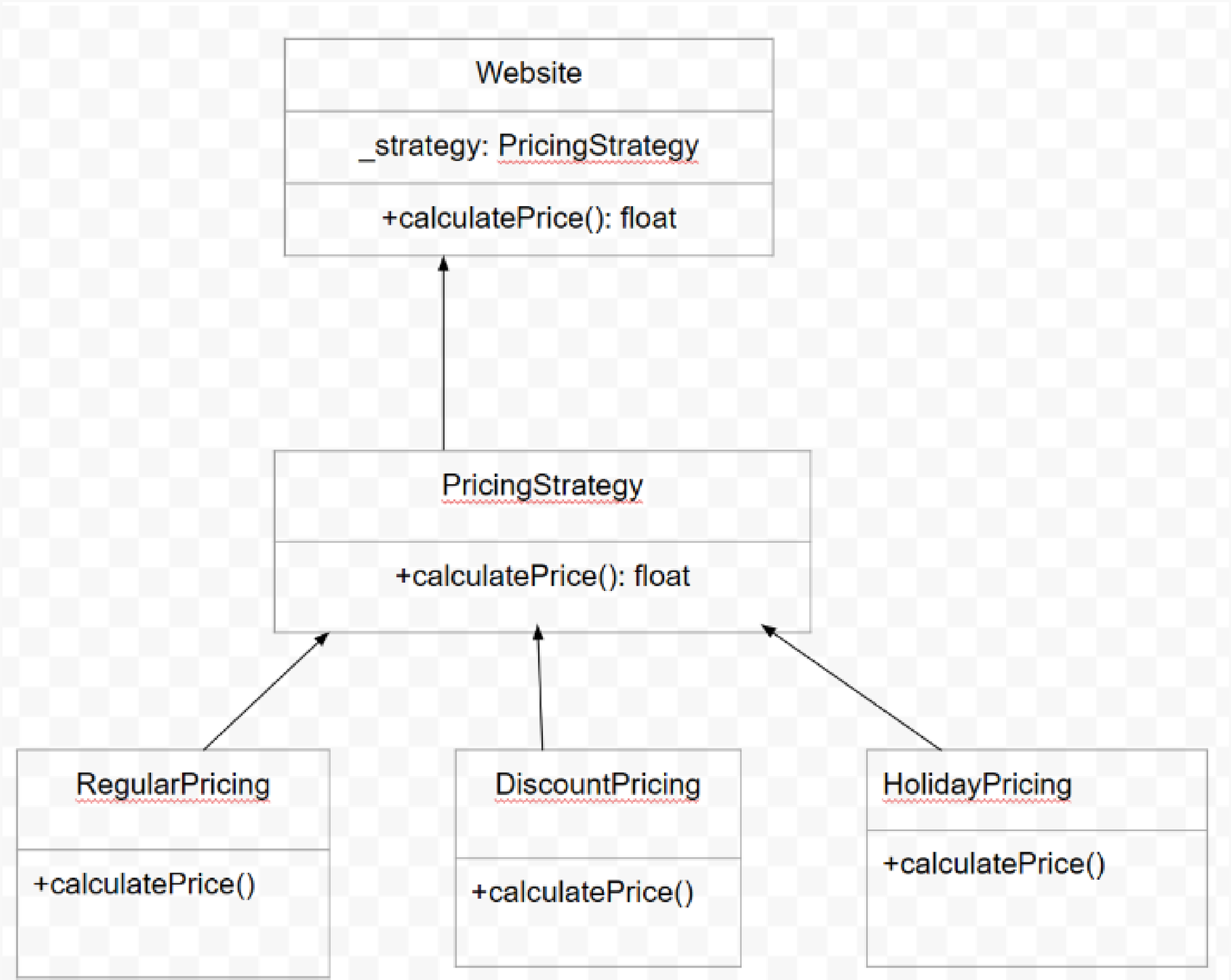
- In the main funcion:

```
mallard->PerformFly();
mallard->PerformQuack();
mallard->SetFlyBehavior(new FlyNoWay);
mallard->PerformFly();
```

Output:

```
I'm flying with wings!
Quack!
I can't fly!
```

REAL LIFE APPLICATION



```cpp
#include <iostream>
#include <string>

using namespace std;

class PricingStrategy {
public:
    virtual float calculatePrice(float basePrice) = 0;
};

class RegularPricing : public PricingStrategy {
public:
    float calculatePrice(float basePrice) override {
        return basePrice;
    }
};

class DiscountPricing : public PricingStrategy {
public:
    float calculatePrice(float basePrice) override {
        return basePrice * 0.9f; // apply 10% discount
    }
};

class HolidayPricing : public PricingStrategy {
public:
    float calculatePrice(float basePrice) override {
        return basePrice * 0.8f; // apply 20% discount
    }
};
```

```cpp
class Website {
private:
    PricingStrategy* strategy;

public:
    Website(PricingStrategy* strategy) : strategy(strategy) {}

    void setStrategy(PricingStrategy* strategy) {
        this->strategy = strategy;
    }

    float calculatePrice(float basePrice) {
        return strategy->calculatePrice(basePrice);
    }
};

int main() {
    Website website(new RegularPricing());

    float price1 = website.calculatePrice(100.0f);
    std::cout << "Regular price: " << price1 << std::endl;

    website.setStrategy(new DiscountPricing());
    float price2 = website.calculatePrice(100.0f);
    std::cout << "Discounted price: " << price2 << std::endl;

    website.setStrategy(new HolidayPricing());
    float price3 = website.calculatePrice(100.0f);
    std::cout << "Holiday price: " << price3 << std::endl;

    return 0;
}
```

THANK YOU FOR LISTENING