SOFTWARE ENGINEERING INSTITUTE
华东师范大学软件学院

Object oriented Analysis &Design
面向对象分析与设计

Lecture_14 Design Pattern Principles

主讲: 陈小红

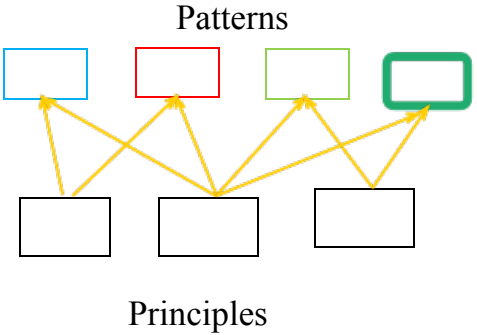
日期:

32种设计模式

- Factory
- Façade
- Observer
- Adaptor
- Singleton
- Command
- Composite
- Strategy
- 内容主要来自如下几本书：
- 《设计模式》《大话设计模式》《深入浅出设计模式》

Design Pattern & Design Principle

- So many patterns
- 3 Principles
- Let's get started!

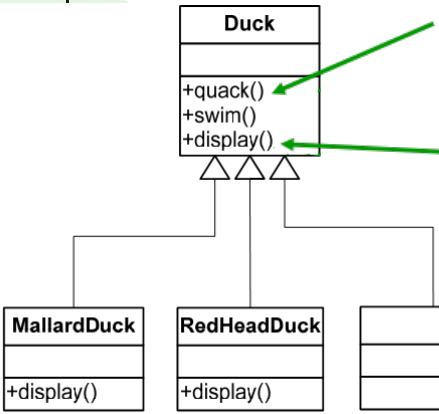


The Job

- Joe works at a company that produces a simulation game called SimUDuck. He is an OO Programmer and his duty is to implement the necessary functionality for the game.

- The game should have the following specifications:
  - A variety of different ducks should be integrated into the game
  - The ducks should swim
  - The duck should quack

A First Design for the Duck Simulator Game



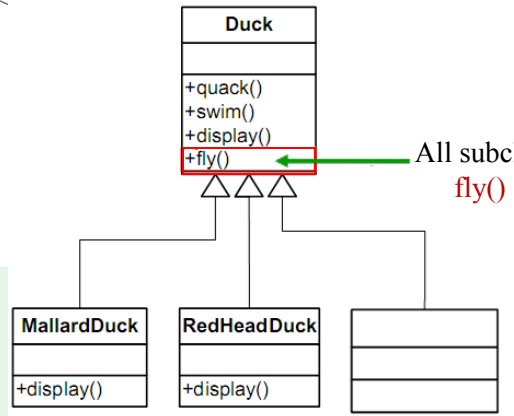
All ducks quack() and swim(). The super-class takes care of the implementation

The display() method is abstract since all the duck subtypes look different

Each duck subtype is responsible for implementing its own display() behavior for how it looks on the screen

Lots of other types of ducks inherit from the Duck type

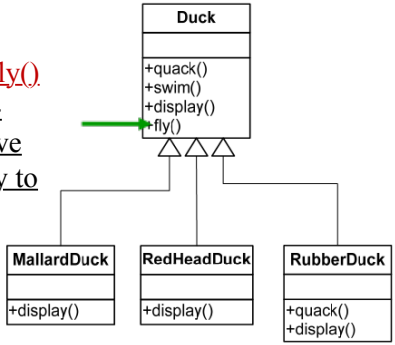
Ducks That Fly



But Something Went Wrong

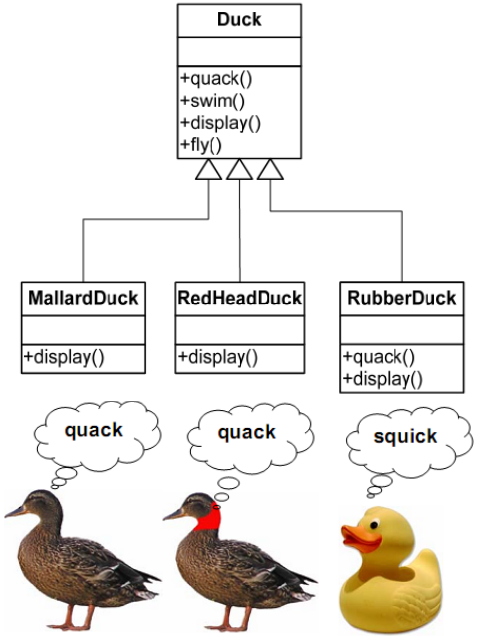
Joe, I'm at the shareholder's meeting. They just gave a demo and there were rubber duckies flying around the screen. Is this a joke or what?

By putting fly() in the super-class Joe gave flying ability to all ducks



OK, so there's a slight flaw in my design. I don't see why they can't just call it a "feature". It's kind of cute

Inheritance at Work

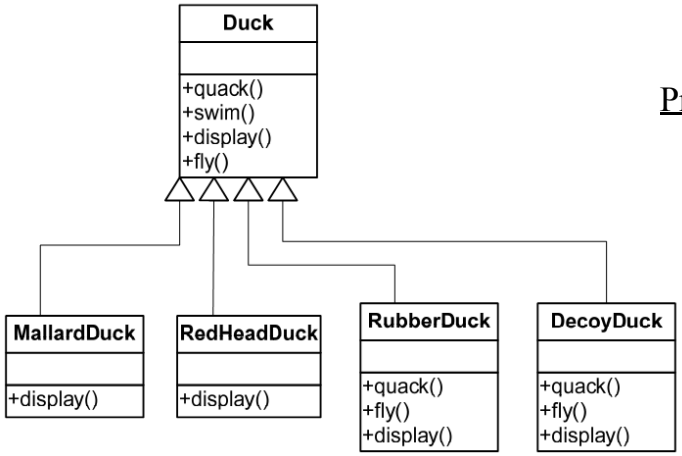



```
void Duck::quack(){
   cout << "quack, quack" << endl;
}

void RubberDuck::quack(){
   cout << "squick, squick" << endl;
}
```

We can override the **`fly()`** method in the rubber duck in a similar way that we override the **`quack()`** method

```
void Duck::fly(){
  // fly implementation
}

void RubberDuck::fly(){
  // do nothing
}
```

Yet Another Duck is Added to the Application



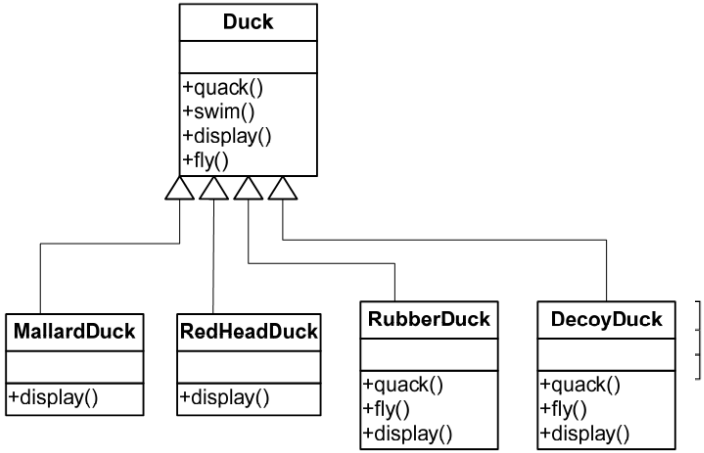
Problem?

```
void DecoyDuck::quack(){
  // do nothing;
}

void DecoyDuck::fly(){
  // do nothing
}
```

How About an Interface

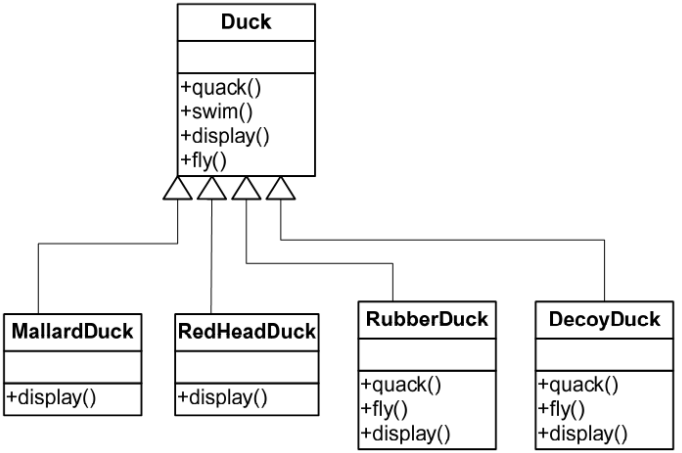
I don't think so!

We can take the fly() out of the Duck superclass and make a Flyable interface with a method fly(). Each duck that is supposed to fly will implement that interface



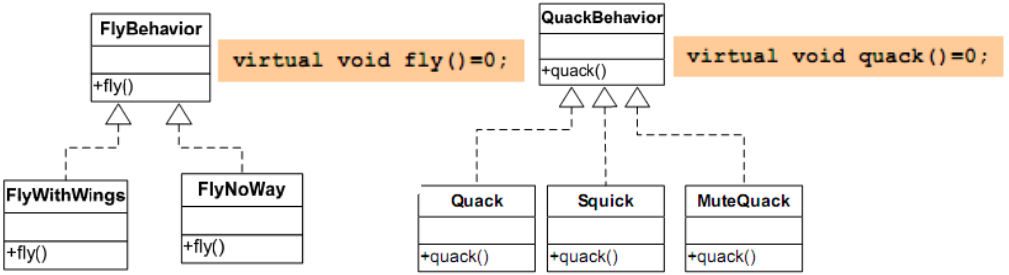
Encapsulate that vary

What is changing?



Embracing Change in Ducks

- fly() and quack() are the parts that vary
- We create a new set of classes to represent each behavior




```
void FlyWithWings::fly(){
  cout << "I'm flying!" << endl;
};
```

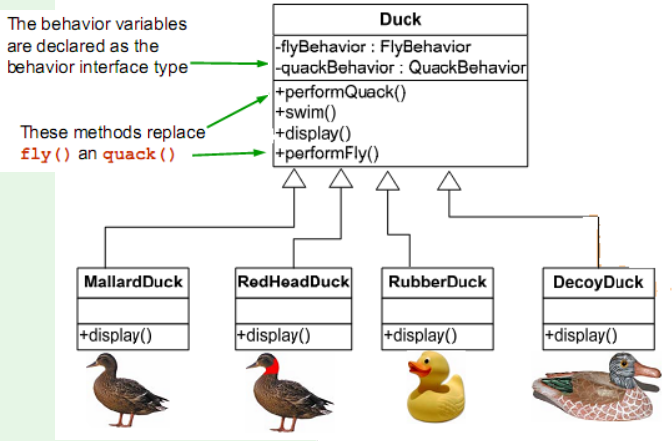
```
void Quack::quack(){
  cout << "Quack" << endl;
};
```

```
Void Squick(): :quack{
   cout<<"Squick"<<endl;
};
```

```
void FlyNoWay::fly(){
  cout << "I can't fly." << endl;
};
```

```
void MuteQuack::quack(){
   cout << "....." << endl;
};
```

Integrating the Duck Behavior



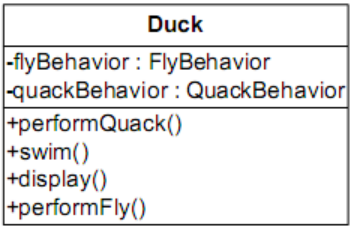

```
class Duck{
public:
  FlyBehavior *flyBehavior;
  QuackBehavior *quackBehavior;
  ...
  void performFly();
  void performQuack();
  ...
};
```

```
void Duck::performFly(){
  flyBehavior->fly();
}
void Duck::performQuack(){
   quackBehavior->quack();
```

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

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

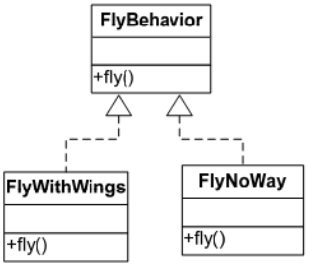
Design Principle Ahead



Each Duck **HAS A** FlyingBehavior and a QuackBehavior to which it delegates flying an quacking

Composition

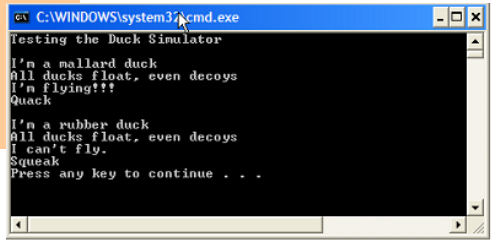
Instead of inheriting behavior, the duck get their behavior by being composed with the right behavior object



Testing the Duck Simulator

```
int main(){
  cout << "Testing the Duck Simulator"
  << endl << endl;

  Duck *mallard = new MallardDuck();
  mallard->display();
  mallard->swim();
  mallard->performFly();
  mallard->performQuack();

  cout << endl;

  Duck *rubberduck = new RubberDuck();
  rubberduck->display();
  rubberduck->swim();
  rubberduck->performFly();
  rubberduck->performQuack();

  return 0;
}
```

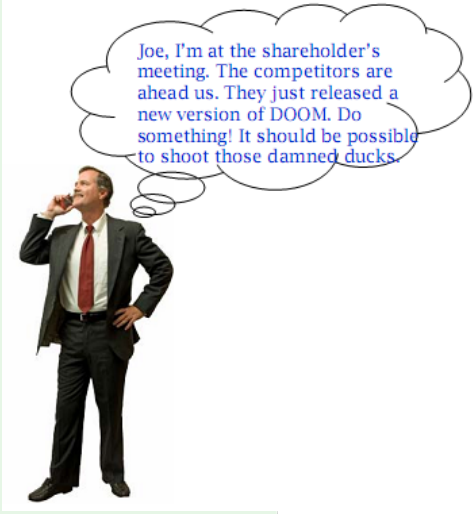
The mallard duck inherited **performQuack()** method which delegates to the object **QuackBehavior** (calls **quack()**) on the duck's inherited **quackBehavior** reference

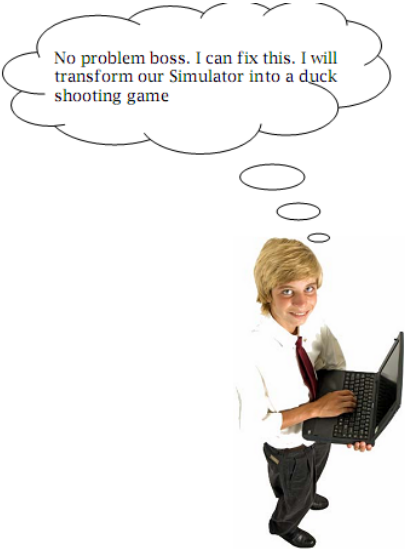


Design Principle 2

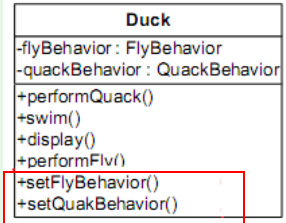
Program to an interface not to an implementation

Shooting ducks dynamically



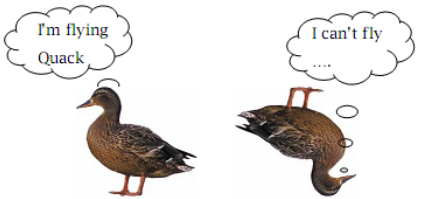


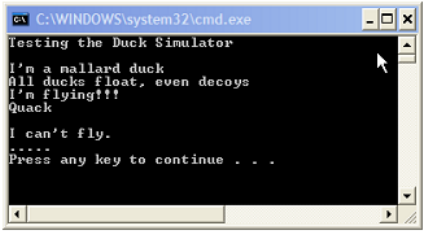
Shooting Ducks Dynamically



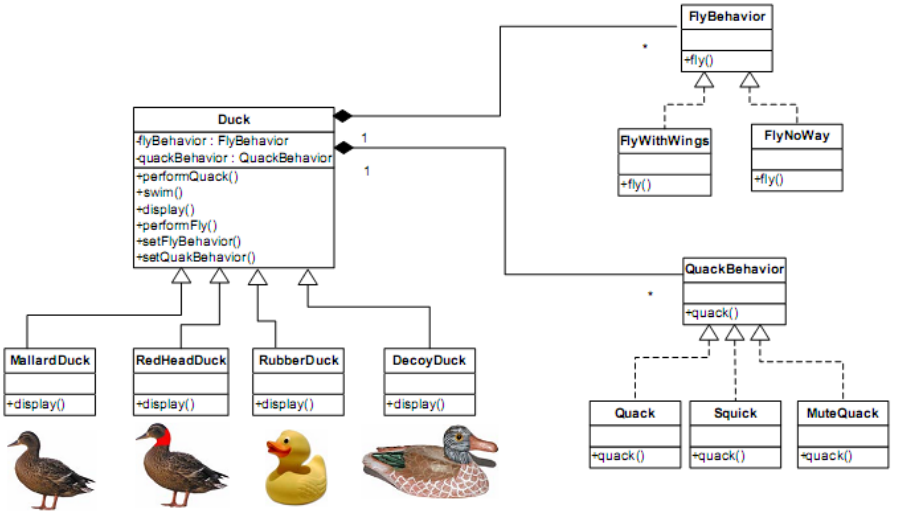

```
void Duck::setFlyBehavior(FlyBehavior *fb){
  flyBehavior = fb;
}
void Duck::setQuackBehavior(QuackBehavior *qb){
  quackBehavior = qb;
}
```

```
int main(){

  Duck *mallard = new MallardDuck();
  mallard->display();
  mallard->swim();
  mallard->performFly();
  mallard->performQuack();

  cout << endl;

  mallard->setFlyBehavior(new FlyNoWay());
  mallard->setQuackBehavior(new MuteQuack());
  mallard->performFly();
  mallard->performQuack();

  return 0;
}
```





The Big Picture

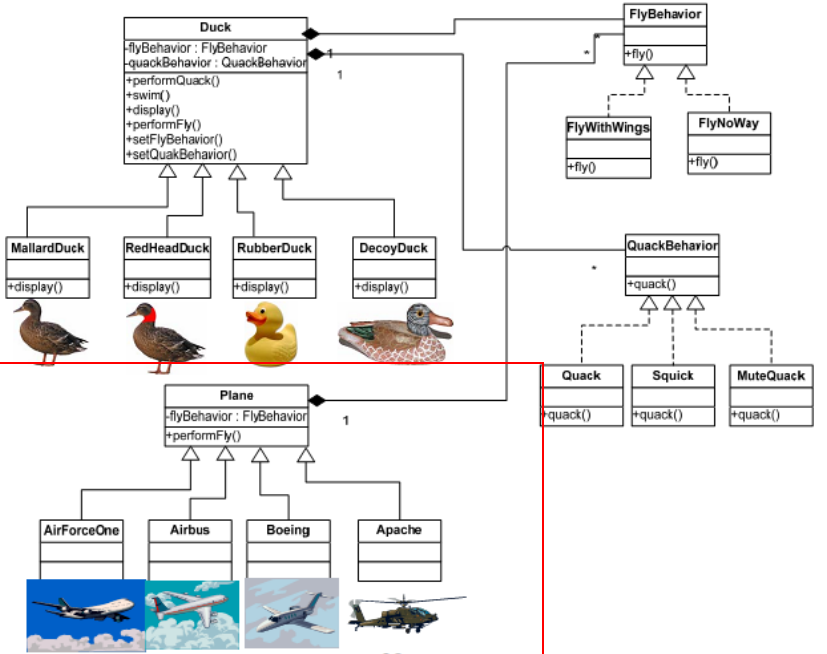


Design Principle 3

Favor Composition over Inheritance

Yet another change

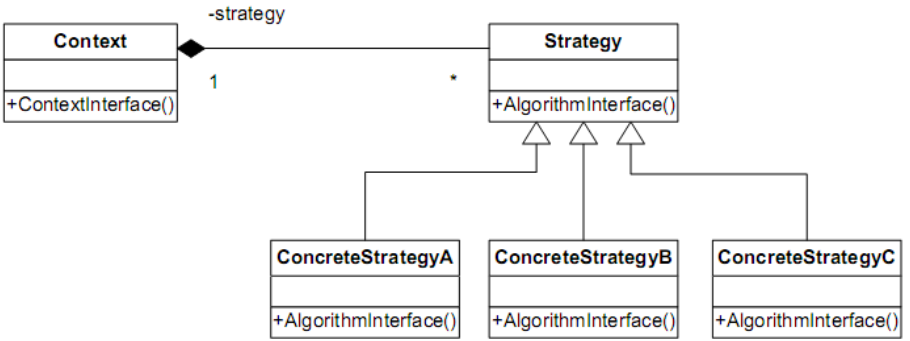
Behavior Reuse



Congratulations

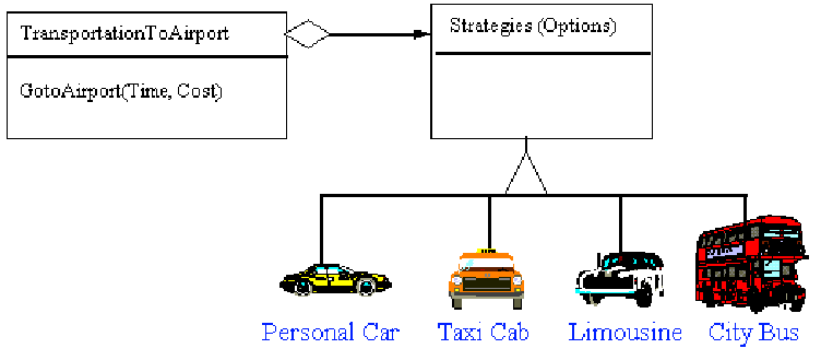
- This is a new pattern called STRATEGE!

Strategy Pattern Diagram



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

Strategy –Non Software Example



Conclusion

- Design principles
  - Encapsulate that vary.
  - Program to an interface not to an implementation.
  - Favor composition over inheritance.
- Design pattern
  - Strategy

Embracing Change

- In SOFTWRRE projects you can count on one thing that is constant:

  CHANGE

- Solution
  - Deal with it
    - Make CHANGE part of your design

思考题

- 如何解决商家打折问题?
- 巴黎春天打折
  - 折扣方式改掉怎么办?
  - 多种折扣方式同时使用怎么办?

Homework

- 写出商场促销的策略模式的设计方案,画出主要的类,并给出简单代码和实现结果。
- 11月19日提交,纸版和电子版.