



深圳大学  
Shenzhen University

# 第15-3讲

## 策略模式

软件体系结构与设计模式  
Software Architecture & Design Pattern

深圳大学计算机与软件学院

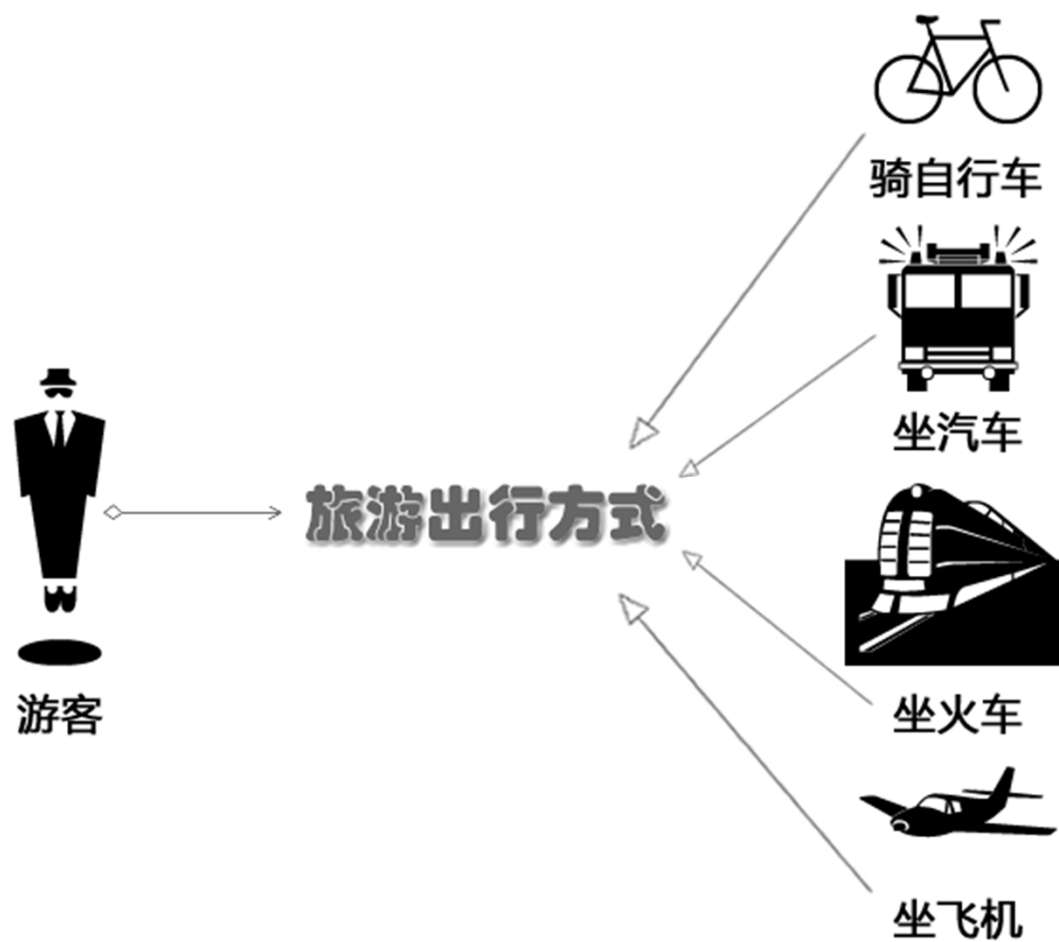


# 主要内容

- ◆ 策略模式动机与定义
- ◆ 策略模式结构与分析
- ◆ 策略模式实例与解析
- ◆ 策略模式效果与应用

# 策略模式动机

## ■ 旅游出行方式示意图





## 策略模式动机

- 实现某个目标的途径不止一条，可根据实际情况选择一条合适的途径
- 软件开发：
  - 多种算法，例如排序、查找、打折等
  - 使用硬编码(Hard Coding)实现将导致系统违背开闭原则，扩展性差，且维护困难
  - 可以定义一些独立的类来封装不同的算法，每一个类封装一种具体的算法 → 策略类 → 策略模式



## 策略模式定义

### ■ 对象行为型模式

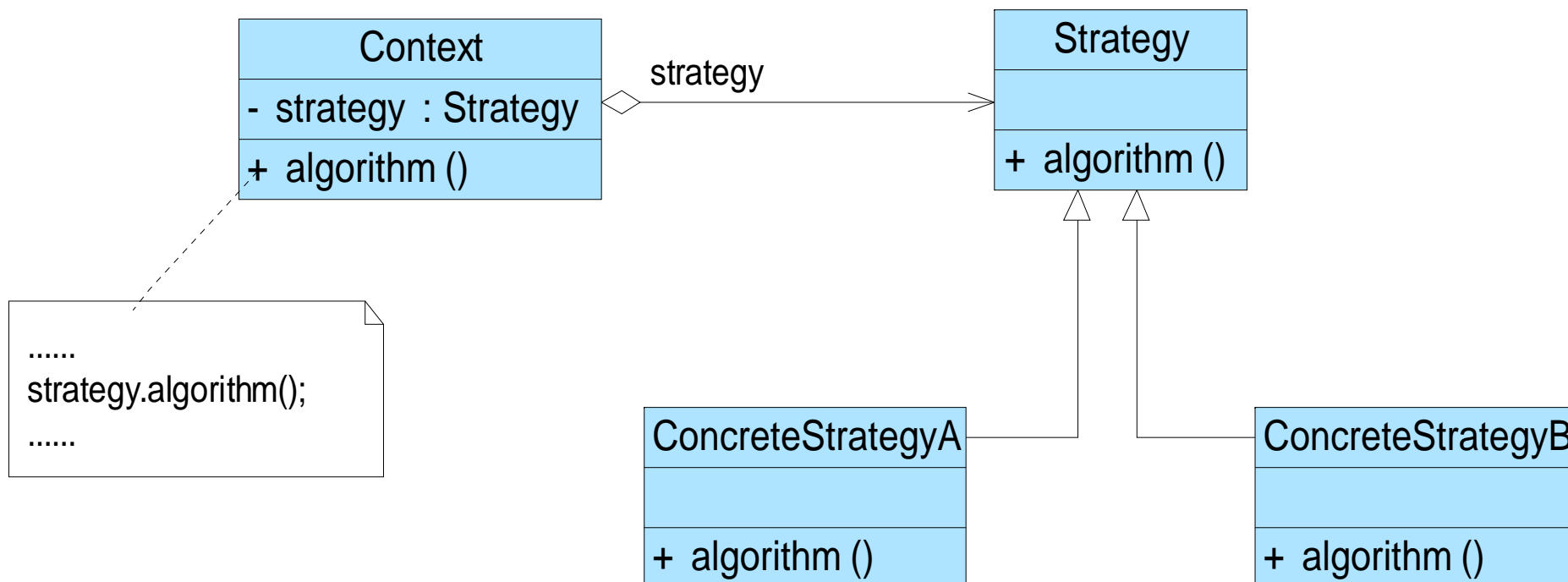
**策略模式：**定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法可以独立于使用它的客户变化。

**Strategy Pattern:** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- 又称为政策(Policy)模式
- 每一个封装算法的类称之为策略(Strategy)类
- 策略模式提供了一种可插入式(Pluggable)算法的实现方案

## 策略模式结构

- 策略模式
- 包含如下角色：
  - Context: 环境类
  - Strategy: 抽象策略类
  - ConcreteStrategy: 具体策略类



## 策略模式分析

- 每一个封装算法的类称之为**策略(Strategy)**类
- 策略模式提供了一种**可插入式(Pluggable)**算法的实现方案
- 环境类示例代码：

```
public class Context {  
    private Strategy strategy; //维持一个对抽象策略类的引用  
  
    //注入策略对象  
    public void setStrategy(Strategy strategy) {  
        this.strategy= strategy;  
    }  
  
    //调用策略类中的算法  
    public void algorithm() {  
        strategy.algorithm();  
    }  
}
```



## 策略模式实例

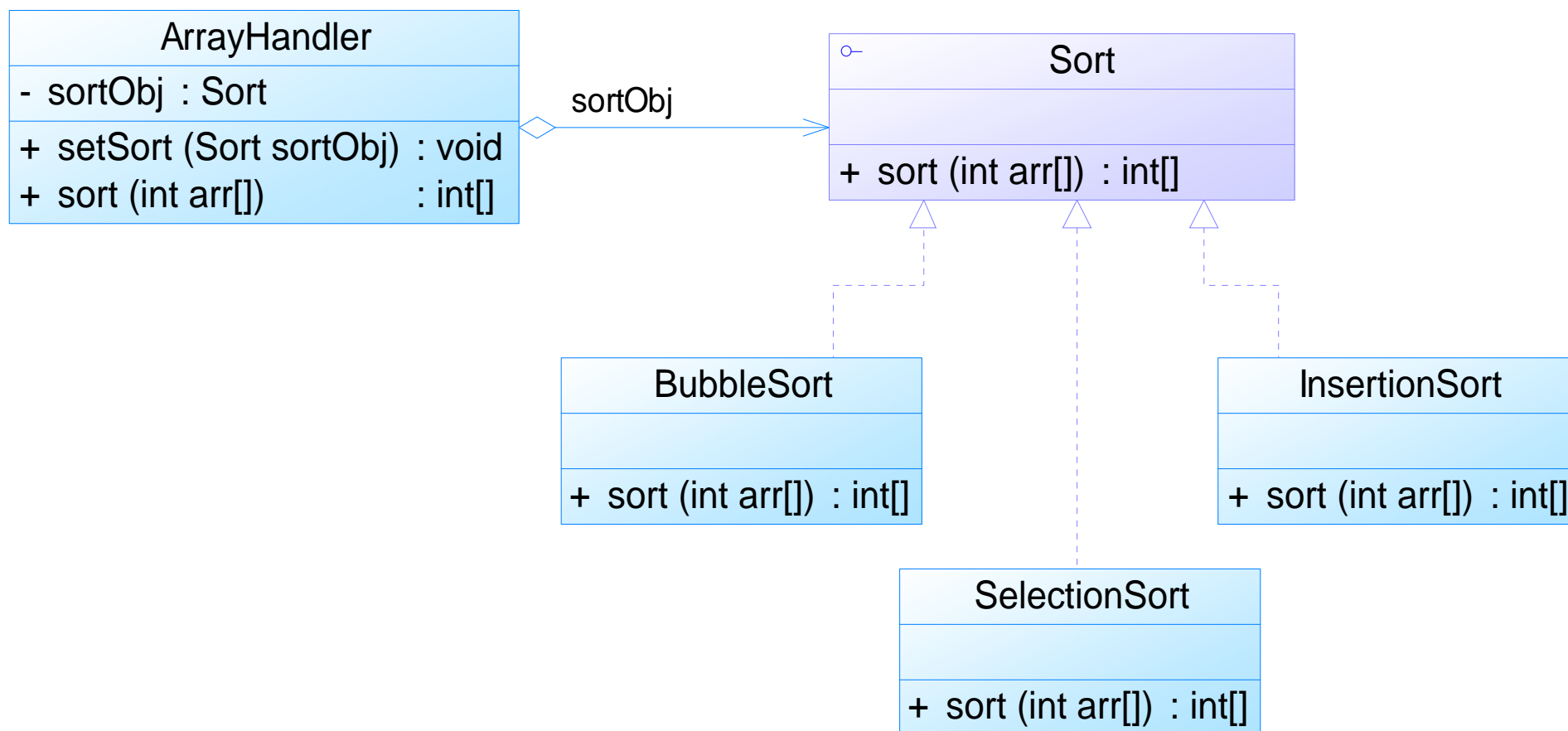
### ■ 排序策略：实例说明

- 某系统提供了一个用于对数组数据进行操作的类，该类封装了对数组的常见操作，如查找数组元素、对数组元素进行排序等。现以排序操作为例，使用策略模式设计该数组操作类，使得客户端可以动态地更换排序算法，可以根据需要选择冒泡排序或选择排序或插入排序，也能够灵活地增加新的排序算法。



## 策略模式实例与解析

### ■ 排序策略：参考类图



## 策略模式实例

- 排序策略：参考代码
- DesignPatterns之strategy包

Sort.java

```
1 package strategy;  
2  
3 public interface Sort  
4 {  
5     public abstract int[] sort(int arr[]);  
6 }
```

## 策略模式实例与解析

ArrayHandler.java

```
1 package strategy;
2
3 public class ArrayHandler
4 {
5     private Sort sortObj;
6
7     public int[] sort(int arr[])
8     {
9         sortObj.sort(arr);
10        return arr;
11    }
12
13    public void setSortObj(Sort sortObj) {
14        this.sortObj = sortObj;
15    }
16 }
```

```
1 package strategy;
2
3 public class BubbleSort implements Sort
4 {
5     public int[] sort(int arr[])
6     {
7         int len=arr.length;
8         for(int i=0;i<len;i++)
9         {
10             for(int j=i+1;j<len;j++)
11             {
12                 int temp;
13                 if(arr[i]>arr[j])
14                 {
15                     temp=arr[j];
16                     arr[j]=arr[i];
17                     arr[i]=temp;
18                 }
19             }
20         }
21         System.out.println("冒泡排序");
22         return arr;
23     }
24 }
```

```
1 package strategy;
2
3 public class InsertionSort implements Sort
4 {
5     public int[] sort(int arr[])
6     {
7         int len=arr.length;
8         for(int i=1;i<len;i++)
9         {
10             int j;
11             int temp=arr[i];
12             for(j=i;j>0;j--)
13             {
14                 if(arr[j-1]>temp)
15                 {
16                     arr[j]=arr[j-1];
17
18                 }else
19                     break;
20             }
21             arr[j]=temp;
22         }
23         System.out.println("插入排序");
24         return arr;
25     }
26 }
```

```
3 public class QuickSort implements Sort
4 {
5     public int[] sort(int arr[])
6     {
7         System.out.println("快速排序");
8         sort(arr,0,arr.length-1);
9         return arr;
10    }
11
12    public void sort(int arr[],int p, int r)
13    {
14        int q=0;
15        if(p<r)
16        {
17            q=partition(arr,p,r);
18            sort(arr,p,q-1);
19            sort(arr,q+1,r);
20        }
21    }
```

```
22
23 public int partition(int[] a, int p, int r)
24 {
25     int x=a[r];
26     int j=p-1;
27     for(int i=p;i<=r-1;i++)
28     {
29         if(a[i]<=x)
30         {
31             j++;
32             swap(a,j,i);
33         }
34     }
35     swap(a,j+1,r);
36     return j+1;
37 }
38
39 public void swap(int[] a, int i, int j)
40 {
41     int t = a[i];
42     a[i] = a[j];
43     a[j] = t;
44 }
45 }
```

```
3 public class SelectionSort implements Sort
4 {
5     public int[] sort(int arr[])
6     {
7         int len=arr.length;
8         int temp;
9         for(int i=0;i<len;i++)
10        {
11            temp=arr[i];
12            int j;
13            int samllestLocation=i;
14            for(j=i+1;j<len;j++)
15            {
16                if(arr[j]<temp)
17                {
18                    temp=arr[j];
19                    samllestLocation=j;
20                }
21            }
22            arr[samllestLocation]=arr[i];
23            arr[i]=temp;
24        }
25        System.out.println("选择排序");
26        return arr;
27    }
28 }
```



```
7 public class XMLUtil
8 { //该方法用于从XML配置文件中提取具体类名，并返回一个实例对象
9     public static Object getBean()
10    {
11        try
12        {
13            //创建文档对象
14            DocumentBuilderFactory dFactory = DocumentBuilderFactory.newInstance();
15            DocumentBuilder builder = dFactory.newDocumentBuilder();
16            Document doc;
17            doc = builder.parse(new File("Strategyconfig.xml"));
18            //获取包含类名的文本节点
19            NodeList nl = doc.getElementsByTagName("className");
20            Node classNode=nl.item(0).getFirstChild();
21            String cName=classNode.getNodeValue();
22            //通过类名生成实例对象并将其返回
23            Class c=Class.forName(cName);
24            Object obj=c.newInstance();
25            return obj;
26        }
27        catch(Exception e)
28        {
29            e.printStackTrace();
30            return null;
31        }
32    }
33 }
```

Strategyconfig.xml

```
1 <?xml version="1.0"?>
2 <config>
3     <className>strategy.BubbleSort</className>
4 </config>
```

Client.java

```
3 public class Client
4 {
5     public static void main(String args[])
6     {
7         int arr[]={1,4,6,2,5,3,7,10,9};
8         int result[];
9         ArrayHandler ah=new ArrayHandler();
10
11         Sort sort;
12         sort=(Sort)XMLUtil.getBean();
13
14         ah.setSortObj(sort); //设置具体策略
15         result=ah.sort(arr);
16
17         for(int i=0;i<result.length;i++)
18         {
19             System.out.print(result[i] + ",");
20         }
21     }
22 }
```



## 策略模式效果与应用

### ■ 策略模式优点：

- 提供了对开闭原则的完美支持，用户可以在不修改原有系统的基础上选择算法或行为，也可以灵活地增加新的算法或行为
- 提供了管理相关的算法族的办法
- 提供了一种可以替换继承关系的办法
- 可以避免多重条件选择语句
- 提供了一种算法的复用机制，不同的环境类可以方便地复用策略类



## 策略模式效果与应用

### ■ 策略模式缺点：

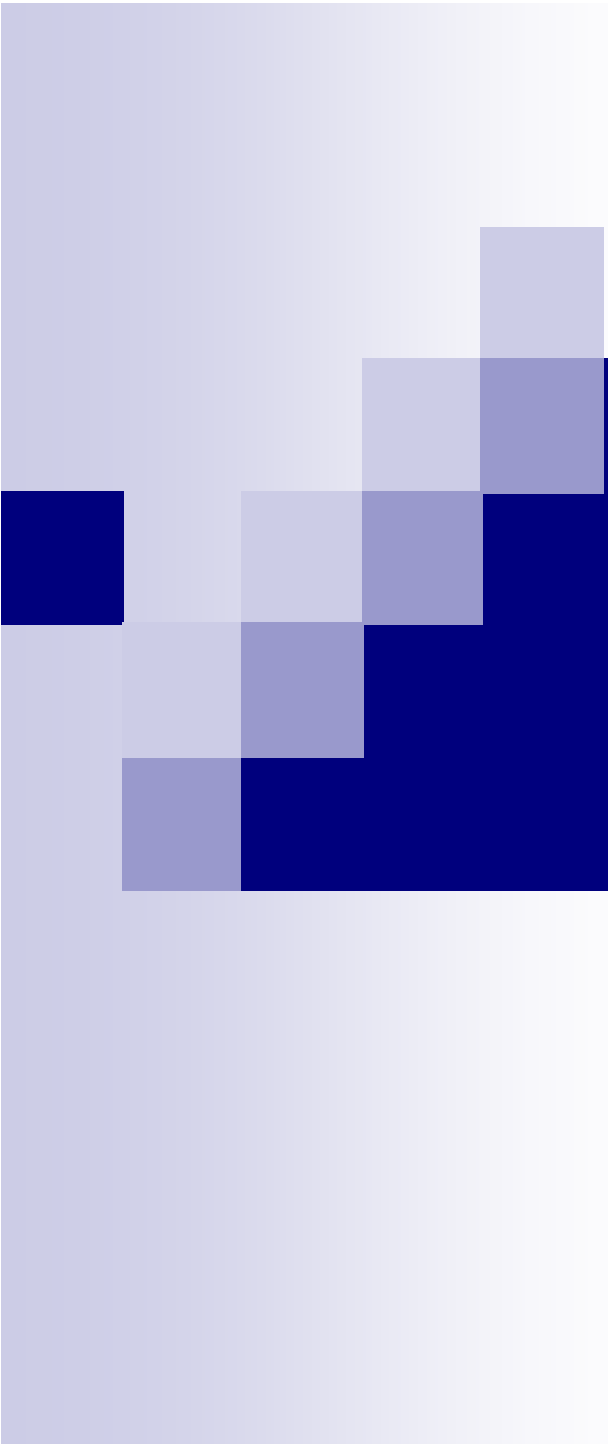
- 客户端必须知道所有的策略类，并自行决定使用哪一个策略类
- 将造成系统产生很多具体策略类
- 无法同时在客户端使用多个策略类



## 策略模式效果与应用

### ■ 在以下情况下可以使用策略模式：

- 一个系统需要动态地在几种算法中选择一种
- 避免使用难以维护的多重条件选择语句
- 不希望客户端知道复杂的、与算法相关的数据结构，提高算法的保密性与安全性



谢谢