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

Lecture_08 通用的职责分配软件原则 GRASP（二）

主讲: 姜宁康 博士

5、其他面向对象设计原则1: 开-闭原则OCP

- Open-Closed Principle (OCP)

5.1 设计变坏的前兆 Signs of Rotting Design

- 僵硬性 Rigidity
  - 难以更改代码 code difficult to change
  - 从管理角度，拒绝任何的变化成为一种制度 management reluctance(拒绝) to change anything becomes policy
- 易碎性 Fragility
  - 即使是小小的改动也会导致级联性的后果的 even small changes can cause cascading effects
  - 代码在意想不到的地方终止 code breaks in unexpected places
- 固定性 Immobility
  - 代码纠缠在一起根本不可能重用 code is so tangled(纠结) that it's impossible to reuse anything
- 黏滞性 Viscosity
  - 宁愿重新编写也不愿意修改 much easier to hack(乱砍) than to preserve original design

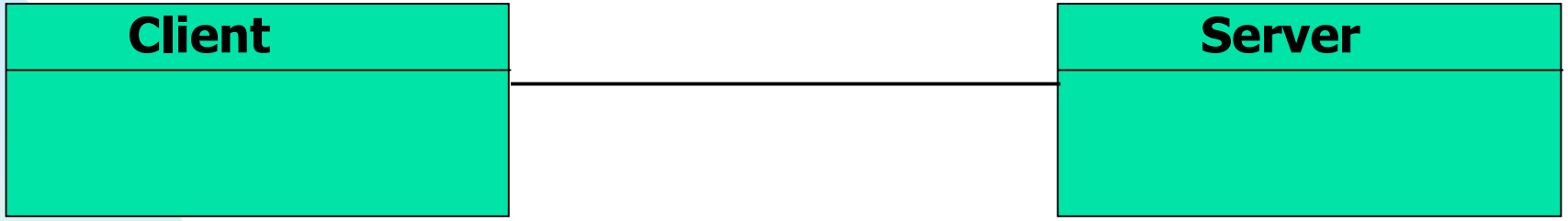
5.2 设计变坏的原因 Causes of Rotting Design

- 需求不停地变化 Changing Requirements
  - 这是不可避免的 is inevitable
  - I. Jacobson, OOSE, 1992说，“所有的系统在其生命周期都在发生变化，如果拟开发的系统生命期多于一个版本的话，就必须记住这一点” All systems change during their life-cycles. This must be borne in mind when developing systems expected to last longer than the first version"
- 设计的问题：“依赖管理”失衡
  - 导致高耦合、低内聚

5.3 开闭原则 Open-Closed Principle (OCP)

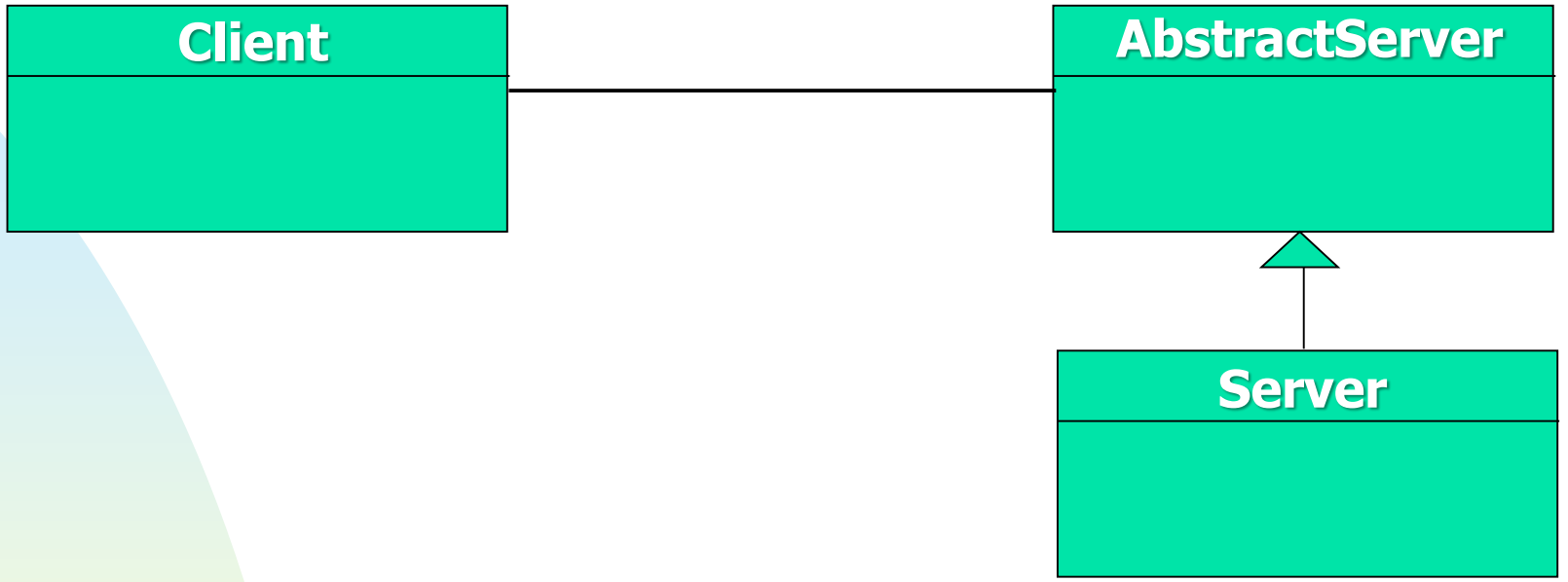
- 软件系统在其生命周期都在发生变化
  - 无论是好的设计还是坏的设计，都面临着这个问题 both better designs and poor designs have to face the changes
  - 但好的设计是稳定的 good designs are stable
- 开-闭原则
  - 软件系统应当允许功能扩展(即开放性) Software entities should be open for extension,
  - 但是不允许修改原有的代码（即关闭性） but closed for modification (OCP)
- 遵循开-闭原则的模块符合下列准则
  - Open for Extension – 可以扩展行为以满足新的需求
  - CLosed for Modification – 但不允许修改模块的源代码 the source code of the module is not allowed to change

5.4 例子: 需要修改的客户端



- Client and Server are concrete classes
- Client class uses Server class
- If Client object wants to switch to a different Server object, what would need to happen?
  - Client code needs to be modified to name the new Server class

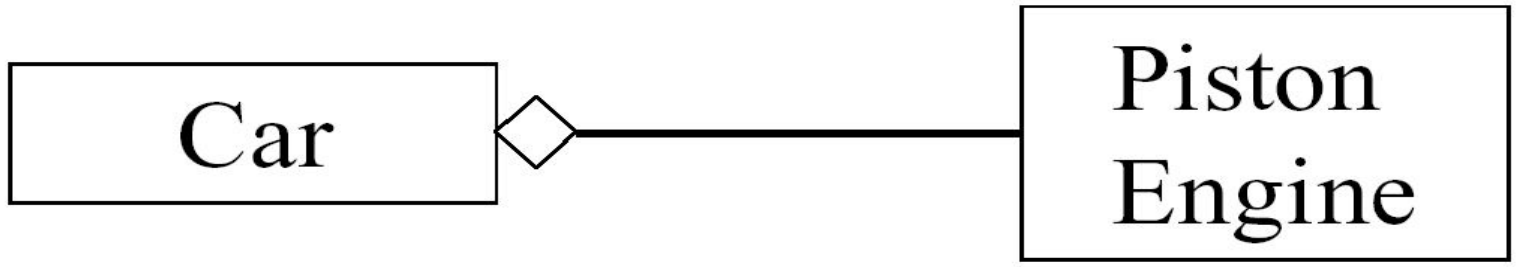
5.4 Example: 可以扩展的客户端



- **How is this "open" ?**

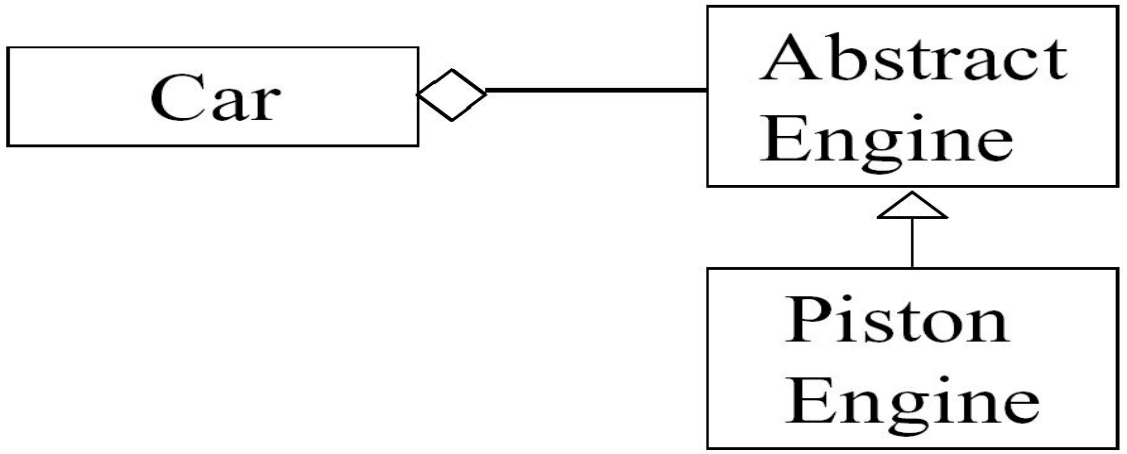
  Since the Client depends on the AbstractServer, we can simply switch the Client to using a different Server, by providing a new Server implementation. Client code is unaffected!

5.4 Example: Open the door ...



- How to make the Car run efficiently with a TurboEngine?
- Only by changing the Car!
  - ...in the given design

5.4 Example: ... But Keep It Closed!

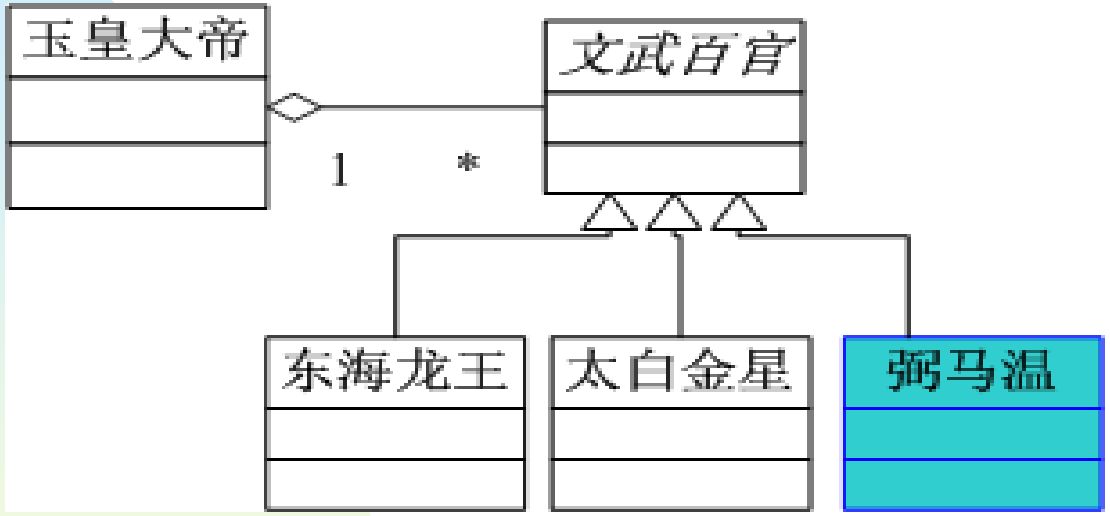


- A client must not depend on a concrete class!
- It must depend on an abstract class ...
- ...using polymorphic dependencies (calls)

Closure not complete but strategic(战略的): "No significant program can be 100% closed "
R.Martin, "The Open-Closed Principle," 1996

5.4 Example

- **OCP Example**



Close

Ultimate Rules

Open

Add a new place

Encapsulate what varies

5.5 OCP的启发 Heuristics

1）定义所有的对象 - 数据为私有的 Make all object-data private
2）不要使用全局变量 No Global Variables!

- 修改公有的数据，经常冒着“打开”模块的风险 Changes to public data are always at risk to “open” the module
  - 它们常常会引起涟漪效应，导致许多地方连锁修改 They may have a rippling effect requiring changes at many unexpected locations
  - 很难找全出错的地方并修改，一处修改会导致多处又出问题 Errors can be difficult to completely find and fix. Fixes may cause errors elsewhere

5.6 Another OCP Example

- **Consider the following method:**

```
public double totalPrice(Part[] parts) {
  double total = 0.0;
  for (int i=0; i<parts.length; i++) {
    total += parts[i].getPrice();
  }
  return total;
}
```

- **这个方法的任务是计算所有零部件（parts）的总价** The job of the above method is to total the price of all parts in the specified array of parts
- **请问，这符合OCP原则吗?** Does this conform to OCP

- **是的，只要Part是一个基类或者是应用了多态的接口，当零部件发生变化时这个类能够适应新的Part，并且不需要修改已有的源码** YES! If Part is a base class or an interface and polymorphism is being used, then this class can easily accommodate new types of parts without having to be modified!

5.6 Another OCP Example

- But what if the Accounting Department now decreed(判决) that motherboard parts and memory parts have a premium applied when figuring the total price?
- Would the following be a suitable modification? Does it conform to OCP?

```
public double totalPrice(Part[] parts) {
  double total = 0.0;
  for (int i=0; i<parts.length; i++) {
    if (parts[i] instanceof Motherboard)
      total += (1.45 * parts[i].getPrice());
    else if (parts[i] instanceof Memory)
      total += (1.27 * parts[i].getPrice());
    else
      total += parts[i].getPrice();
  }
  return total;
}
```

5.6 Another OCP Example

- **No! Every time the Accounting Department comes out with a new pricing policy, we have to modify totalPrice ( ) method. This is not "Closed for modification"**

- **These policy changes have to be implemented some place, so what is a solution?**

- **Version 1 - Could incorporate the pricing policy in getPrice ( ) method of Part**

5.6 Another OCP Example

- **Here are example Part and Concrete Part classes:**

```
// Class Part is the superclass for all parts.
public class Part {
  private double price;
  public Part(double price) (this.price = price;}
  public void setPrice(double price) {this.price = price;}
  public double getPrice() {return price;}
}

// Class ConcretePart implements a part for sale.
// Pricing policy explicit here!
public class ConcretePart extends Part {
  public double getPrice() {
    // return (1.45 * price);    //Premium
    return (0.90 * price);       //Labor Day Sale
  }
```

Add method

Code in concrete classes

- **Does this work? Is it "closed for modification"?**
  - **No. We must now modify each subclass of Part whenever the pricing policy changes!**

5.6 Another OCP Example

- How to make it "Closed for Modification"?
- Better idea - have a PricePolicy class which can be used to provide different pricing policies

```
// The Part class now has a contained PricePolicy object.
public class Part {
  private double price;
  private PricePolicy pricePolicy;

  public void setPricePolicy(PricePolicy pricePolicy) {
    this.pricePolicy = pricePolicy;}
  public void setPrice(double price) {this.price = price;}
  public double getPrice() {return pricePolicy.getPrice(price);}
}
```

5.6 Another OCP Example

```
/**
 * Class PricePolicy implements a given price policy.
 */
public class PricePolicy {
  private double factor;

  public PricePolicy (double factor) {
    this.factor = factor;
  }

  public double getPrice(double price) {return price * factor;}

}
```

With this solution we can dynamically set pricing policies at run time by changing the PricePolicy object that an existing Part object refers to

5.7 小结

- OCP解决软件的僵硬性和易碎性 attacks software rigidity and fragility!
  - When one change causes a cascade of changes
- OCP 宣言
  - 我们应当尝试设计永远不需要修改的模块 we should attempt to design modules that never need to be changed
  - 系统行为的扩展只需要增加新的代码，不能修改已有的代码 extend the behavior of the system by adding new code. We do not modify old code
  - 模块不允许修改已有的代码，而这种修改会影响客户端 the module is closed to modification in ways that affect clients

- 本讲结束