

设计模式



设计模式基础

课程整体目标

课程目标

- 理解面向对象的编程思想
- 了解设计模式的本质思想
- 掌握常用的设计模式
- 能够将设计模式应用在项目开发与设计中

本章目标

- 了解设计模式的来源
- 理解设计模式的本质
- 理解设计模式的基本原则
- 了解设计模式的分类
- 如何利用设计模式解决设计问题

设计模式基础介绍

- 理解设计模式的本质
- 理解设计模式的基本原则

建筑中的设计模式

说明

- Christopher Alexander说过：“每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动”。
- 尽管Alexander所指的是城市和建筑模式，但他的思想也同样适用于面向对象设计模式，只是在面向对象的解决方案里，我们用对象和接口代替了墙壁和门窗。两类模式的核心都在于提供了相关问题的解决方案。

设计模式四巨头(GOF)

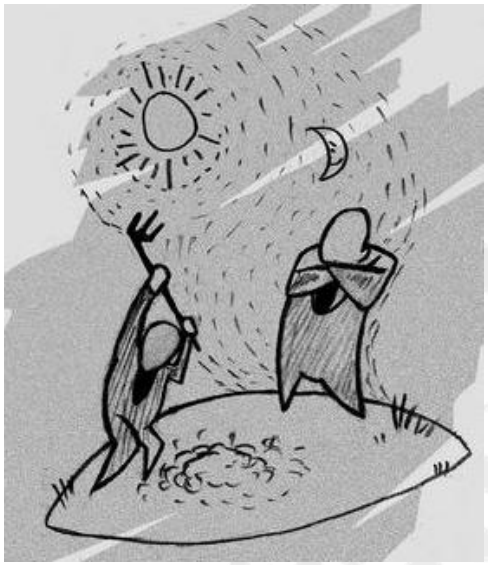
Richard Helm, Eric Gamma, Ralph Johnson,John Vlissides

设计模式的来源

- 建筑师Christopher Alexander在1977/79年编制了一本汇集设计模式的书。但是这种设计模式的思想在建筑设计领域里的影响远没有后来在软件开发领域里传播的广泛。
- Kent Beck和Ward Cunningham在1987年，利用Christopher Alexander在建筑设计领域里的思想开发了设计模式，并把此思想应用在Smalltalk中的图形用户接口的生成中。
- 一年后Erich Gamma在他的苏黎世大学博士毕业论文中开始尝试把这种思想改写为适用于软件开发。与此同时James Coplien在1989年至1991 年也在利用相同的思想致力于C++的开发，而后于1991年发表了他的著作Advanced C++ Idioms。
- 就在这一年Erich Gamma 得到了博士学位，然后去了美国，在那与Richard Helm, Ralph Johnson ,John Vlissides 合作出版了Design Patterns - Elements of Reusable Object-Oriented Software一书，在此书中共收录了23个设计模式。
- 这四位作者在软件开发领域里也以他们的匿名著称Gang of Four（四巨头，简称GoF），并且是他们在此书中的协作导致了软件设计模式的突破。有时这个匿名GoF也会用于指代前面提到的那本书。

设计模式的本质

说明



- 设计模式是面向对象软件设计经验的总结；利用设计模式可以更简单方便地复用成功的设计和体系结构。
- 设计模式的本质仍然是面向对象设计，其精髓就是封装变化。只有将变化封装，进行合理的抽象，才能更大的保证软件的可扩展性以及模块间的松散耦合。
- 在设计时，我们需要寻找软件中可能存在的“变化”，然后利用抽象的方式对这些变化进行封装。由于抽象没有具体的实现，就代表了一种无限的可能性，使得扩展成为可能。

理解设计模式的基本原则

- 针对接口编程，而不是针对实现编程。
  - 由于接口没有提供具体的实现，因此它是可以被替换的。
  - 该原则意味着定义的变量以及传递的参数其类型应该为接口（抽象）类型。
  - 在设计时，应考虑系统的外部接口，而不要首先考虑具体的实现。应用该原则，可以在抽象层面上统一各种实现的接口定义。

理解设计模式的基本原则

- 优先使用对象组合，而不是类继承。
  - 组合方式在对象的依赖关系上，要弱于继承的方式。使用组合有利于保证对象的弱依赖。
  - 本原则通常被称为“合成/聚合复用原则”。
  - 本原则并不是要求我们完全抛弃继承，相反，它是对继承的一种规范或约束。
  - 父类若存在多个变化点，采用继承方式就会导致子类的泛滥。如果将这些变化点分离出来，分别建立细粒度的继承体系，然后在抽象层面上表达出对象之间的合成/聚合关系，就能保证父类的细粒度，并降低对象之间的耦合度。

设计模式的分类

- 模式依据其目的可分为创建型(Creational)、结构型(Structural)、或行为型(Behavioral)三种。创建型模式与对象的创建有关；结构型模式处理类或对象的组合；行为型模式对类或对象怎样交互和怎样分配职责进行描述。
- 根据范围，可以指定模式是用于类还是用于对象。类模式处理类和子类之间的关系，这些关系通过继承建立，是静态的，在编译时刻便确定下来了。对象模式处理对象间的关系，这些关系在运行时刻是可以变化的，更具动态性。

设计模式的分类

		目　的		
		创　建　型	结　构　型	行　为　型
范围	类	Factory Method(3.3)	Adapter(类)(4.1)	Interpreter(5.3) Template Method(5.10)
	对象	Abstract Factory(3.1) Builder(3.2) Prototype(3.4) Singleton(3.5)	Adapter(对象)(4.1) Bridge(4.2) Composite(4.3) Decorator(4.4) Facade(4.5) Flyweight(4.6) Proxy(4.7)	Chain of Responsibility(5.1) Command(5.2) Iterator(5.4) Mediator(5.5) Memento(5.6) Observer(5.7) State(5.8) Strategy(5.9) Visitor(5.10)

如何解决设计问题

- 寻找合适的对象
  - 面向对象程序由对象组成，对象包括数据和对数据进行操作的过程，过程通常称为方法或操作。对象在收到客户的请求(或消息)后，执行相应的操作。
  - 面向对象设计方法学支持许多设计方法。你可以写出一个问题描述，挑出名词和动词，进而创建相应的类和操作；或者，你可以关注于系统的协作和职责关系；或者，你可以对现实世界建模，再将分析时发现的对象转化至设计中。
  - 设计模式帮你确定并不明显的抽象和描述这些抽象的对象。例如，描述过程或算法的对象现实中并不存在，但它们却是设计的关键部分。策略(Strategy)模式描述了怎样实现可互换的算法族。状态(State)模式将实体的每一个状态描述为一个对象。

如何解决设计问题

- 决定对象的粒度
  - 我们不能设置一个庞大的对象来负责处理系统的全部请求，这样的设计既不利于重用，也不利于扩展。良好的设计应该是对象各司其职，同时相互之间进行合理的协作。
  - 我们应该尽可能地保证一个对象只完成一项职责，这样的设计能够保证引起对象变化的原因只有一个。
  - 设计模式可以帮助我们分解对象，并为不同的对象分配不同的职责。例如Factory Method模式就专门定义了用于创建其他对象的对象。

如何解决设计问题

- 指定对象接口
  - 对象声明的每一个操作指定操作名、作为参数的对象和返回值，这就是所谓的操作签名(signature)。对象操作所定义的所有操作签名的集合被称为该对象的接口(interface)。
  - 在面向对象系统中，接口是基本的组成部分。对象只有通过它们的接口才能与外部交流，如果不通过对象的接口就无法知道对象的任何事情，也无法请求对象做任何事情。对象接口与其功能实现是分离的，不同对象可以对请求做不同的实现，也就是说，两个有相同接口的对象可以有完全不同的实现。
  - 对象的多态(polymorphism)保证了类型的动态绑定，它使得接口的实现是可以互相替换的。

如何解决设计问题

- 描述对象的实现
  - 对象的实现是由它的类决定的，类指定了对象的内部数据和表示，也定义了对象所能完成的操作。
  - 对象通过实例化类来创建，此对象被称为该类的实例。
- 设计应支持重用
  - 使用继承和组合可以实现对象的重用。
- 设计应支持变化
  - 寻找可能变化的点，考虑如何分离和隔离变化，以及抽象变化。

总结

- 设计模式是面向对象软件设计经验的总结；利用设计模式可以更简单方便地复用成功的设计和体系结构。
- 设计模式的本质仍然是面向对象设计，其精髓就是封装变化。
- 设计模式的基本原则：
  - 针对接口编程，而不是针对实现编程。
  - 优先使用对象组合，而不是类继承。
- 设计模式的分类：
  - 创建型模式
  - 结构型模式
  - 行为型模式

作业

- 熟悉面向对象设计的核心要素
- 熟悉UML的类图和时序图
- 思考合成/聚合复用原则的思想
- 思考面向接口编程的思想

设计模式

工厂方法(Factory Method)模式

设计模式基础回顾

- 设计模式是面向对象软件设计经验的总结；
- 设计模式的本质仍然是面向对象设计，其精髓就是封装变化。
- 设计模式的基本原则：
  - 针对接口编程，而不是针对实现编程。
  - 优先使用对象组合，而不是类继承。
- 设计模式的分类：
  - 创建型模式
  - 结构型模式
  - 行为型模式

本章目标

- 理解创建型模式的本质
- 了解工厂方法模式的意图和本质
- 理解工厂方法模式的结构
- 掌握工厂方法模式的适用性
- 掌握工厂方法模式在设计中的应用

工厂方法模式

- 工厂方法模式的结构
- 在设计中如何应用工厂方法模式

创建型模式的本质

- 创建型模式要处理的是对象的创建。在具体的实例中，我们会遇到这样的情况，就是我们要创建的对象可能会根据需求的不同，而发生变化。此时，我们认为，创建是变化的。
- 而创建型模式的目的就是封装创建的变化。例如，工厂方法模式和抽象工厂模式，通过建立抽象的工厂类，封装未来对象的创建所引起的可能变化。
- 建造者模式则是对对象内部的创建进行封装，由于细节对抽象的可替换性，使得将来面对对象内部创建方式的变化，可以灵活的进行扩展或替换。

工厂方法模式的意图和本质

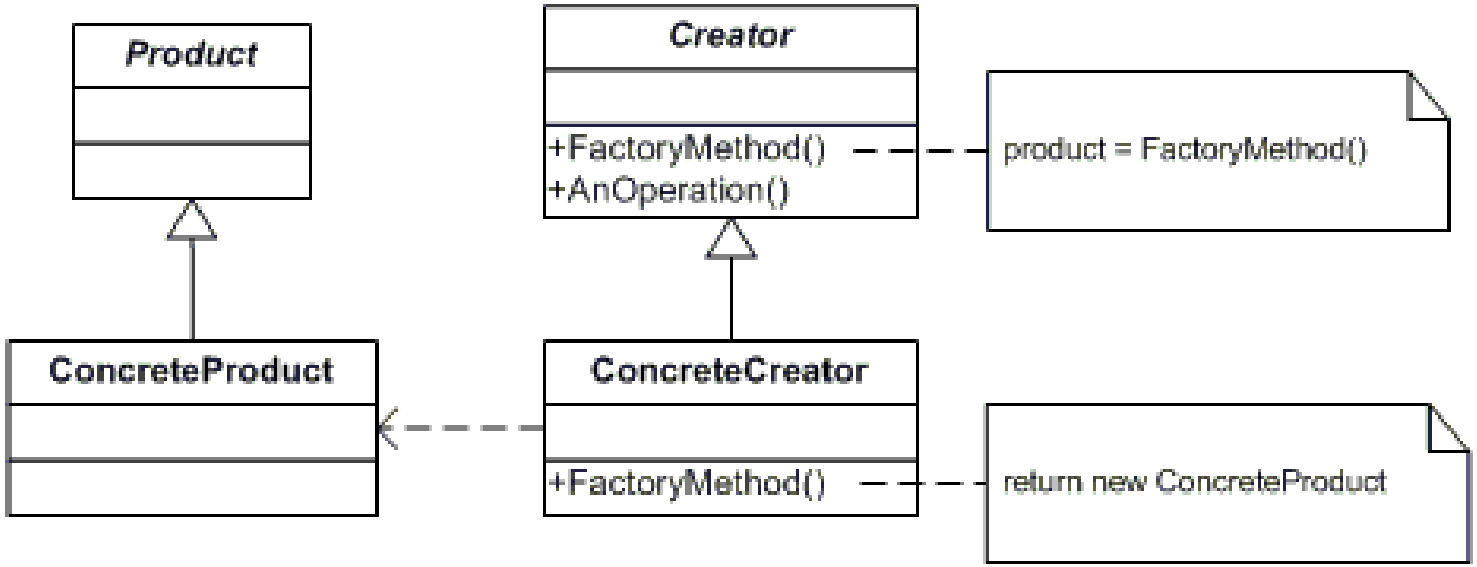
- 意图
  - 定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method使一个类的实例化延迟到其子类。
  - 用于创建一组产品，其目的在于更好的管理对象的创建，支持程序的扩展性。

本质

- 工厂方法模式转移了创建对象的职责，由专门的类负责产品的创建；同时，它对创建逻辑进行了抽象，从而支持产品创建的扩展。

工厂方法模式的结构

说明

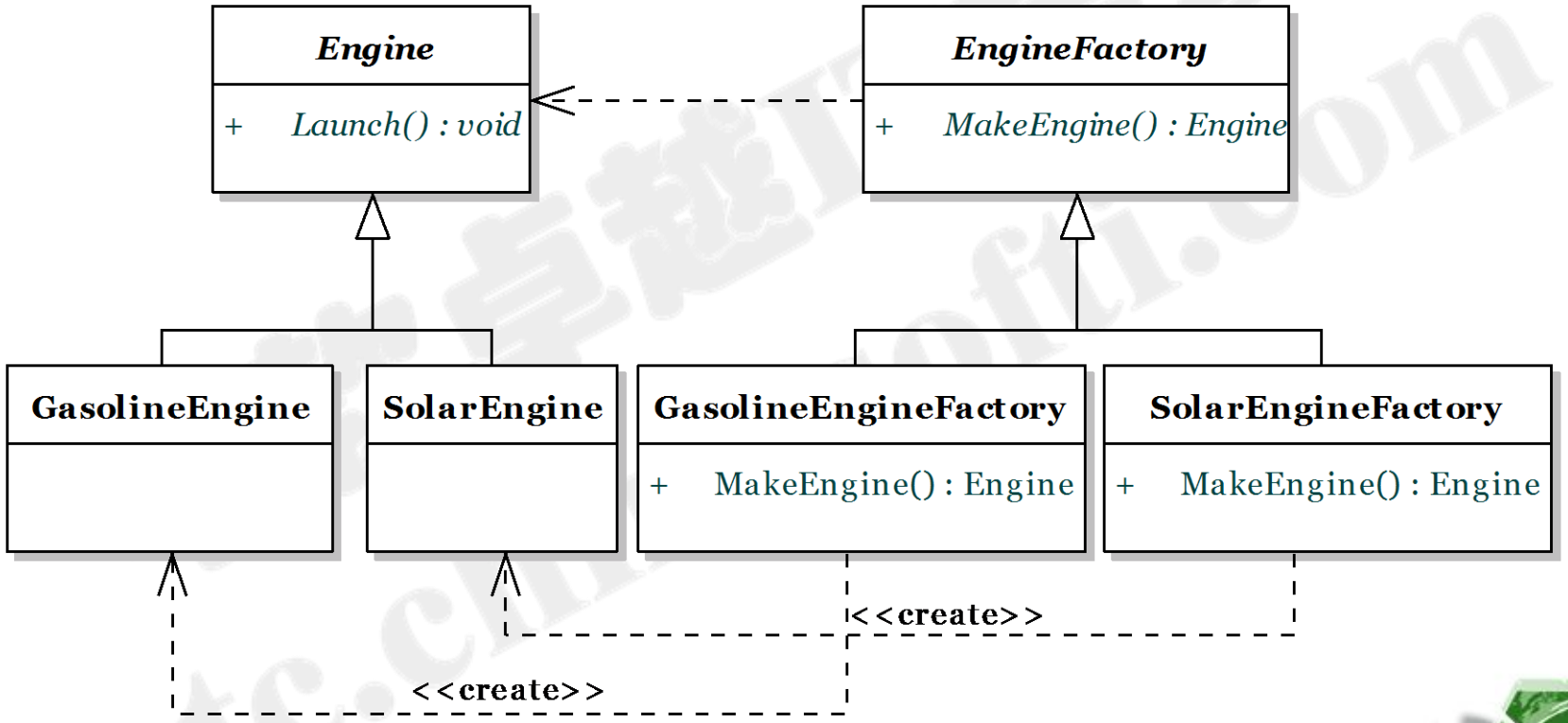


工厂方法模式的适用性

- 在下列情况下可以使用工厂方法模式：
  - 当一个类不知道它所必须创建的对象的类的时候。这意味着对象的创建者只关心被创建对象的接口，而不是具体的实现。
  - 当一个类希望由它的子类来指定它所创建的对象的时候。
  - 当创建的一组对象在未来可能存在变化的时候。

汽车引擎产品的创建

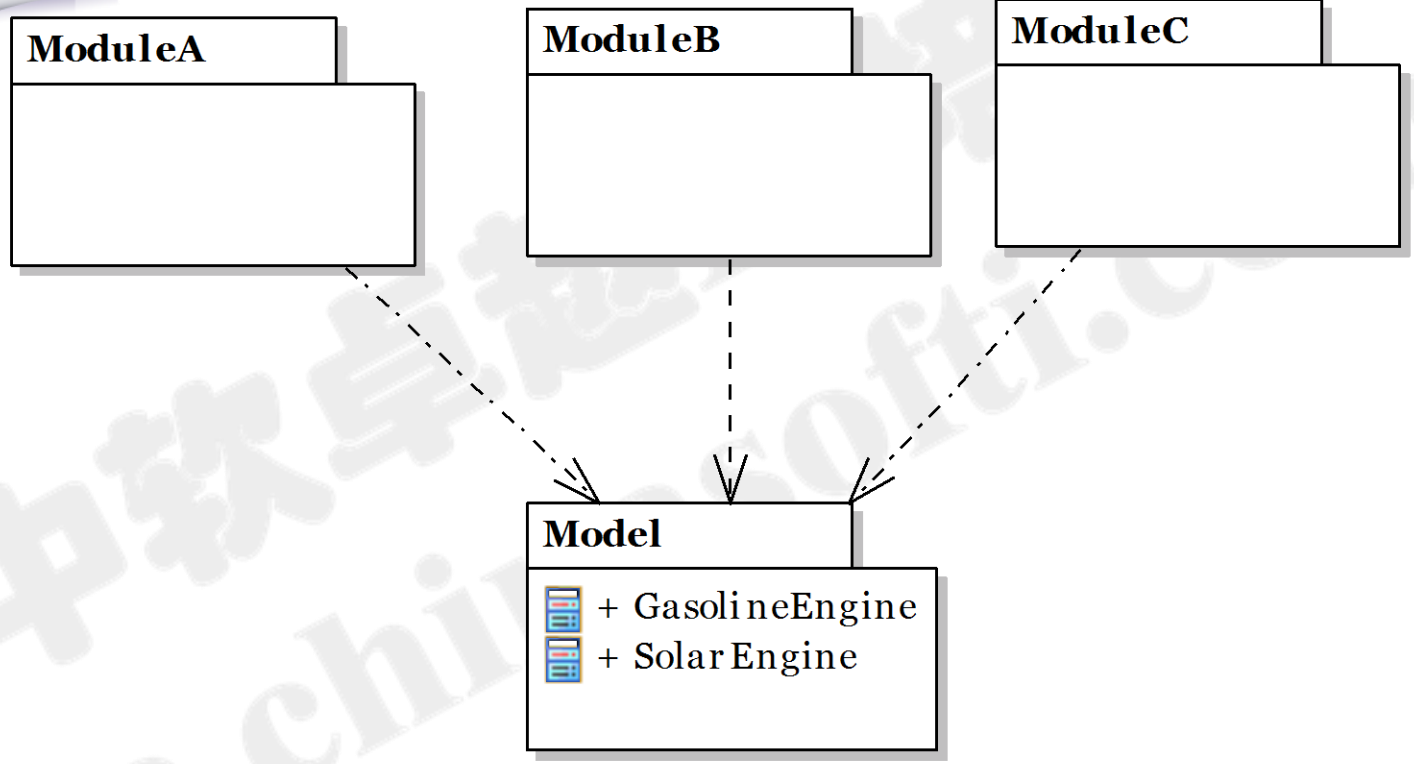
案例



工厂方法模式在设计中的应用

- 工厂方法模式可以避免调用者直接创建具体的产品对象，从而解除具体产品对象与调用者之间的耦合。然而，它却需要创建具体的工厂对象，导致调用者与工厂之间的耦合。这其中的区别在于： 工厂对象与产品对象创建的频率不同。
- 以汽车引擎为例，系统需要创建大量的Engine对象，而EngineFactory对象则只需要创建一次。因而，在产品对象发生变化时，涉及到的修改就只有一处。

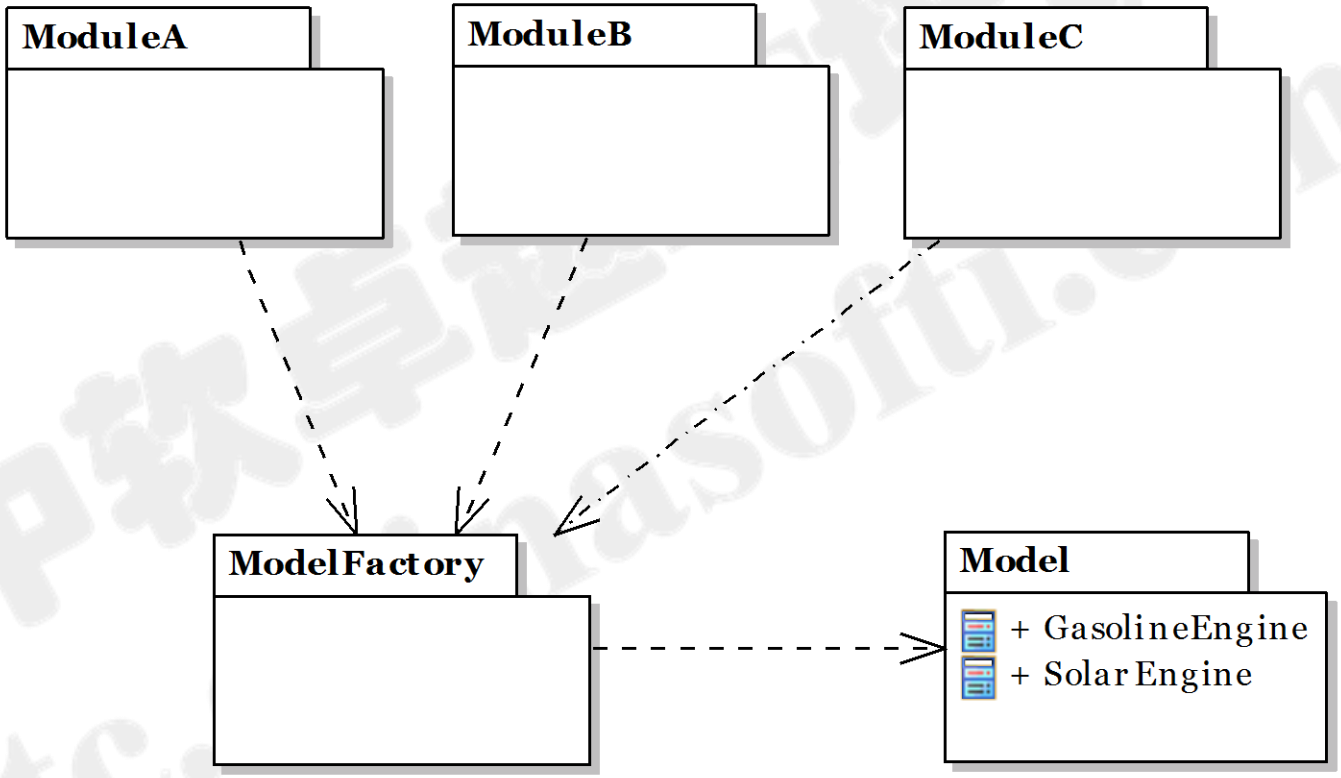
如果没有使用工厂方法模式



直接依赖于产品对象所属的Model模块

使用工厂方法模式

案例



利用工厂解除与Model模块的依赖

总结

- 工厂方法模式是创建型模式中的一种。
- 它的意图是定义一个用于创建对象的接口，让子类决定实例化哪一个类。
- 工厂方法模式的结构中，包括抽象工厂角色、具体工厂角色、抽象产品角色和具体产品角色。

思考题

- URL会根据URL地址的协议创建不同的URLConnection对象，例如当传入的地址为http://时，则创建HttpURLConnection对象，如果为ftp://，则创建FtpURLConnection对象。
- 应该如何设计，以应对URLConnection对象可能的扩展。

作业

- 根据工厂方法模式的结构，使用C#语言编写代码实现工厂方法模式；

设计模式

抽象工厂(Abstract Factory)模式

工厂方法模式回顾

- 工厂方法模式是创建型模式中的一种。
- 它的意图是定义一个用于创建对象的接口，让子类决定实例化哪一个类。
- 工厂方法模式的结构中，包括抽象工厂角色、具体工厂角色、抽象产品角色和具体产品角色。

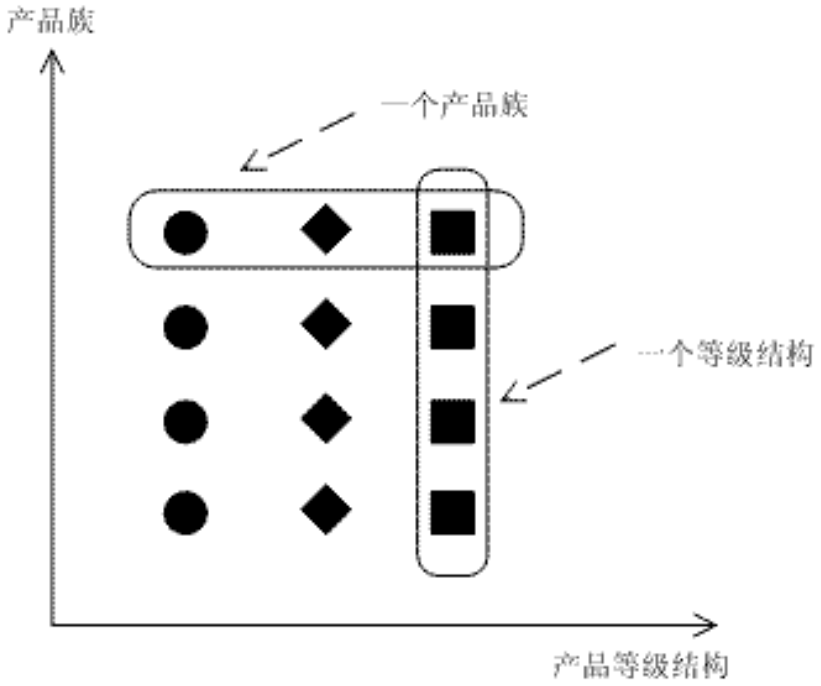
本章目标

- 了解抽象工厂模式的本质
- 理解抽象工厂模式的结构
- 掌握抽象工厂模式的适用性
- 掌握抽象工厂模式在设计中的应用

抽象工厂模式

- 抽象工厂模式的结构
- 抽象工厂模式与工厂模式的区别
- 在设计中如何应用抽象工厂模式

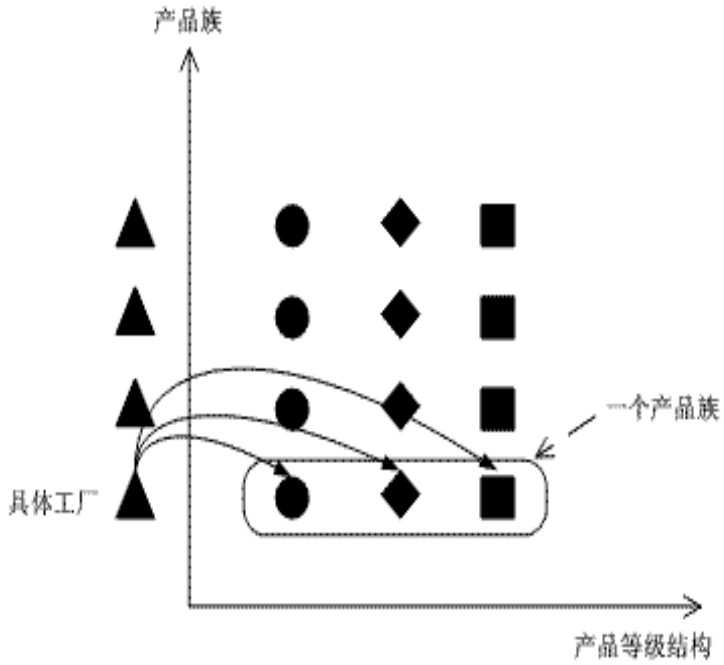
抽象工厂模式的本质



- 为了方便引进抽象工厂模式，引进一个新概念：产品族（Product Family）。所谓产品族，是指位于不同产品等级结构，功能相关联的产品组成的家族。
- 左图中一共有四个产品族，分布于三个不同的产品等级结构中。只要指明一个产品所处的产品族以及它所属的等级结构，就可以唯一的确定这个产品。

抽象工厂模式的本质

说明



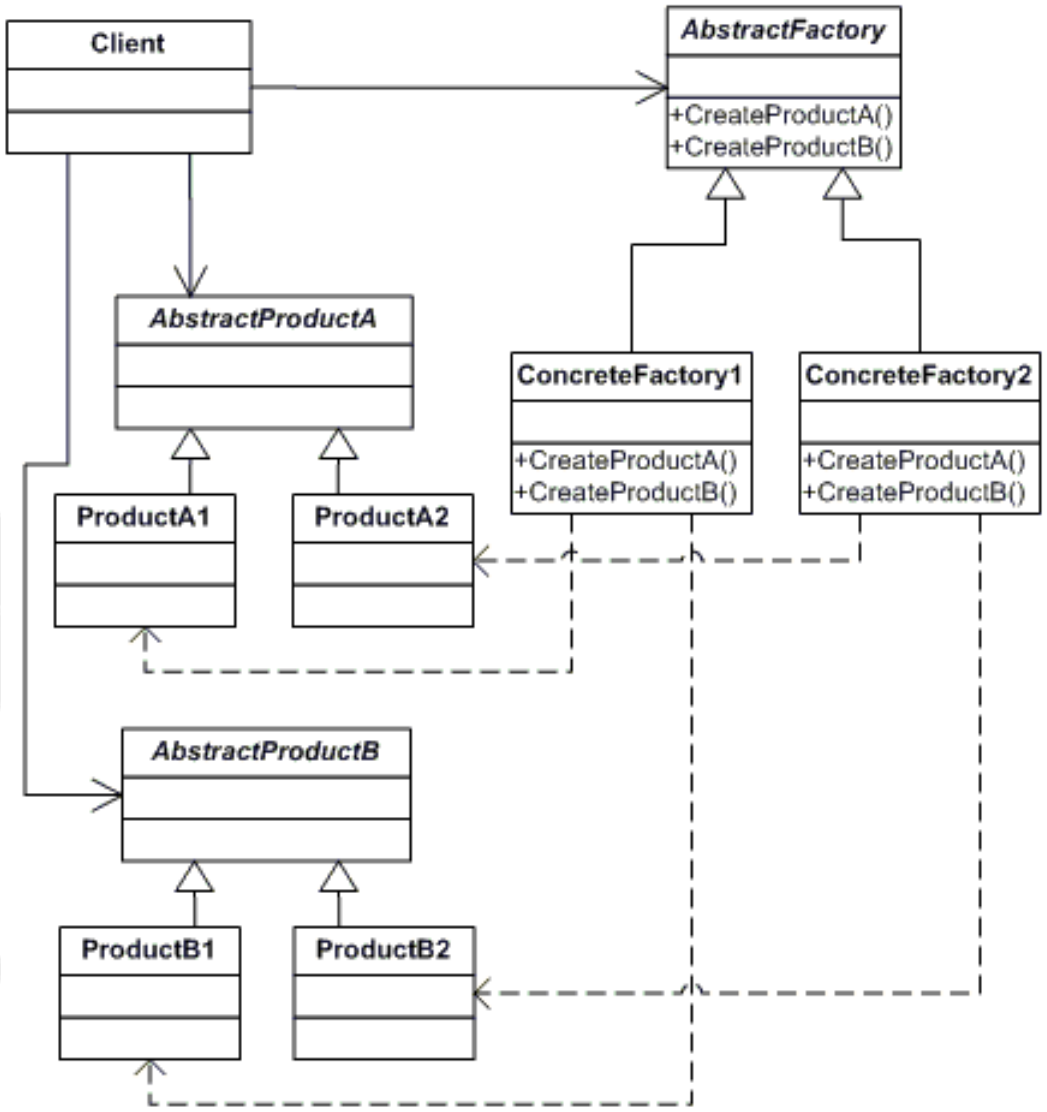
- 所谓的抽象工厂是指一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象。

抽象工厂模式的本质

- 简单地讲，我们可以认为抽象工厂模式是工厂方法模式的高级形式。在工厂对象中，抽象工厂模式需要创建几种产品，而每种产品的继承体系都是一致的。
- 这几种产品就被称之为“产品族（product family）”

抽象工厂模式的结构


说明


Client
AbstractFactory
+CreateProductA()
+CreateProductB()
AbstractProductA
ConcreteFactory1
+CreateProductA()
+CreateProductB()
ConcreteFactory2
+CreateProductA()
+CreateProductB()
ProductA1
ProductA2
AbstractProductB
ProductB1
ProductB2

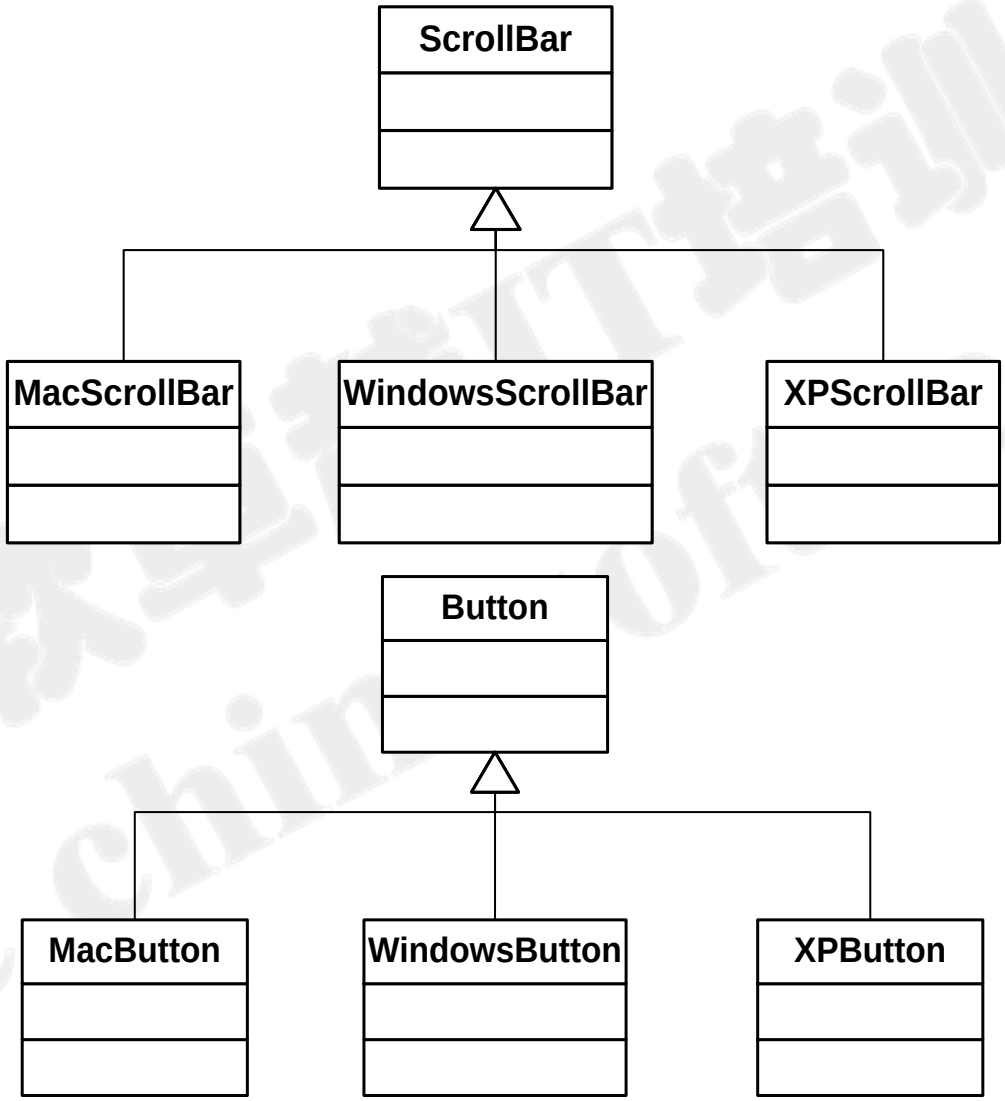
抽象工厂模式的适用性

- 在下列情况下可以使用抽象工厂模式：
  - 当一系列相关的产品对象的创建方式可能发生变化时。
  - 当一系列的产品对象必须保持一致性时。
  - 当创建的产品族中不会需要新的产品对象时。

仿Windows界面的工具软件

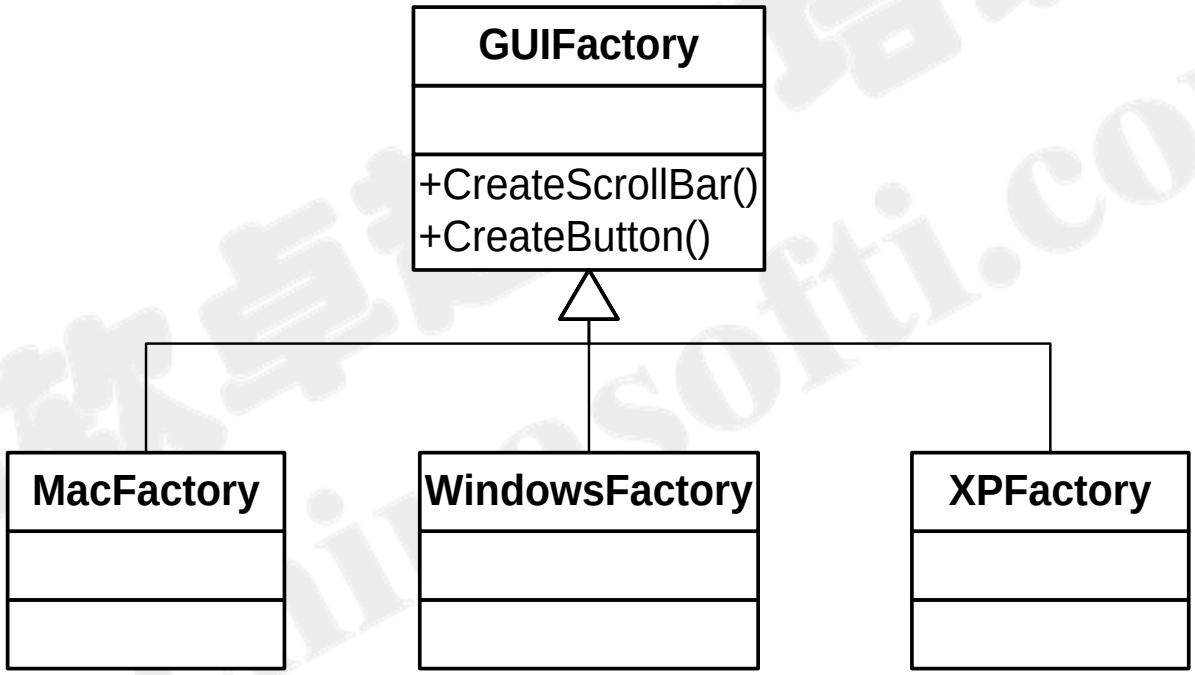
- 设计一个仿Windows界面的工具软件。我们需要创建一些组成窗口的对象。例如ScrollBar, Button, Menu等。同时，该工具能支持多种操作系统，例如Macintosh，Windows，XP等。也就是说，它们的GUI元素可能会有多种风格。
- 例如，对于ScrollBar而言，就有MacScrollBar，WindowsScrollBar, XPScrollBar等。

使用抽象工厂模式



产品的类结构

使用抽象工厂模式



工厂的类结构

仿Windows界面的工具软件

- 利用Abstract Factory模式，将ScrollBar, Button等对象的创建封装起来，形成抽象工厂类GUIFactory。则之前的sBar对象的创建则相应修改为：

```
ScrollBar sBar = guiFactory.CreateScrollBar();
```

- 注意，以这种方式对ScrollBar对象进行创建时，是与具体的Mac，Windows，XP无关的。即：这种设计与具体的实现无关，它是抽象的。而所谓创建的变化，则完全被封装到guiFactory对象中了。

```
GUIFactory guiFactory = new XPFactory();
```

总结

- 抽象工厂模式是创建型模式中的一种。
- 它的意图是定义一个用于创建一组对象的接口，让子类决定实例化哪一个类。
- 抽象工厂模式的结构中，包括抽象工厂角色、具体工厂角色、抽象产品角色和具体产品角色。

思考题

- 抽象工厂模式与工厂方法模式的区别？

作业

需求：

- 1、创建一个报表组件，它
  - 1）可以存储报表数据；
  - 2）设置报表格式；
  - 3）处理报表，例如导出数据文件等；
- 2、支持多种报表系统，例如水晶报表Crystal、用友华表等Cell；
- 3、如何设计一个可扩展性强的报表组件？



设计模式

建造者(Builder)模式

抽象工厂模式回顾

- 抽象工厂模式是创建型模式中的一种。
- 它的意图是定义一个用于创建一组对象的接口，让子类决定实例化哪一个类。
- 抽象工厂模式的结构中，包括抽象工厂角色、具体工厂角色、抽象产品角色和具体产品角色。

本章目标

- 了解建造者模式的本质
- 理解建造者模式的结构
- 掌握建造者模式在设计中的应用

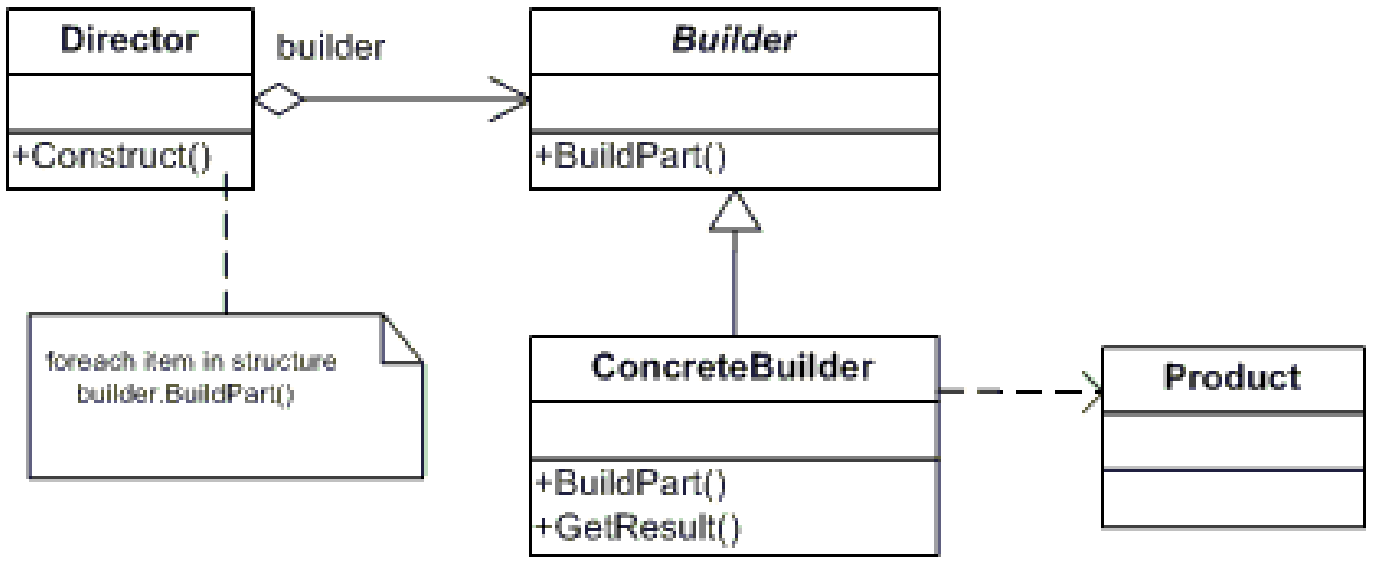
建造者模式

- 建造者模式与工厂方法模式的区别
- 在设计中如何应用建造者模式

建造者模式的本质

- 通过Builder模式可以封装产品的内部结构，客户端无须知道产品内部组成的细节。
- 创建的过程被抽象化，便于扩展。
- 使用Builder模式的好处：
  - 简化对象结构的构造；
  - 简化创建复杂对象的客户代码；
  - 解除客户代码与产品部件之间的耦合关系；

建造者模式的结构



建造者模式的角色

- 抽象建造者：指定创建产品零部件的抽象接口；
- 建造者：
  - 构建产品的零部件；
  - 组装产品；
  - 提供获得产品对象的接口；
- 指导者（Director）：通过建造者构造对象
- 产品（Product）

建造者模式的实现

```
abstract class Builder
{
  public abstract void BuildPartA();
  public abstract void BuildPartB();
  public abstract Product GetResult();
}
class Product
{
  private List<string> _parts = new List<string>();

  public void Add(string part)
  {
    _parts.Add(part);
  }

  public void Show()
  {
    Console.WriteLine("\nProduct Parts -------");
    foreach (string part in _parts)
      Console.WriteLine(part);
  }
}
}
```

建造者模式的实现

```
class ConcreteBuilder1 : Builder
{
  private Product _product = new Product();

  public override void BuildPartA()
  {
    _product.Add("PartA");
  }

  public override void BuildPartB()
  {
    _product.Add("PartB");
  }

  public override Product GetResult()
  {
    return _product;
  }
}
```

建造者模式的实现

```
class ConcreteBuilder2 : Builder
{
  private Product _product = new Product();

  public override void BuildPartA()
  {
    _product.Add("PartX");
  }

  public override void BuildPartB()
  {
    _product.Add("PartY");
  }

  public override Product GetResult()
  {
    return _product;
  }
}
```

建造者模式的实现

```
class Director
{
  // Builder uses a complex series of steps
  public void Construct(Builder builder)
  {
    builder.BuildPartA();
    builder.BuildPartB();
  }
}

//客户端
Director director = new Director();

Builder b1 = new ConcreteBuilder1();
Builder b2 = new ConcreteBuilder2();

director.Construct(b1);
Product p1 = b1.GetResult();
p1.Show();

director.Construct(b2);
Product p2 = b2.GetResult();
p2.Show();
```

总结

- 建造者模式是创建型模式中的一种。
- 它的意图是将一个复杂对象的创建与表示分离，使得同样的建造过程可以创建不同的表示。
- 建造者模式还可以用来简化复杂对象的创建。
- 建造者模式的结构中，包括抽象建造者角色、具体建造者角色、指导者角色和产品角色。

思考题

- 建造者模式与工厂方法模式的区别？

作业

- 假设生产车间（Shop）需要建造汽车（Car）和摩托车（MotorCycle）。汽车和摩托车都具有零部件：车架（Frame）、引擎（Engine）、车轮（Wheel）和车门（Door）。
- 汽车的引擎为2500cc，车轮为4个，车门为2；摩托车的引擎为500cc，车轮为2，车门为0。
- 提示：汽车的零部件可以用类的索引器来表示；
- 提示：所有零部件信息都可以用字符串表示。例如，生产出来的汽车显示信息为：

Vehicle Type: Car
Frame : Car Frame
Engine : 2500 cc
#Wheels: 4
#Doors : 2

设计模式

单例(Singleton)模式

建造者模式回顾

- 建造者模式是创建型模式中的一种。
- 它的意图是将一个复杂对象的创建与表示分离，使得同样的建造过程可以创建不同的表示。
- 建造者模式还可以用来简化复杂对象的创建。
- 建造者模式的结构中，包括抽象建造者角色、具体建造者角色、指导者角色和产品角色。

本章目标

- 了解单例模式的作用和用途
- 理解单例模式的实现
- 掌握单例模式在设计中的应用

单例模式

- 作用：创建一个唯一的实例
- 用途：常用在无状态对象中，例如Façade类、工厂类
- 注意：
  - 考虑并发访问时的线程安全性；
  - 单例对象尽量是只读的（不变对象）；
  - 尽量避免将资源对象做成单例对象，例如数据库连接及其他稀有资源。原因：
  - 某些资源不止需要一个；
  - 单例采用静态方式存储类实例，无法及时释放资源

单例模式的实现

```
public class Singleton
{
    private Singleton(){};
    private static Singleton instance = null;
    public static Singleton Instance()
    {
        if (instance == null) {instance = new Singleton();}
        return instance;
    }
}
```

没有考虑并发的情况

单例模式的实现

```
public  class Singleton
{
    public static Singleton Instance =
        new Singleton();
    private Singleton() {}
}
```

考虑并发的情况

单例模式模拟实现负载均衡

- 通过该案例可以学到：
  - 如何在单例对象中维持集合对象；
  - 另外一种保证线程安全的实现方式；



设计模式

适配器(Adapter)模式

创建型模式回顾

- 创建型模式的本质是封装创建的变化。
- 常见的创建型模式包括：工厂方法模式、抽象工厂模式、建造者模式、单例模式。
- 工厂方法模式和抽象工厂模式，通过建立抽象的工厂类，封装未来对象的创建所引起的可能变化。
- 建造者模式则是对对象内部的创建进行封装，由于细节对抽象的可替换性，使得将来面对对象内部创建方式的变化，可以灵活的进行扩展或替换。

本章目标

- 理解结构型模式的本质
- 了解适配器模式的本质
- 理解适配器模式的结构
- 掌握适配器模式在设计中的应用

- 类的适配器模式与对象的适配器模式的区别

结构型模式的本质

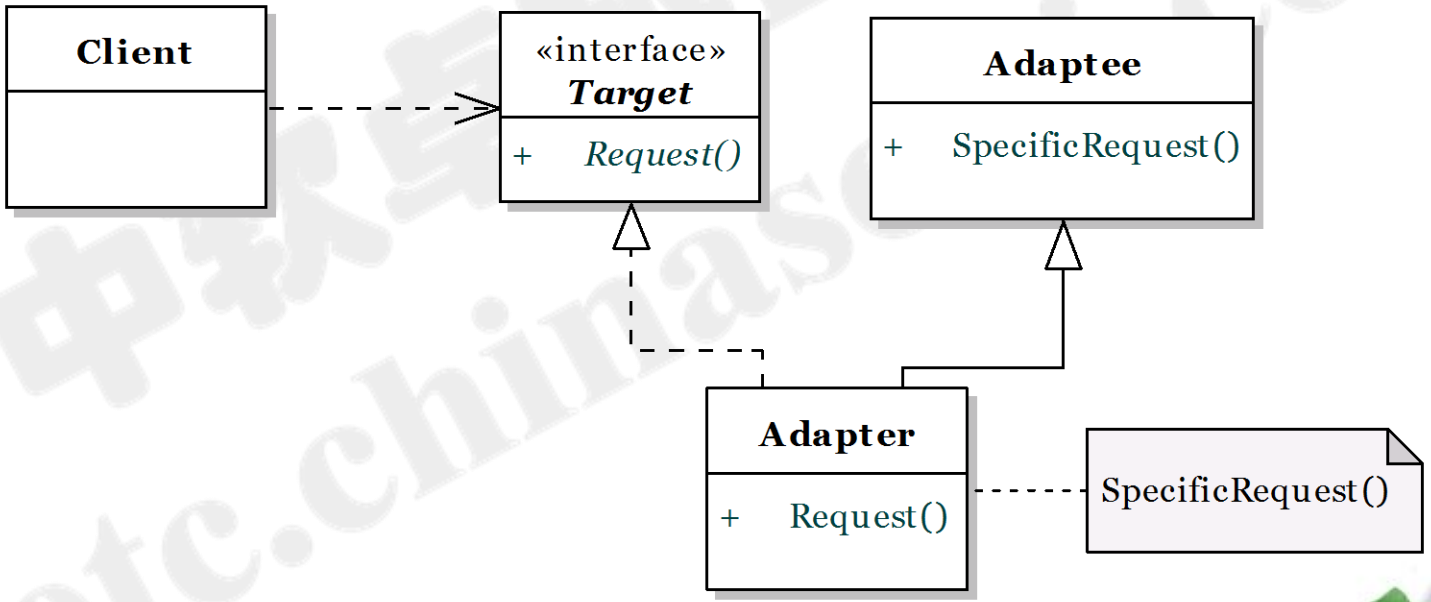
- 结构型模式关注的是对象之间组合的方式。本质上说，如果对象结构可能存在变化，主要在于其依赖关系的改变。当然对于结构型模式来说，处理变化的方式不仅仅是封装与抽象那么简单，还要合理地利用继承与聚合的方法，灵活地表达对象之间的依赖关系。
- 例如，装饰器模式，描述的就是对象间可能存在的多种组合方式，这种组合方式是一种装饰者与被装饰者之间的关系，因此封装这种组合方式，抽象出专门的装饰对象显然是“封装变化”的体现。同样地，桥接模式封装的则是对象实现的依赖关系，而合成模式所要解决的则是对象间存在的递归关系。

适配器模式的本质

- 适配器模式把一个类的接口变换成客户端所期待的另一种接口，从而使原本接口不匹配而无法在一起工作的两个类能够在一起工作。
- 如何理解Adapter模式？
  - 变压器（Adapter），变压器把一种电压变换成另一种电压。
  - 货物的包装过程：被包装的货物的真实样子被包装所掩盖和改变，因此有人把这种模式叫做包装（Wrapper）模式。事实上，大家经常写很多这样的Wrapper类，把已有的一些类包装起来，使之有能满足需要的接口。(后面介绍的装饰器模式是另一种包装)
- 适配器模式的两种形式：
  - 类的适配器模式；
  - 对象的适配器模式。

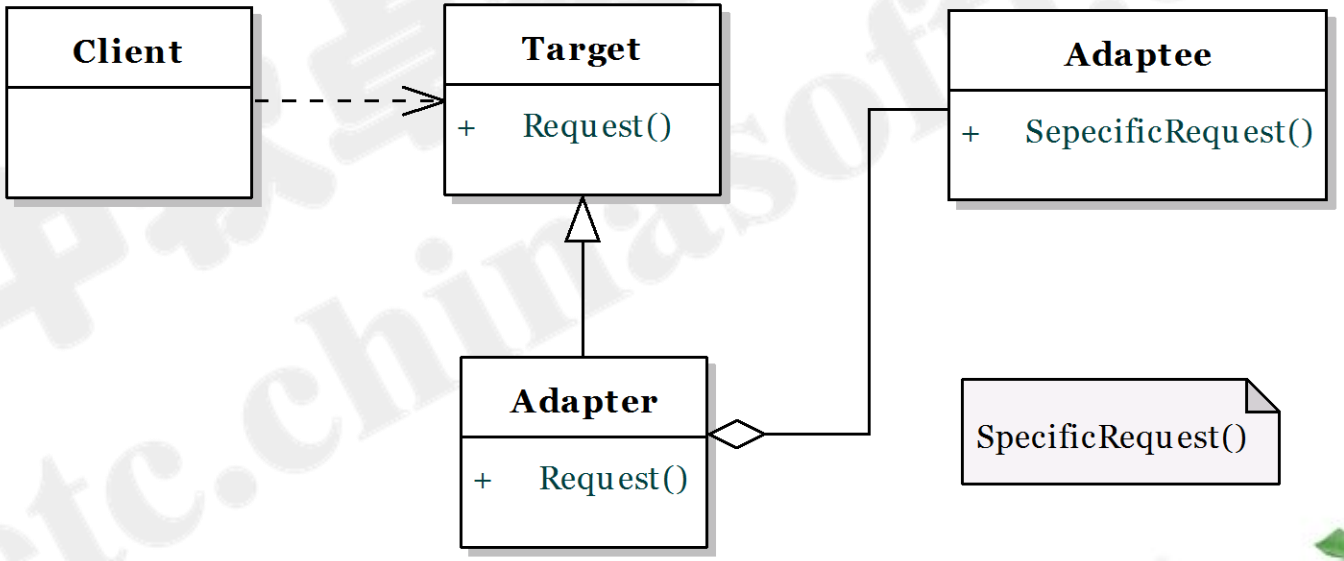
类的适配器模式

- 目标（Target）角色：这是客户所期待的接口。因为C#不支持多继承，所以Target必须是接口，不可以是类。
- 源（Adaptee）角色：需要适配的类。
- 适配器（Adapter）角色：把源接口转换成目标接口。这一角色必须是类。



对象的适配器模式

- 目标（Target）角色：这是客户所期待的接口。目标可以是具体的或抽象的类，也可以是接口。
- 源（Adaptee）角色：需要适配的类。
- 适配器（Adapter）角色：通过在内部包装（Wrap）一个Adaptee对象，把源接口转换成目标接口。



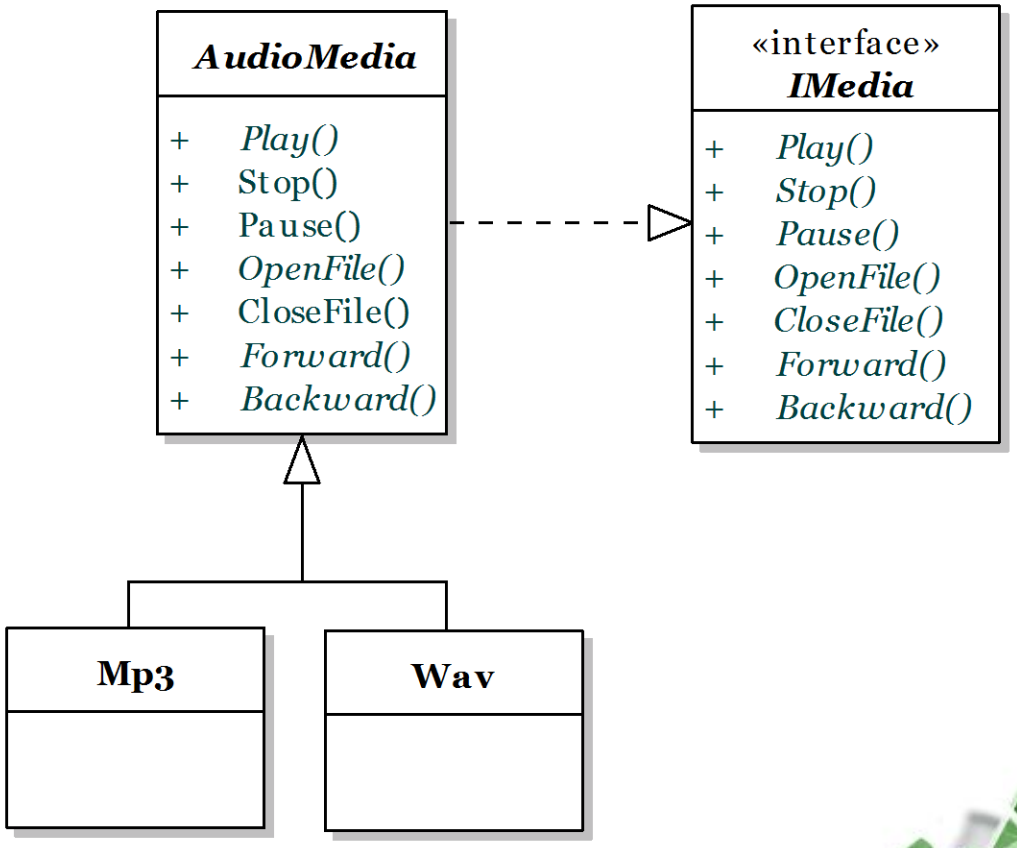
应用Adapter模式的前提

- 两个类所做的事情相同或相似，但是具有不同的接口；
- 如果类共享同一个接口，客户代码会更简单、更直接、更紧凑；
- 无法轻易改变其中一个类的接口，因为它是第三方类库的一部分，或者它是一个已经被其他客户代码广泛使用的框架的一部分，或者无法获得源代码。
- 遵循“面向接口编程”，否则将难以应用适配器模式，即设计时需要遵循“依赖倒置原则”。

媒体播放器

一案例

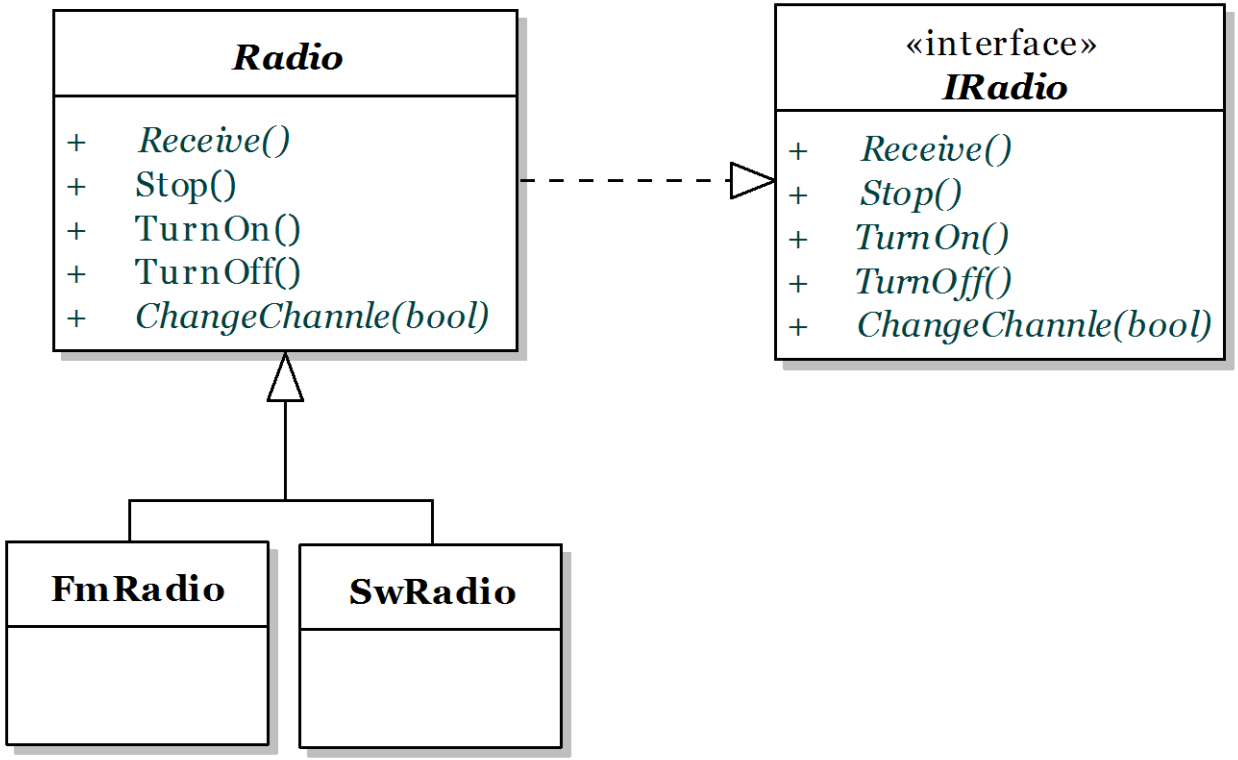
- 我们已经为客户开发了一个媒体播放器，能够播放音频文件。结构如右所示：



Radio仿真软件

一案例

- 同时，公司还开发了另外一款产品，能够通过该软件收听广播，它的结构如右所示：



需求分析

- 假定这两款产品分别为：MediaPlayer和RadioPlayer。客户的需求是将Radio的功能合并到MediaPlayer中。
- 怎么办？最简单的办法是对MediaPlayer模块动手术，修改IMedia接口，将Radio的功能添加进去。问题是，这两个模块已经是被封装好的，客观条件是不允许我们对其进行修改了。即使修改，工作量也不足以在规定时间内完成。

设计分析

- Adapter模式能够吸收我们所需要的接口和所有接口之间的不同。如本例中的IMedia接口与IRadio接口。利用Adpater模式，我们可以建立一个单独的Adapter模块。
- Adapter模式分为两种形式：类的Adapter模式和对象的Adapter模式。前者利用继承或实现的方式，而后者则利用合成的方式。

能否适配？

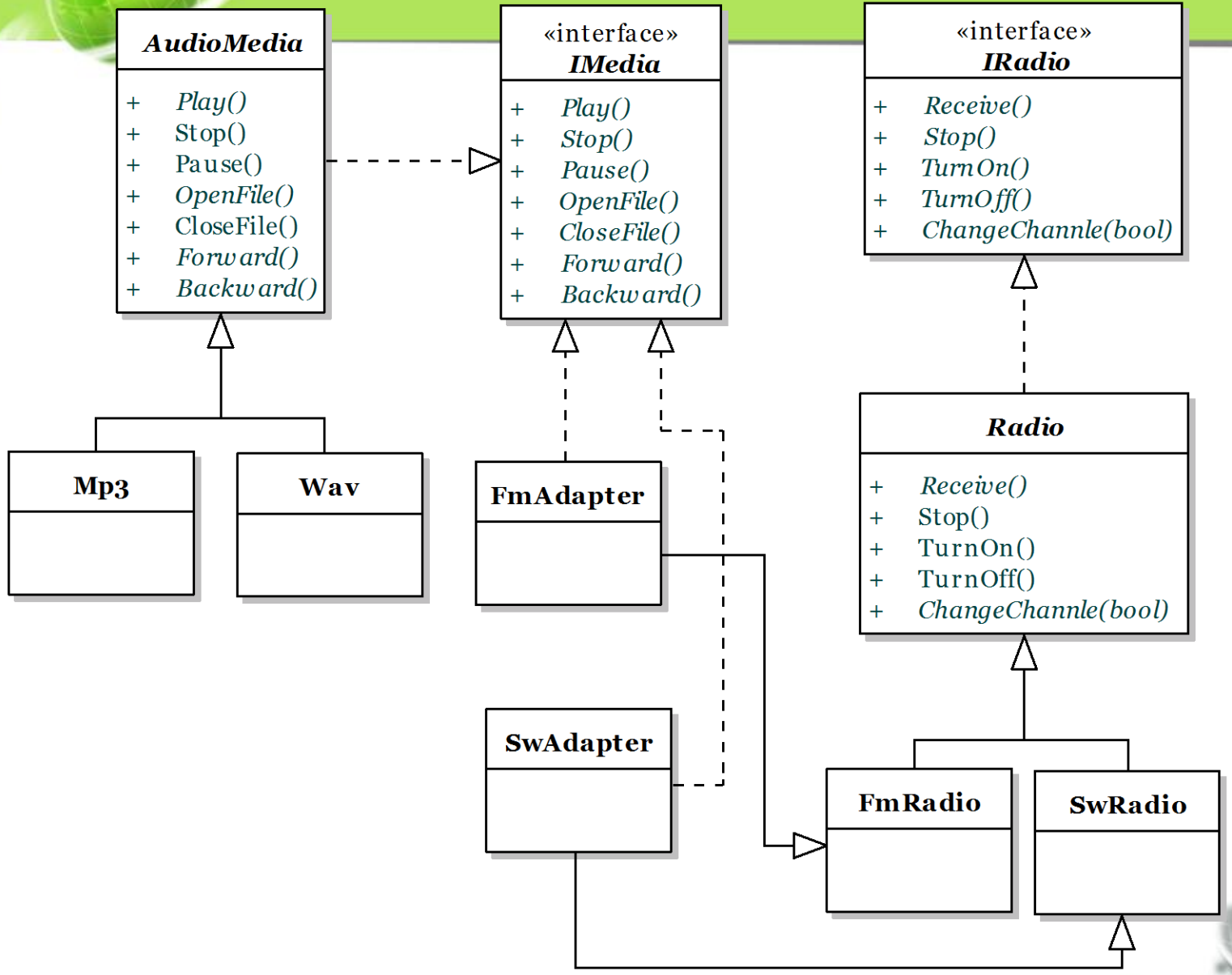
- 在使用Adapter模式时，前提条件是两者必须是能够适配的。如IMedia和IRadio之间：

IMedia	IRadio
Play()	Receive()
Stop()	Stop()
OpenFile()	TurnOn()
CloseFile()	TurnOff()
Forward()	ChangeChannel(true)
Back()	ChangeChannel(false)

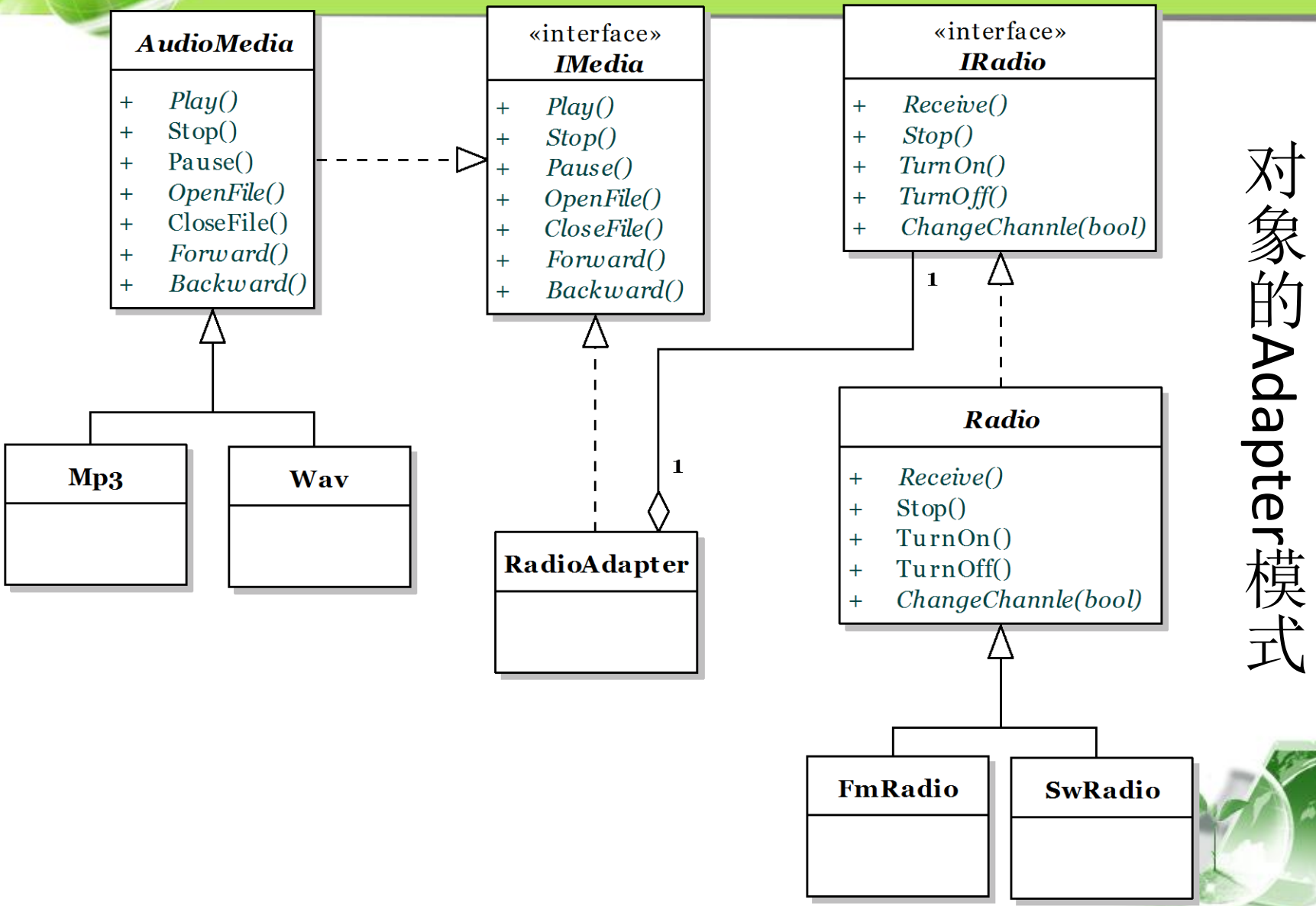
辨别角色

- 被适配对象(第三方接口)：IRadio
- 适配对象(期望接口)：IMedia
- 当方法数量不匹配时：
  - 被适配对象(IRadio)不存在的方法，则在适配器对象(Adapter)中实现为空；
  - 被适配对象多出的方法，则适配器对象仍然保留，接口变宽。这属于扩展，因而符合开放-封闭原则。

类的Adapter模式



对象的Adapter模式



选择Adapter模式

- 类的Adapter模式不仅实现了适配接口，同时还继承了被适配类。如果被适配类的接口宽于适配接口，则优先选择类的Adapter模式；
- 如果适配器类需要从多个对象中提取信息，那么就应当使用对象的Adapter模式，因为对象的组合关系不会受到继承的限制；
- 其余情况，优先选择对象的Adapter模式。

总结

- 适配器模式是结构型模式中的一种。
- 它的意图是把一个类的接口变换成客户端所期待的另一种接口，从而使原本接口不匹配而无法在一起工作的两个类能够在一起工作。。
- 适配器模式分为类的适配器模式和对象的适配器模式。

作业

- 企业开发一个电子商务网站，使用了自己开发的付费服务，用于银行储蓄卡和信用卡的网上支付。随着该企业和银行的合作协议的终止，该付费服务也将失效。企业决定采用第三方的付费服务。然而，第三方的付费服务与原有服务的接口并不一致。那么，我们应该怎么设计才能将代码的修改降到最少？
- 假设自身系统的付费服务接口定义如下：

```
public interface IPayService {
    bool Pay(Account account, Money amount);
}
```

- 第三方付费服务的接口定义如下：

```
public interface ICommonPayService {
    bool Pay(int BankId, Account account, double amount);
}
```

实现类为：CommonPayService

设计模式

装饰器(Decorator)模式

适配器模式回顾

- 适配器模式是结构型模式中的一种。
- 它的意图是把一个类的接口变换成客户端所期待的另一种接口，从而使原本接口不匹配而无法在一起工作的两个类能够在一起工作。。
- 适配器模式分为类的适配器模式和对象的适配器模式。

本章目标

- 了解装饰器模式的本质
- 理解装饰器模式的结构
- 掌握装饰器模式在设计中的应用

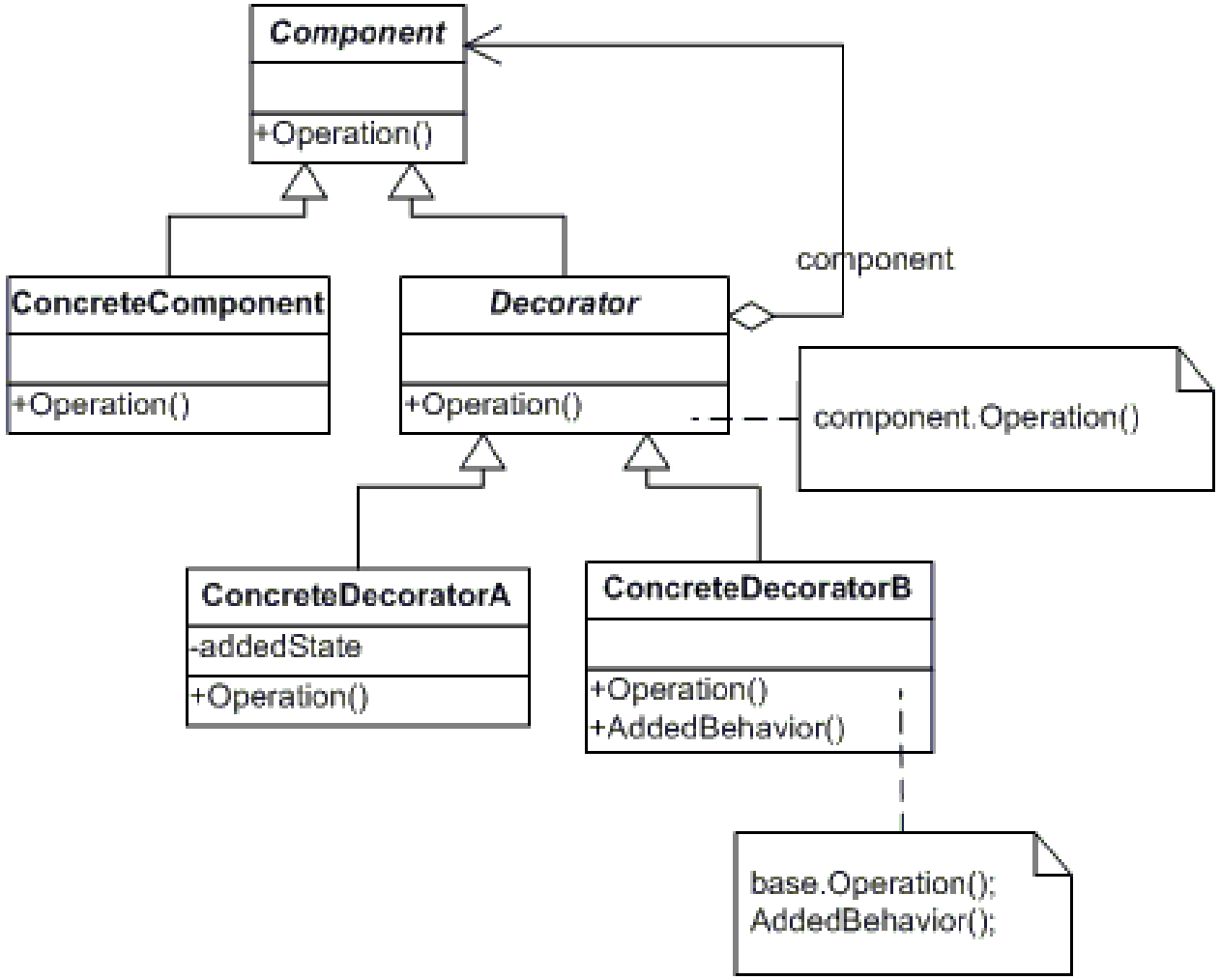
装饰器模式

- 理解装饰器模式的适用场景
- 理解装饰器模式的执行序列

装饰器模式的本质

- Decorator模式以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案。
- Decorator模式有效地利用了对象的合成/聚合与继承。
- 调用者通过Decorator模式可以动态地为一个对象添加多个职责，就像油漆工一样，不断地在墙面上粉刷不同颜色的油漆。

装饰器模式的结构



何时使用装饰器模式

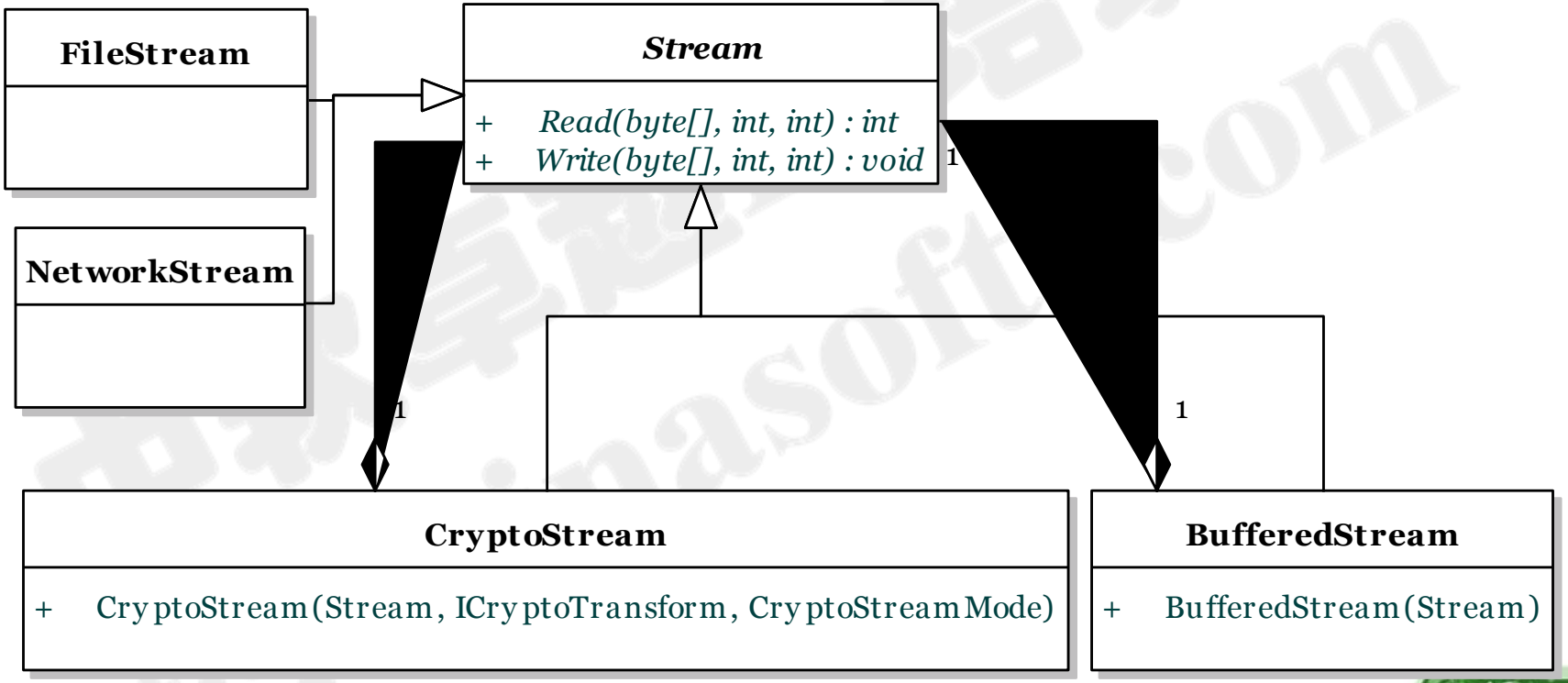
- 在以下情况下应当使用Decorator模式：
  - 需要扩展一个类的功能，或给一个类增加附加责任。
  - 需要动态地给一个对象增加功能，这些功能可以再动态地撤销。
  - 需要增加由一些基本功能的排列组合而产生的非常大量的功能，从而使继承关系变得不现实。

考虑.NET对流的处理

- 我们需要实现对流的处理。流包括文件流FileStream和网络流NetworkStream。
- 对于调用者而言，可能会根据实际的情况，为流的处理（Read和Write）提供缓存（Buffer）或加密(Crypto)功能。
- 我们应该如何设计？

对流的处理

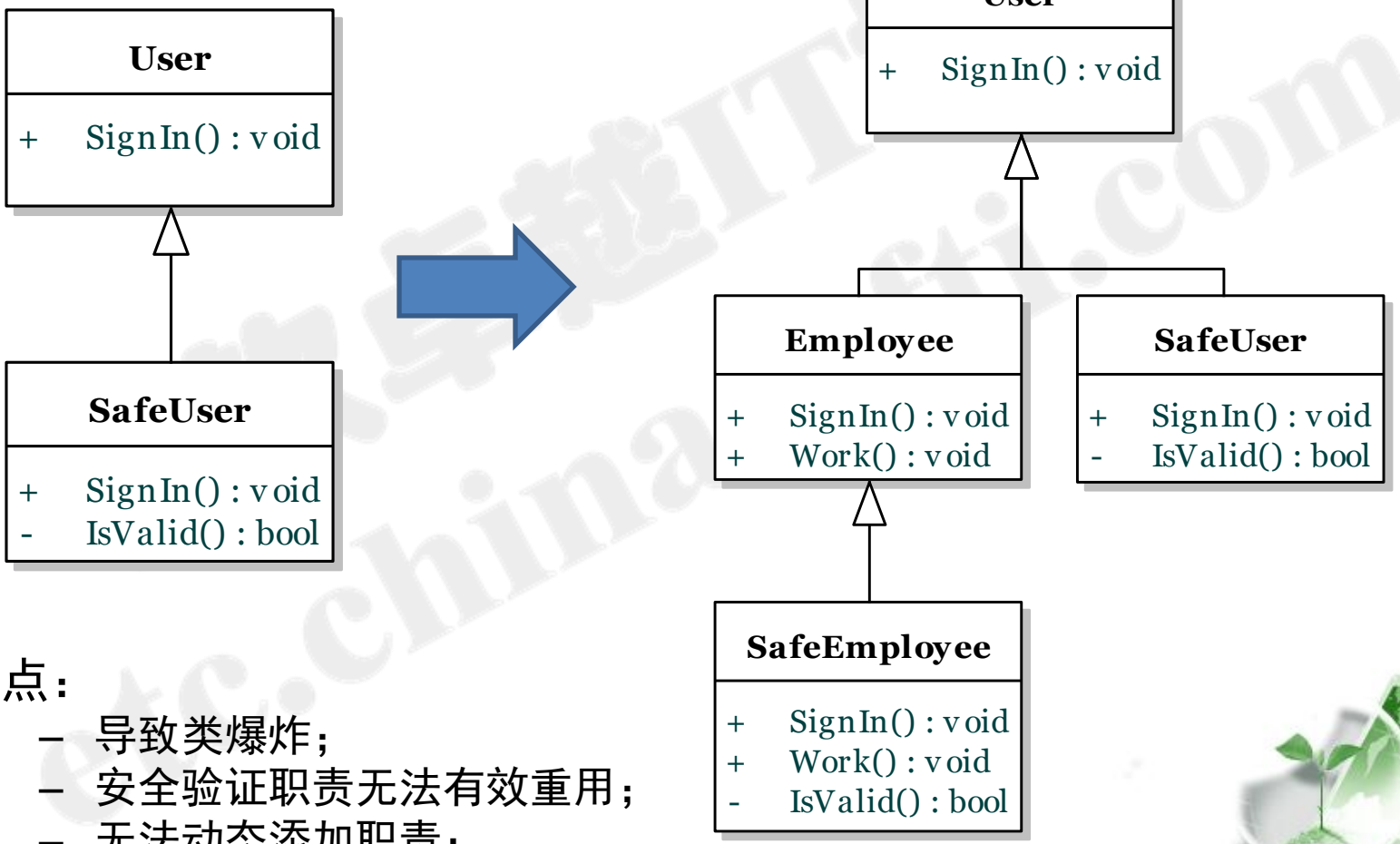
思考


FileStream
Stream
+ Read(byte[], int, int) : int
+ Write(byte[], int, int) : void
NetworkStream
1
1
1
CryptoStream
+ CryptoStream(Stream, ICryptoTransform, CryptoStreamMode)
BufferedStream
+ BufferedStream(Stream)

案例：用户登录

- 我们开发了一个Web Portal，此时需要对注册用户进行管理。
- 需求会随着业务的复杂化而逐渐变化。最初，我们需要为用户的登录提供权限验证。接着，我们需要为用户的登录提供日志记录功能。
- 然后，客户要求区别员工和注册用户，为他们提供不同的权限验证功能和日志记录功能。

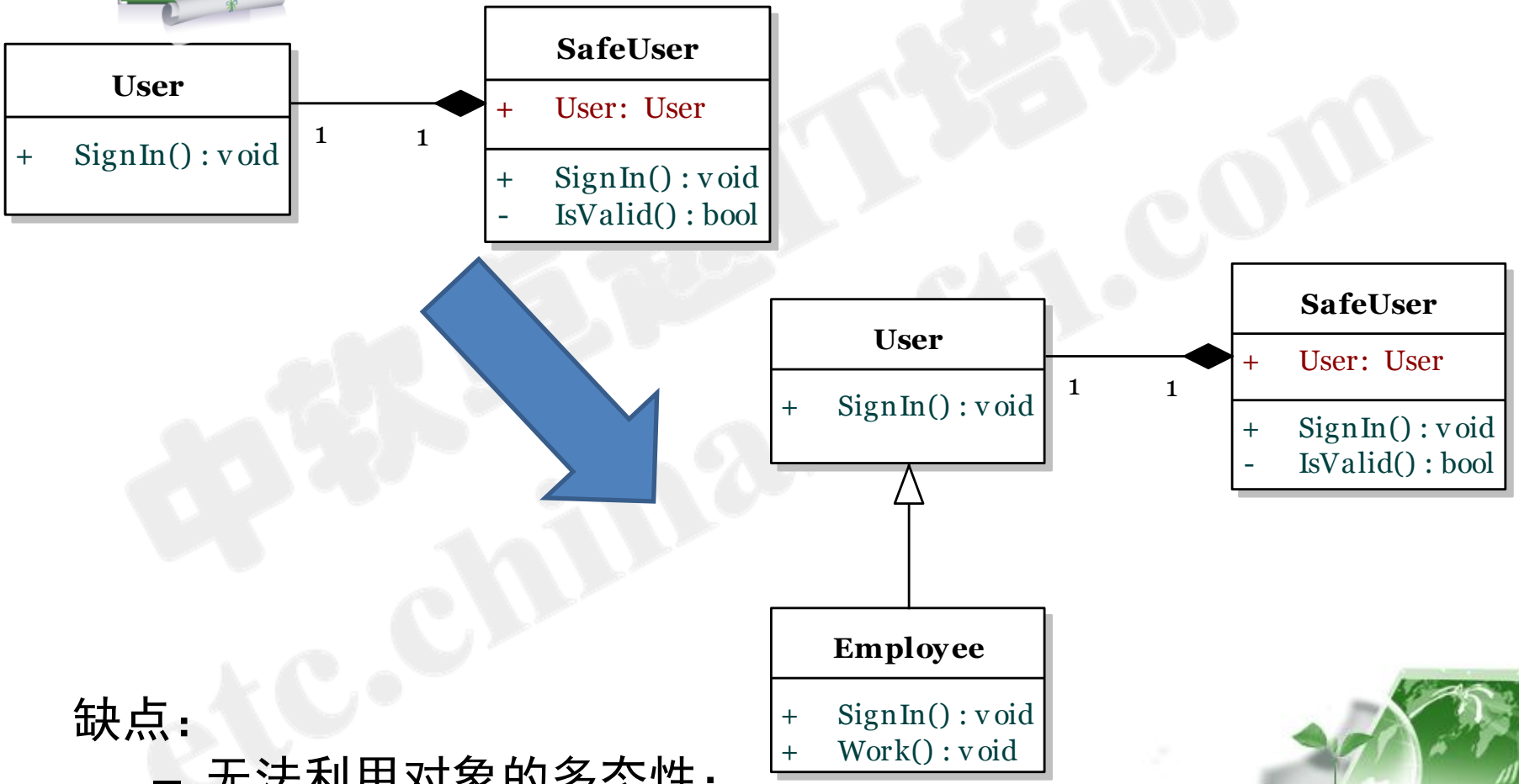
使用继承



缺点：

- 导致类爆炸；
- 安全验证职责无法有效重用；
- 无法动态添加职责；

使用组合

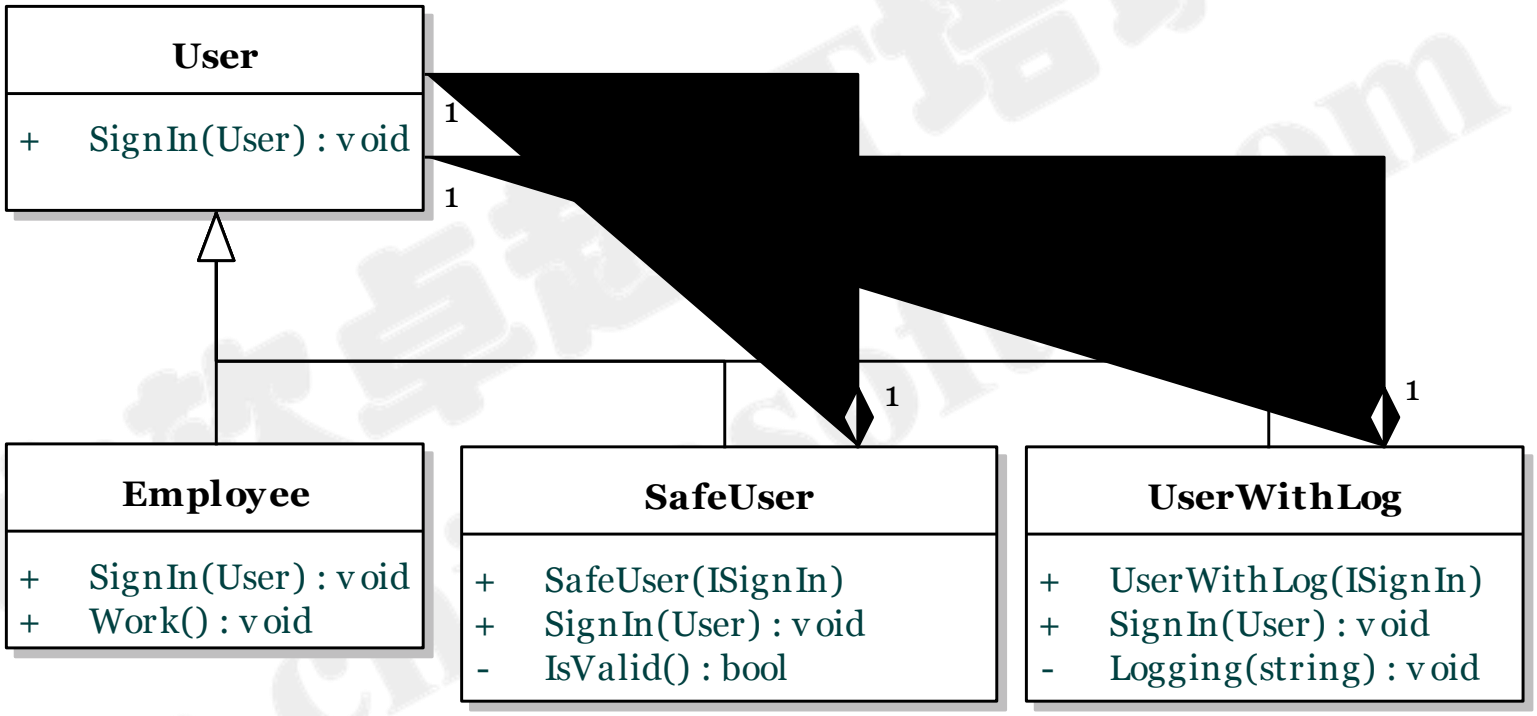


缺点：

- 无法利用对象的多态性；
- 无法动态添加职责；

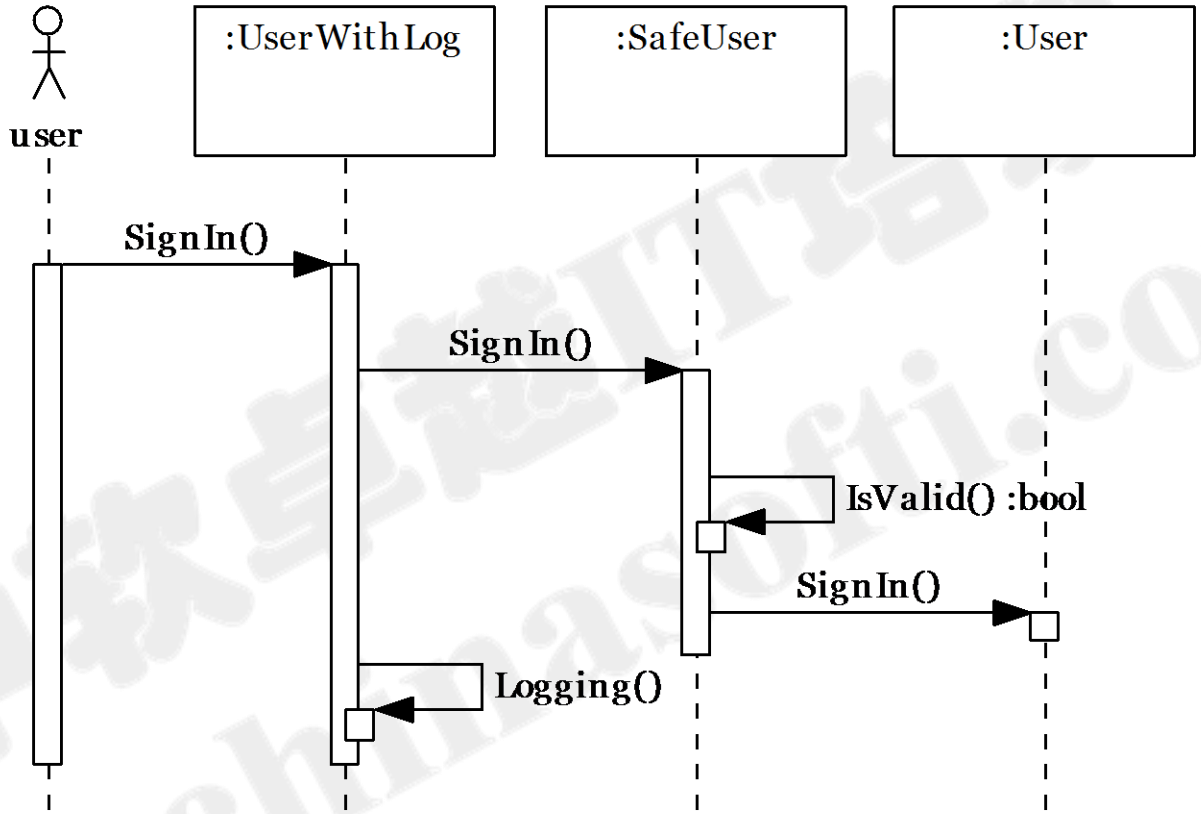
使用装饰器模式

案例


User
+ SignIn(User) : void
1
1
1
1
Employee
+ SignIn(User) : void
+ Work() : void
SafeUser
+ SafeUser(ISignIn)
+ SignIn(User) : void
- IsValid() : bool
UserWithLog
+ UserWithLog(ISignIn)
+ SignIn(User) : void
- Logging(string) : void

执行时序图

案例



执行：

```
User user = new UserWithLog(new SafeUser(new User("admin", "password")));
user.SignIn();
```

总结

- 装饰器模式是结构型模式中的一种。
- 装饰器模式关注的是如何为目标对象动态添加功能或职责。
- 装饰器模式巧妙地运用了继承与组合的设计思想。一方面它通过继承保持类型的多态性，使得被装饰器对象可以被动态替换；另一方面则通过组合的方式接收被装饰对象，从而能够重用被装饰对象的职责，以便于在该职责之上进行新功能的装饰。

作业

- 为图书（Book）、视频资料（Video）等图书馆资料动态添加职能，例如，显示借阅者清单如下：

  Video -----

  Director: Spielberg

  Title: Jaws

  # Copies: 21

  Playtime: 92

  borrower: Customer #1

  borrower: Customer #2

作业

- 图书和视频资料的抽象父类定义如下：

```
abstract class LibraryItem
 {
     private int _numCopies;

     public int NumCopies
     {
         get { return _numCopies; }
         set { _numCopies = value; }
     }

     public abstract void Display();
 }
```

作业

- 图书的定义如下：

```
class Book : LibraryItem
 {
  private string _author;
  private string _title;

  public Book(string author, string title, int numCopies)
  {
   this._author = author;
   this._title = title;
   this.NumCopies = numCopies;
  }
  public override void Display()
  {
   Console.WriteLine("\nBook ------ ");
   Console.WriteLine(" Author: {0}", _author);
   Console.WriteLine(" Title: {0}", _title);
   Console.WriteLine(" # Copies: {0}", NumCopies);
  }
 }
```

作业

- 视频资料的定义如下：

```
class Video : LibraryItem
 {
  private string _director;
  private string _title;
  private int _playTime;
  public Video(string director, string title,
    int numCopies, int playTime)
  {
    this._director = director;
    this._title = title;
    this.NumCopies = numCopies;
    this._playTime = playTime;
  }
  public override void Display()
  {
    Console.WriteLine("\nVideo ----- ");
    Console.WriteLine(" Director: {0}", _director);
    Console.WriteLine(" Title: {0}", _title);
    Console.WriteLine(" # Copies: {0}", NumCopies);
    Console.WriteLine(" Playtime: {0}\n", _playTime);
  }
 }
```



设计模式

合成(Composite)模式

装饰器模式回顾

- 装饰器模式是结构型模式中的一种。
- 装饰器模式关注的是如何为目标对象动态添加功能或职责。
- 装饰器模式巧妙地运用了继承与组合的设计思想。一方面它通过继承保持类型的多态性，使得被装饰器对象可以被动态替换；另一方面则通过组合的方式接收被装饰对象，从而能够重用被装饰对象的职责，以便于在该职责之上进行新功能的装饰。

本章目标

- 了解合成模式的本质
- 了解合成模式的分类
- 理解合成模式的结构

合成模式

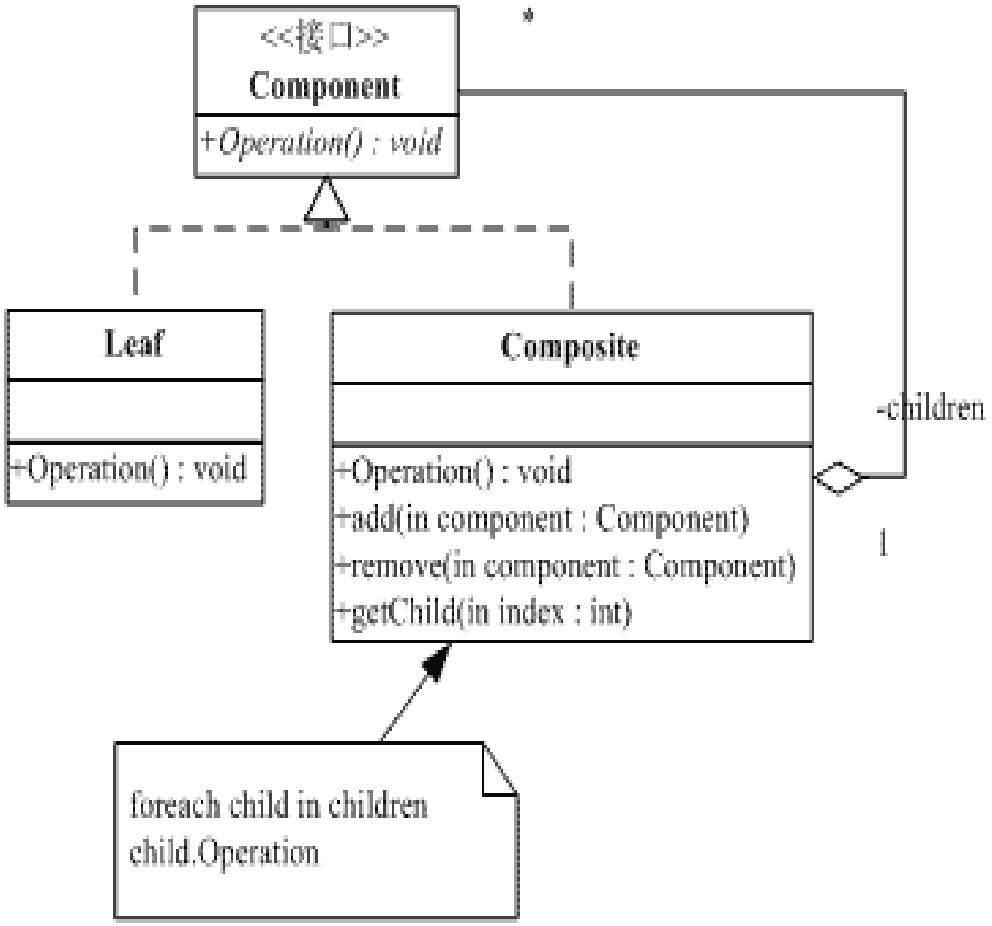
- 理解合成模式的本质

合成模式的本质

- 合成模式有时又叫做部分－整体模式（Part-Whole）。合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。合成模式可以使客户端将单纯元素与复合元素同等看待。可以认为复合元素为树枝，单纯元素为树叶。
- 合成模式体现了行为的高度抽象，即面对不同结构的对象，如果其行为的表现形式以及客户端的调用方式存在一致性，则在抽象层面上就可以视为一体。
- 合成模式的实现根据所实现接口的区别分为两种形式，分别称为安全模式和透明模式。

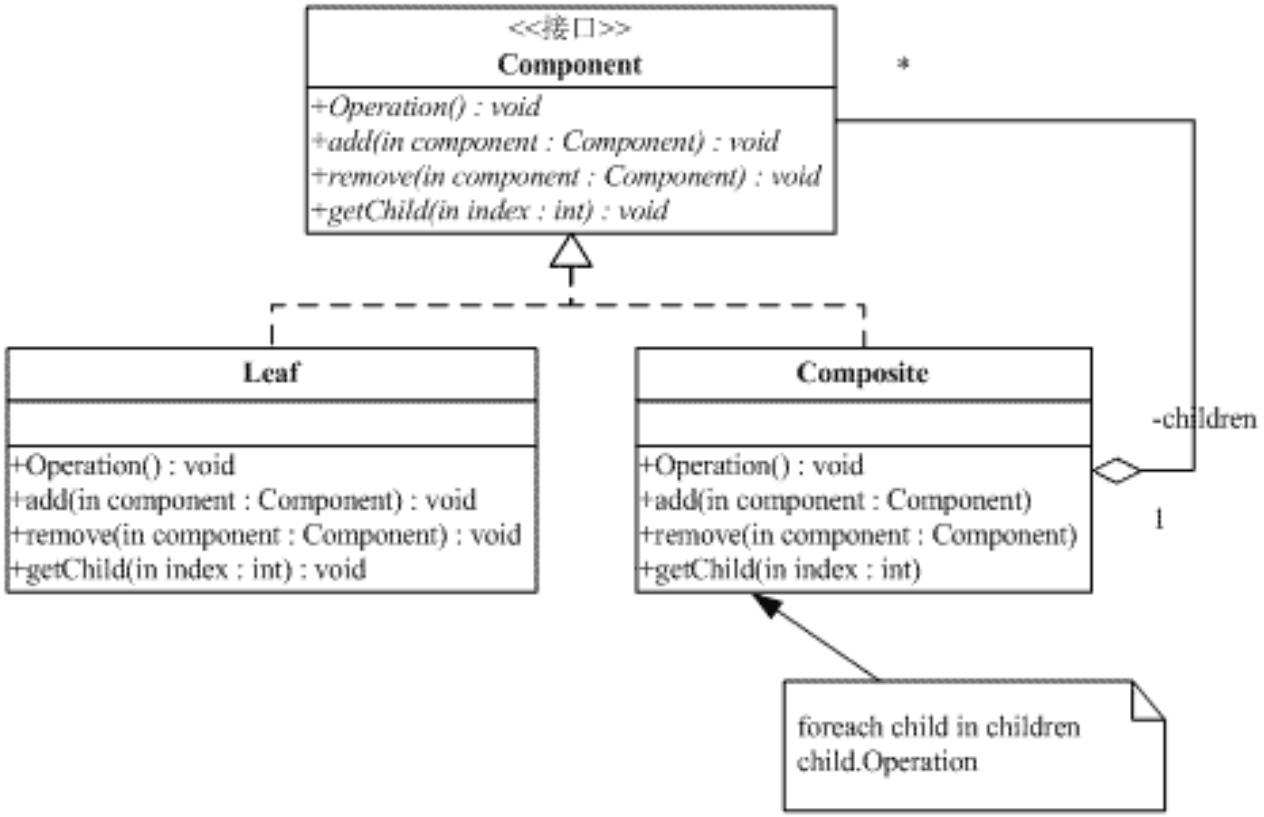
安全的合成模式

- 区别树枝对象和树叶对象，只有树枝对象才有管理子对象的能力，这样就避免了客户端调用时出现错误。缺点是树枝对象和树叶对象的接口不一致。



透明的合成模式

- 无论是树枝对象还是树叶对象，都具有统一的接口。不过在树叶对象中，如果客户端要调用管理子对象的方法时，应该考虑出现提示，或抛出异常。

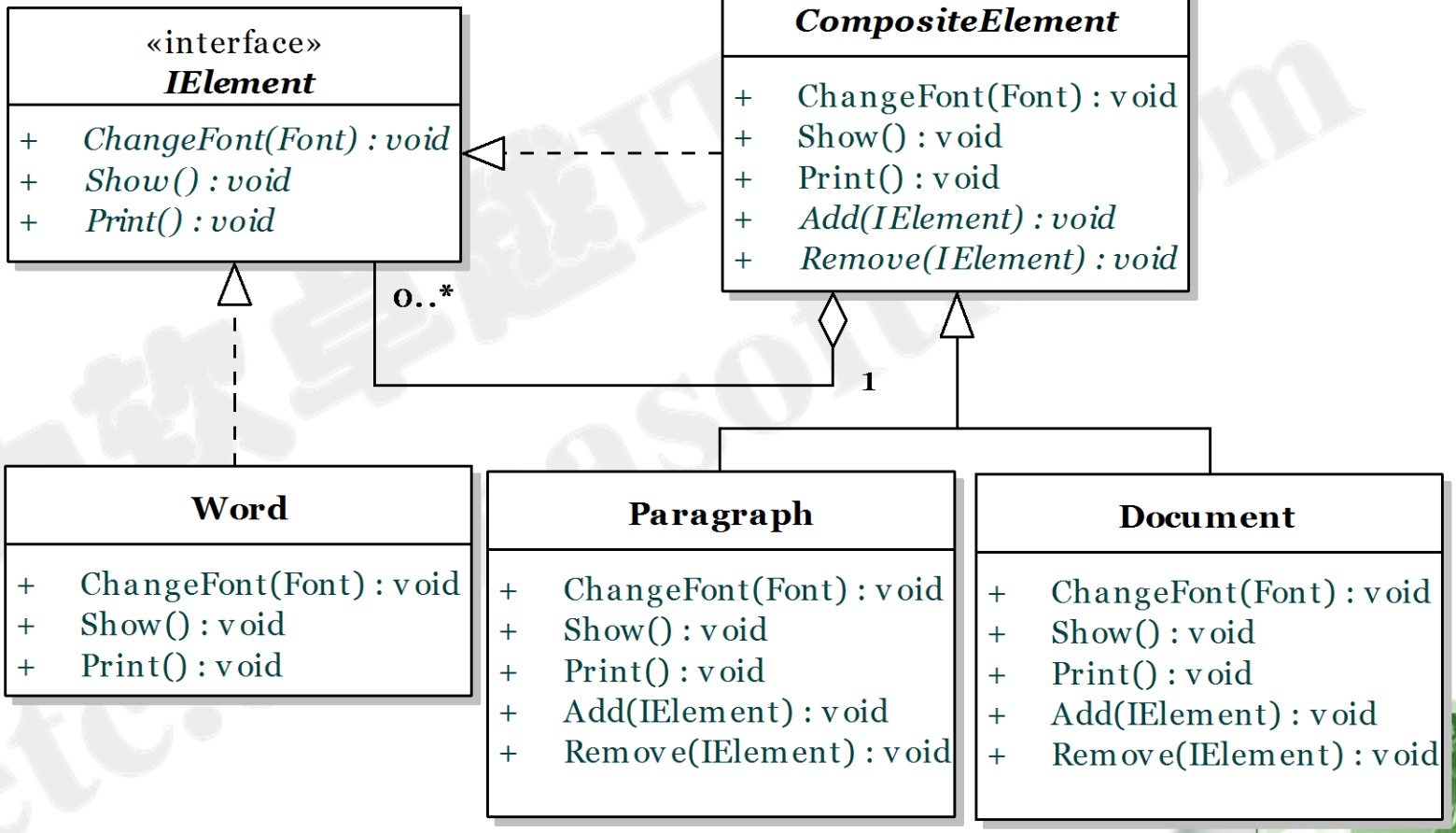


案例：文档编辑器

- 在文档编辑器中，要处理的对象包括单个的文字(Word)，以及由文字组成的段落(Paragraph)，乃至整篇文档(Document)。
- 这些对象均支持改变字体、显示和打印等操作。
- 设计它们的类结构。

文档编辑器的类结构

案例



总结

- 合成模式是结构型模式中的一种。
- 合成模式用于处理复合元素与单元素的对象结构。
- 合成模式体现了行为的高度抽象，即面对不同结构的对象，如果其行为的表现形式以及客户端的调用方式存在一致性，则在抽象层面上就可以视为一体。
- 合成模式的实现根据所实现接口的区别分为两种形式，分别称为安全模式和透明模式。

设计模式

代理(Proxy)模式

合成模式回顾

- 合成模式是结构型模式中的一种。
- 合成模式用于处理复合元素与单元素的对象结构。
- 合成模式体现了行为的高度抽象，即面对不同结构的对象，如果其行为的表现形式以及客户端的调用方式存在一致性，则在抽象层面上就可以视为一体。
- 合成模式的实现根据所实现接口的区别分为两种形式，分别称为安全模式和透明模式。

本章目标

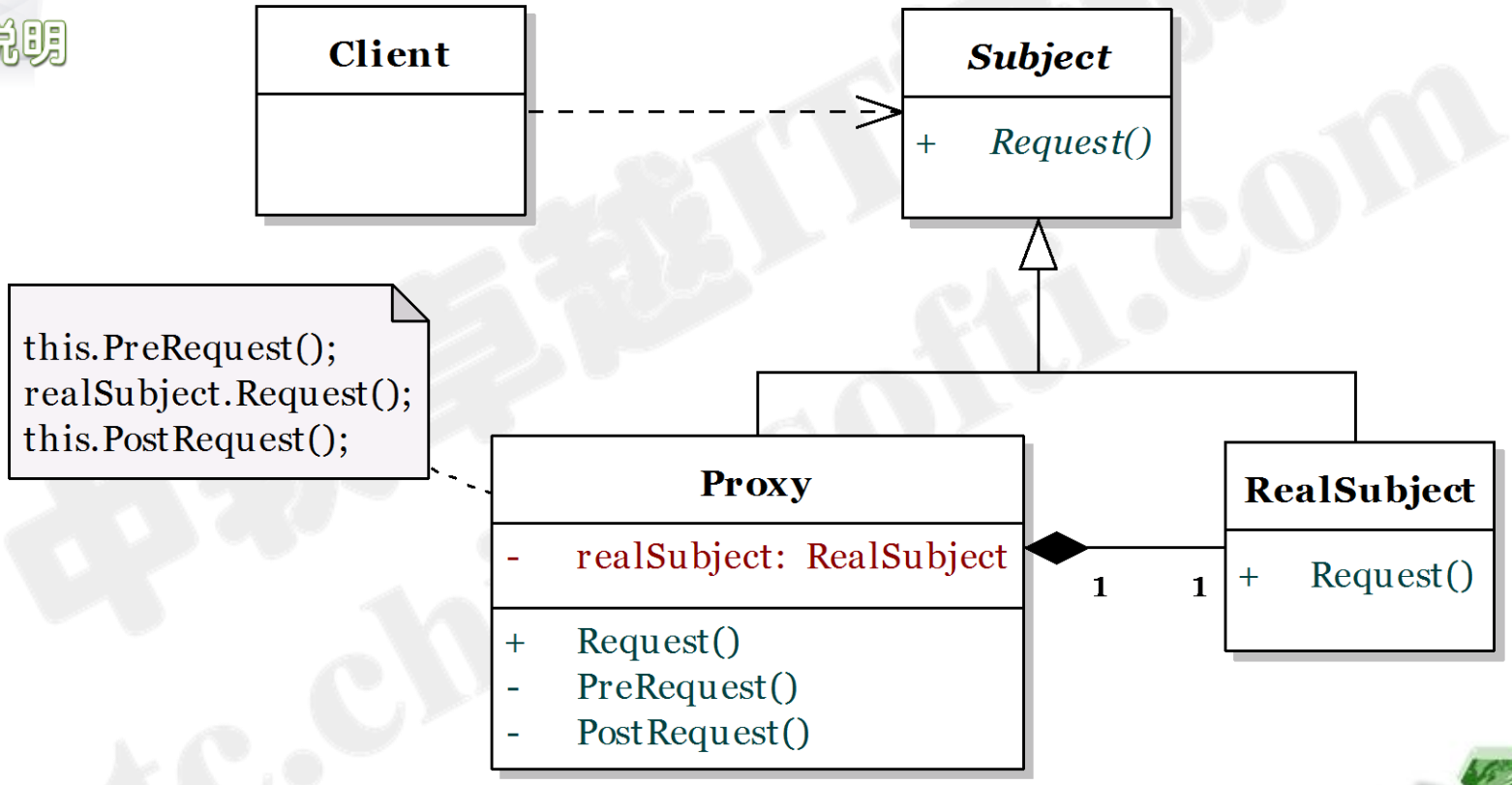
- 了解代理模式的角色
- 理解代理模式的结构
- 理解代理模式的本质

- 理解代理模式的本质

代理模式的角色

- 代理模式给某一个对象提供一个代理，并由代理对象控制对原对象的引用。
- 代理模式的角色：
  - 抽象主题角色（Subject）：真实主题和代理主题的共同接口，从而使得任何使用真实主题的地方都可以使用代理主题。
  - 代理主题（Proxy）角色：代理主题角色内部含有对真实主题的引用，从而可以在任何时候操作真实主题对象；代理角色通常在将客户端调用传递给真实的主题之前或之后，都要执行某个操作，而不是单纯的将调用传递给真实主题对象。
  - 真实主题角色（Real Subject）角色：定义了代理角色所代表的真实对象。

代理模式的结构



代理模式的本质

- 体现了职责分离。例如在领域逻辑层引入代理模式。真实对象为领域对象，代理对象则负责对它持有的真实对象进行持久化；
- 可以实现对职责的装饰。与装饰器模式不同的是，它不能实现动态装饰。
- 如果要解除代理对象与真实对象的耦合，代理对象应组合它们共同的接口Subject，而不是直接组合真实对象。此时，就变成了装饰器模式。

案例：缓存处理

- 在一个电子商务网站中，会对Product、Category等对象进行操作。如果频繁对这些数据进行操作，会因为频繁建立连接，而造成数据库负担过重，或者资源的浪费，从而影响查询性能。
- 引入领域对象的代理对象，为其加入缓存。通常的步骤：
  - 从缓存中获取领域对象；
  - 如果没有，则调用领域对象的方法，获取数据；然后将获得的数据放入到缓存中，并返回。
  - 如果有，则直接返回。

总结

- 代理模式是结构型模式中的一种。
- 代理模式给某一个对象提供一个代理，并由代理对象控制对原对象的引用。
- 代理模式利用了继承与组合的方式实现。



设计模式

桥接(Bridge)模式

代理模式回顾

- 代理模式是结构型模式中的一种。
- 代理模式给某一个对象提供一个代理，并由代理对象控制对原对象的引用。
- 代理模式利用了继承与组合的方式实现。

本章目标

- 了解桥接模式的本质
- 理解桥接模式的结构
- 理解桥接模式在项目中的应用

桥接模式

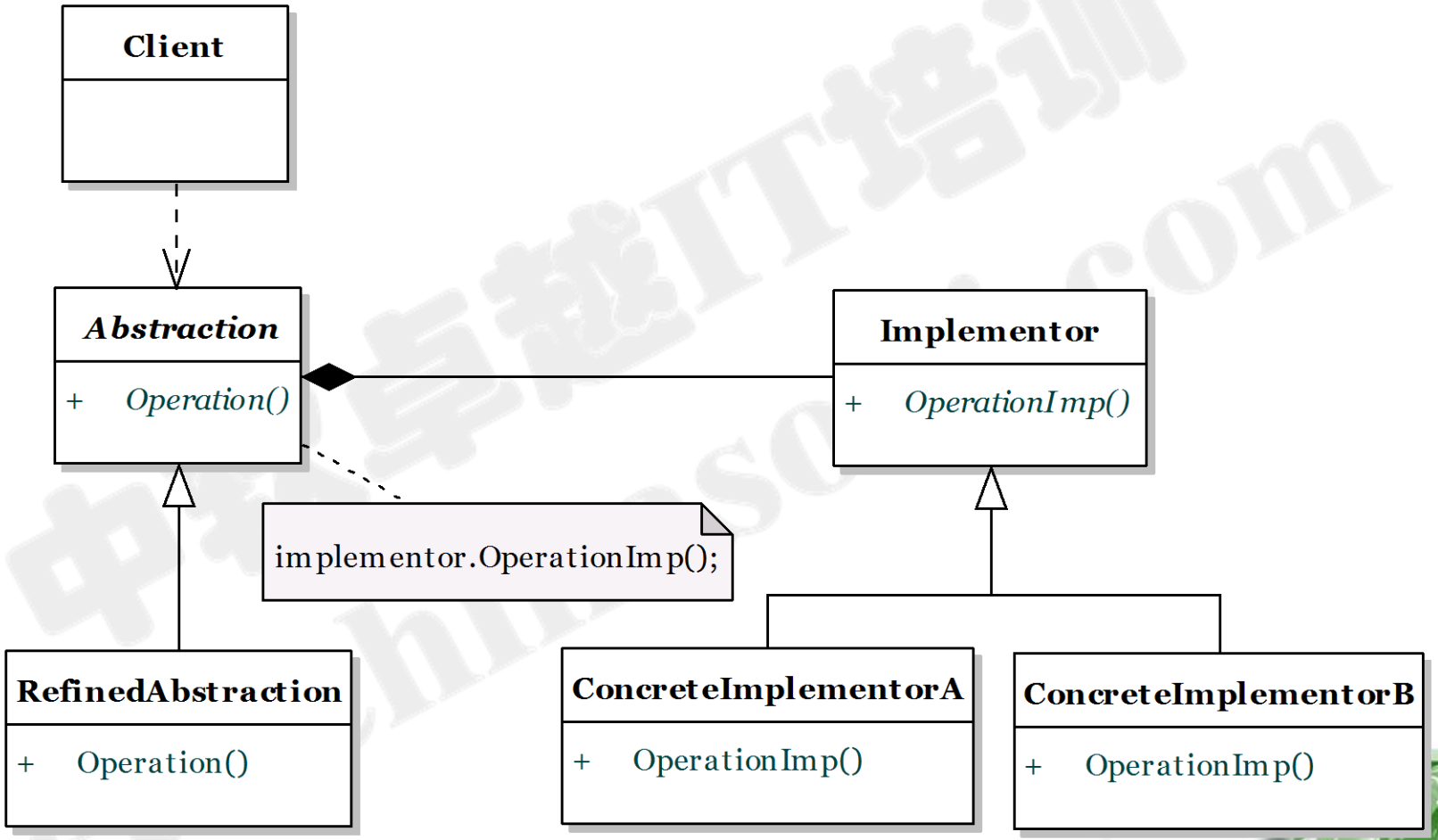
重点难点

- 理解桥接模式的本质

桥接模式的本质

- 桥接模式的用意是“将抽象化(Abstraction)与实现化(Implementation)脱耦，使得二者可以独立地变化”。
- 我们可以理解为，一个对象的实现存在两个方向的变化，我们可以对这两个方向的变化分别进行抽象，然后在以组合的方式关联起来。这样就实现了两者的脱耦。

桥接模式的结构


Client
Abstraction
+ Operation()
Implementor
+ OperationImp()
implementor.OperationImp();
RefinedAbstraction
+ Operation()
ConcreteImplementorA
+ OperationImp()
ConcreteImplementorB
+ OperationImp()

比较毛笔与蜡笔

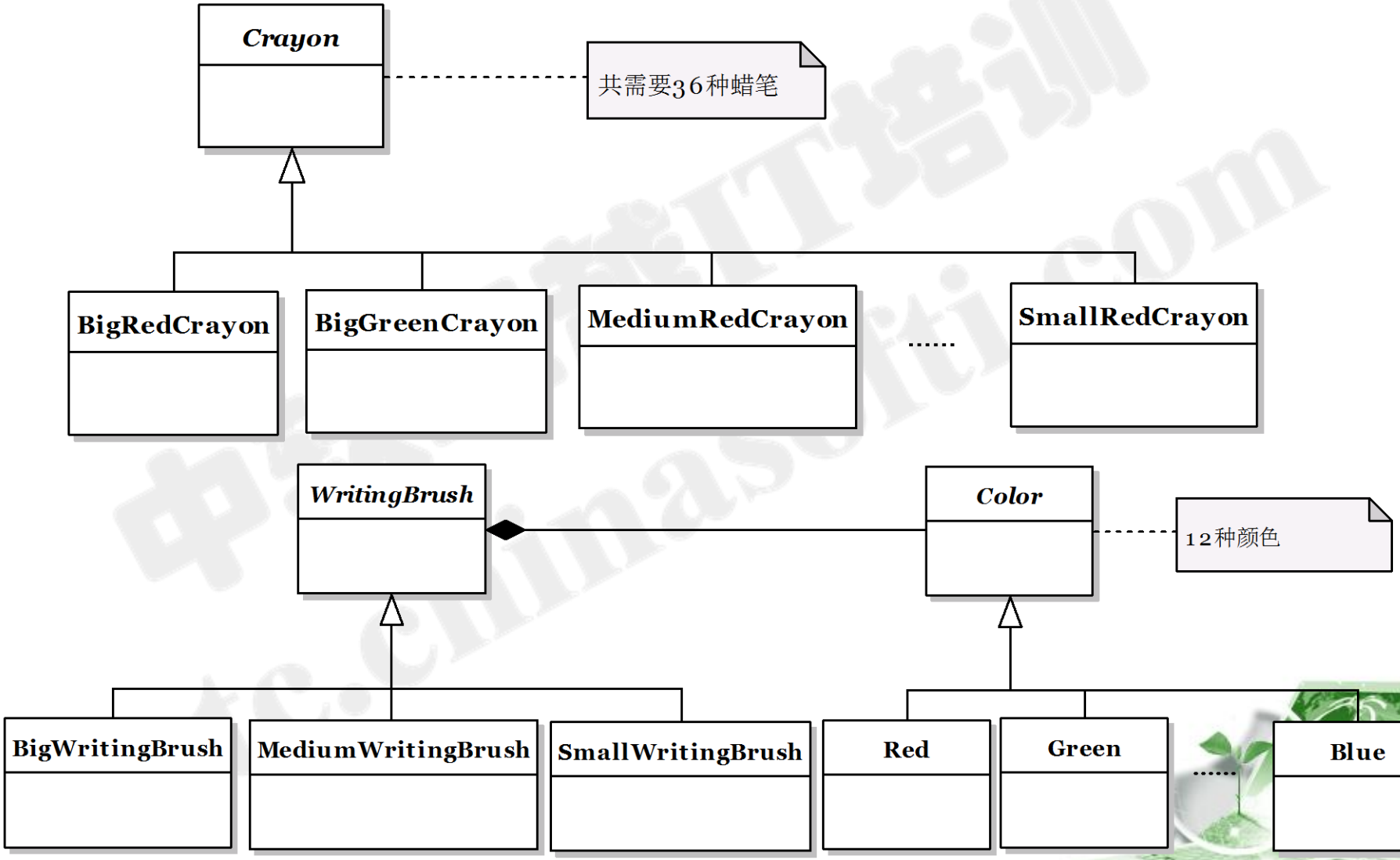
思考

作画时，使用毛笔的数量多，还是蜡笔的数量多？

蜡笔与毛笔的区别

- 用蜡笔作画，假设蜡笔有三种型号：大、中、小。每种蜡笔有七种颜色：赤、橙、黄、绿、青、蓝、紫。因此需要21种蜡笔。
- 用毛笔作画，假定毛笔有三种型号：大、中、小。同样需要七种颜色，但这些颜色是由调色板提供。因此需要三种毛笔，七种调色板。
- 如果，我们增加颜色为12种。则需要蜡笔36种，而毛笔的个数则不变，只需要增加五种调色板即可。想象一下，如果颜色更多，蜡笔的数量会倍数的增加，最后形成了“蜡笔爆炸”！

毛笔优于蜡笔，其原因在于颜色和笔是分开的



案例：票的处理

- 在一个培训系统中，对票的处理可能会在两种情况下执行：
  - 取消票时；
  - 在设定的Deadline到期时；
- 根据事先的设定，票的处理可以分为如下方式：
  - 通知相关人员；
  - 按照优先级分配给后备人员；
  - 将票放弃；
- 如何设计？

利用桥接模式实现

桥接模式的本质

- 桥接模式体现了合成/聚合复用原则：
  - 处理票的情况与处理票的方式之间采用合成关系；
- 桥接模式符合开放-封闭原则：
  - 利用抽象，使得我们可以自由地增加处理票的情况，以及处理票的方式。方式是直接添加实现IActionHandler或IAction的子类；
- 桥接模式体现了职责分离原则：
  - 将两个方向的不同变化分离开，并利用抽象减弱之间的耦合关系。

总结

- 桥接模式是结构型模式中的一种。
- 桥接模式体现了合成/聚合复用原则，将两种变化分解，并进行抽象，以解除二者之间的耦合关系。

设计模式

策略(Strategy)模式

桥接模式回顾

- 桥接模式是结构型模式中的一种。
- 桥接模式体现了合成/聚合复用原则，将两种变化分解，并进行抽象，以解除二者之间的耦合关系。

结构型模式回顾

- 结构型模式关注的是对象之间组合的方式。本质上说，如果对象结构可能存在变化，主要在于其依赖关系的改变。当然对于结构型模式来说，处理变化的方式不仅仅是封装与抽象那么简单，还要合理地利用继承与组合的方法，灵活地表达对象之间的依赖关系。
- 例如，装饰器模式，描述的就是对象间可能存在的多种组合方式，这种组合方式是一种装饰者与被装饰者之间的关系，因此封装这种组合方式，抽象出专门的装饰对象显然是“封装变化”的体现。同样地，桥接模式封装的则是对象实现的依赖关系，而合成模式所要解决的则是对象间存在的递归关系。

本章目标

- 了解行为模式的本质
- 理解策略模式的结构
- 理解策略模式的本质
- 理解策略模式在项目中的应用

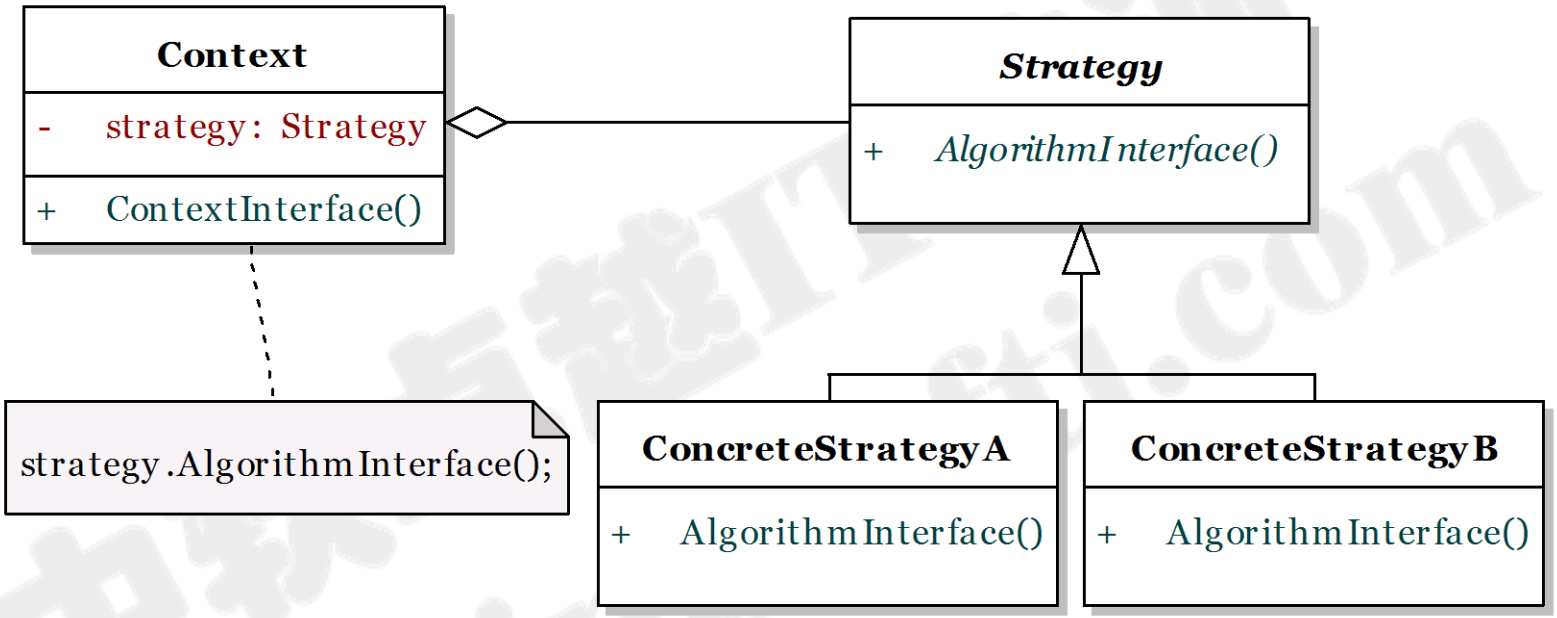
行为模式的本质

- 行为模式关注的是对象的行为。该类型的模式需要做的是对变化的行为进行抽象，通过封装达到整个架构的可扩展性。
- 例如策略模式，就是将可能存在变化的策略或算法抽象为一个独立的接口或抽象类，从而实现未来策略的扩展。命令模式封装一个请求、访问者模式封装“访问”的方式等。
- 行为模式所要封装的行为，恰恰是软件架构中最不稳定的部分，其扩展的可能性也最大。将这些行为封装起来，利用抽象的特性，就提供了扩展的可能。

策略模式的本质

- 策略模式的用意是针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。 策略模式使得算法可以在不影响到客户端的情况下发生变化。
- 使用策略模式可以把行为和环境分割开来。环境类负责维持和查询行为类，各种算法则在具体策略类（Concrete Strategy）中提供。由于算法和环境独立开来，算法的增减、修改都不会影响环境和客户端。
- 策略模式的本质是将算法或策略进行抽象。

策略模式的结构



策略模式就是“准备一组算法，并将每一个算法封装起来，使得它们可以互换。”

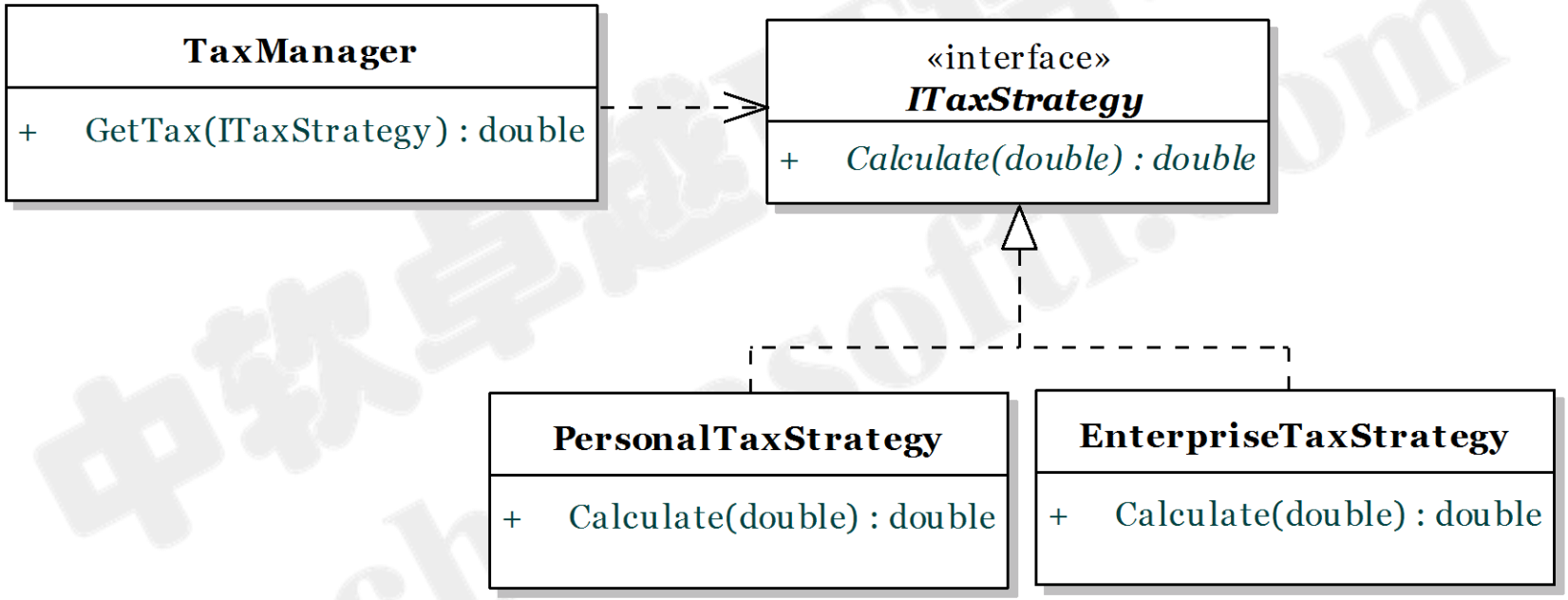
何时使用策略模式？

- 如果一个系统的许多类，它们之间的区别仅在于它们的行为，那么使用Strategy模式可以动态地让一个对象在许多行为中选择一种行为。
- 一个系统需要动态地在几种算法中选择一种。就可以将该算法抽象为统一的接口。由于多态性原则，客户端可以选择使用任何一个具体算法类。
- 使用Strategy模式，可以封装算法使用的数据，避免让客户端知道。
- 如果一个对象包含众多行为，如果不用恰当的模式，就只能使用多重条件选择语句来实现。使用Strategy模式，可以将这些行为转移到相应的具体策略类里面，从而避免使用难以维护的多重条件选择语句。

案例：税务系统

- 假设要开发一个税务系统，有关税务的计算依照纳税人的不同分为个人所得税和企业所得税，这两种税收类型依法应缴纳的税金在计算方式上是迥然不同的两种策略。

案例：税务系统



如何灵活实现策略类的调用

- 客户端在调用策略类时，可能会根据不同的情况选择调用不同的具体策略类。如何灵活地实现客户端对策略类的调用？

解决方案：使用字典


```
//添加策略类
Context.RegisterStrategy("key", new ConcreteStrategy());

//初始化
Context.Initialize();

//调用
Context.InvokeStrategy("key", params object[]);
```

作业

- 将案例中税务管理系统的实现改用委托的方式实现。

总结

- 行为模式的本质是对变化的行为进行封装。
- 策略模式是行为模式中的一种。
- 策略模式封装了算法或策略的实现，使得调用者不用了解具体的算法或策略实现，且能够根据不同的情况，替换不同的算法或策略实现。

设计模式

命令(Command)模式

策略模式回顾

- 策略模式是行为模式中的一种。
- 策略模式封装了算法或策略的实现，使得调用者不用了解具体的算法或策略实现，且能够根据不同的情况，替换不同的算法或策略实现。

本章目标

- 理解命令模式的结构
- 理解命令模式的本质
- 理解命令模式在项目中的应用

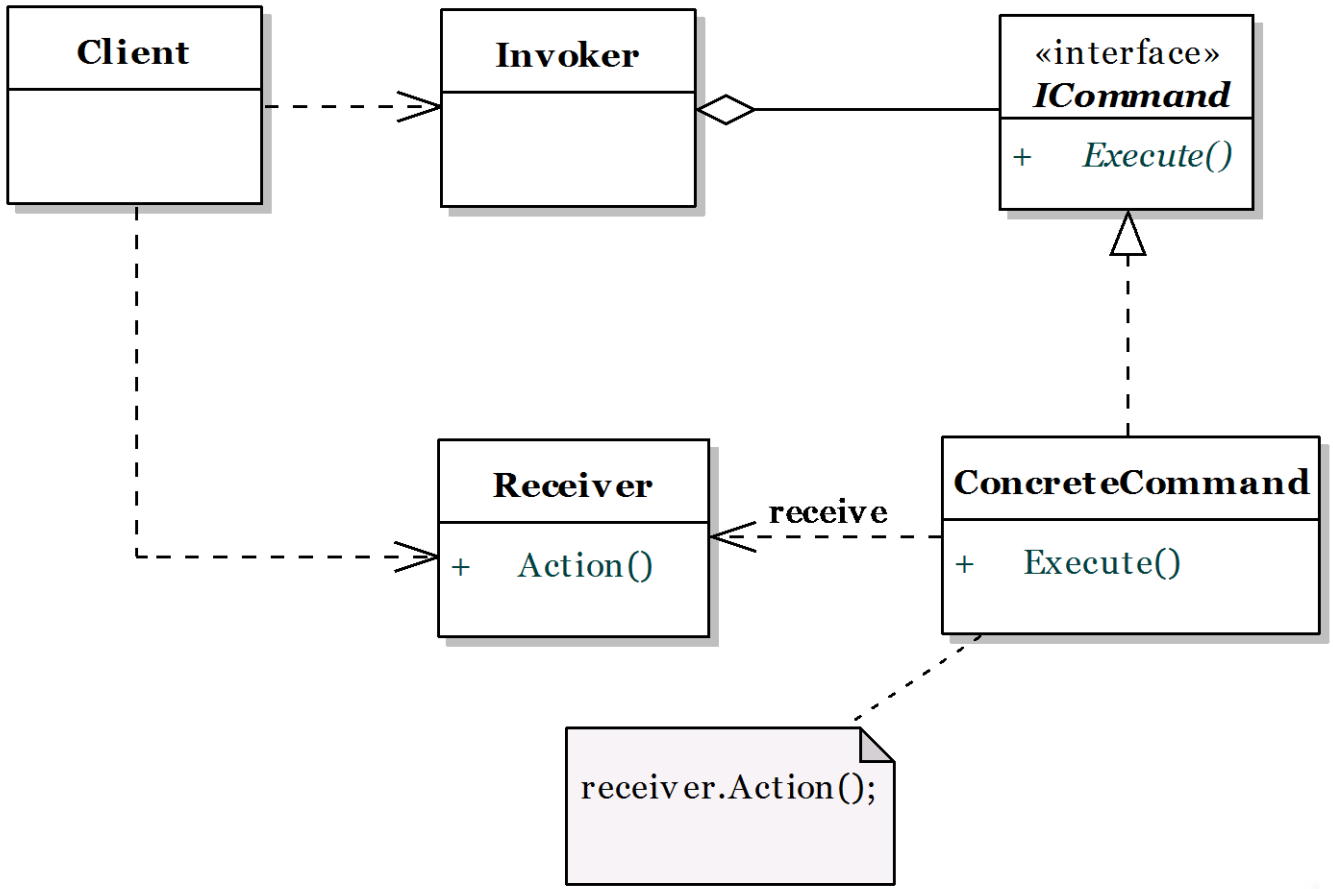
- 区别命令模式与策略模式

命令模式的本质

- 命令模式是对命令的封装。它把发出命令的责任和执行命令的责任分割开，委派给不同的对象。
- 每一个命令（或者说是请求）都是一个操作：请求的一方发出请求，要求执行一个操作；接收的一方收到请求，并执行操作。
- 命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道请求是怎么被接收，以及操作是否被执行、何时被执行，以及是怎么被执行的。换言之，请求的具体实现被彻底封装起来了。

命令模式的结构


说明


Client
Invoker
«interface»
ICommand
+ Execute()
Receiver
+ Action()
receive
ConcreteCommand
+ Execute()
receiver.Action();

区别命令模式与策略模式

- 命令模式与策略模式有何异同之处？

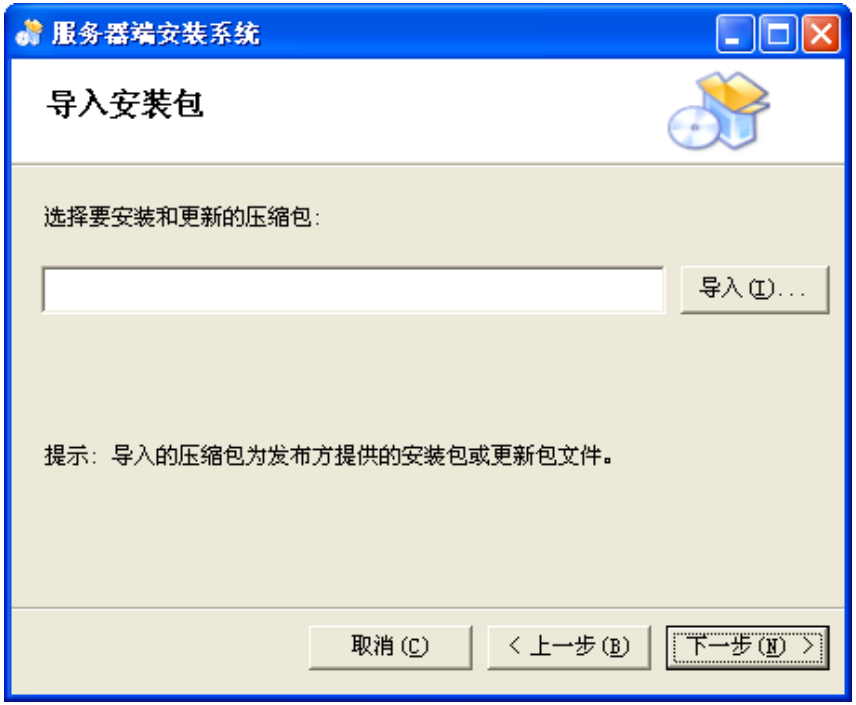
区别命令模式与策略模式

思考

- 从类的结构来讲，两者并无明显区别，都是抽象出统一的接口，以降低与调用者之间的耦合度。不过，在标准的命令模式中，引入了一个Receiver角色，命令对象可以将具体的命令请求委托给Receiver。
- 命令模式封装的是命令或请求，策略模式封装的是算法或策略，即两者的关注点不同。
- 从设计思想来看，两者都是将某种行为（命令或算法）封装起来，以应对未来的变化。
- 命令模式的具体命令对象可以进行组合（命令模式和合成模式的结合），策略对象则每次只能选择一个。
- 在实际开发中，我们无须拘泥于某种模式的名字，而是关注其内涵的思想。

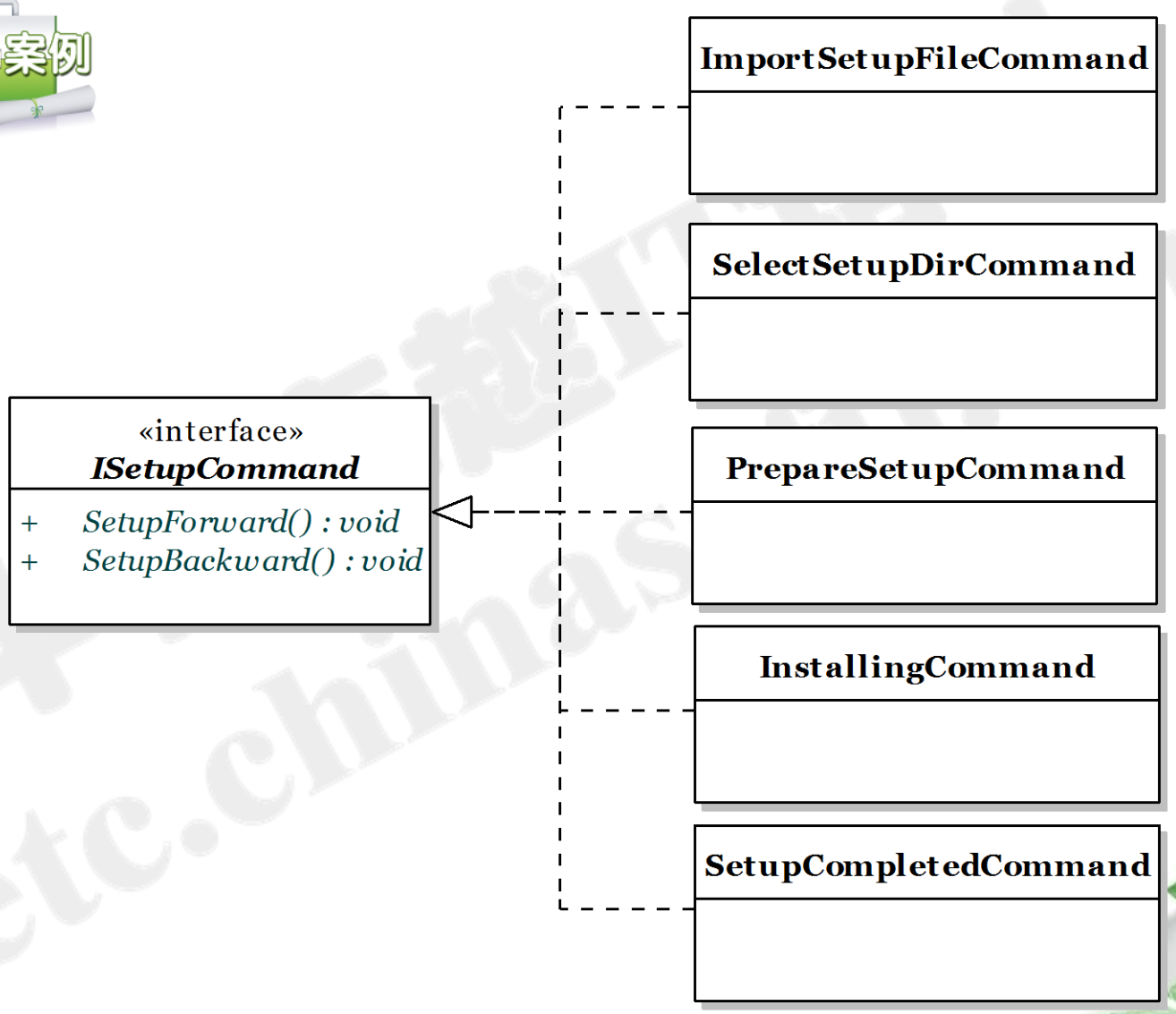
安装向导

案例



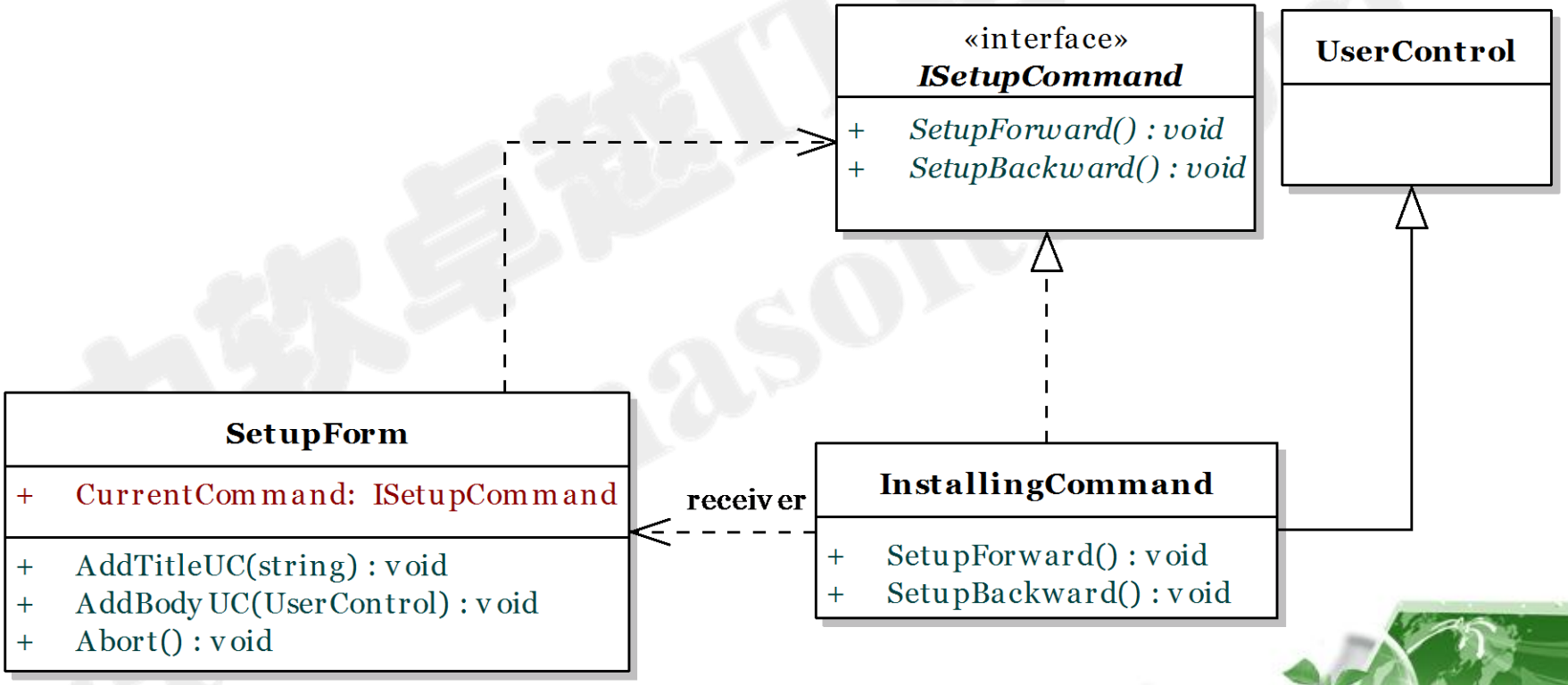
- 开发一个安装向导工具，整个安装共分为五个步骤：
  - 导入要安装的压缩包；
  - 选定安装目录；
  - 准备安装；
  - 执行安装；
  - 安装成功，退出。

引入命令模式



完善命令模式

- 由于安装步骤中某些执行逻辑与安装主窗体相关，因此，可以将安装主窗体作为命令模式的Receiver角色，将相关的执行逻辑放到SetupForm对象中。



总结

- 命令模式是行为模式中的一种。
- 命令模式是对命令的封装。它把发出命令的责任和执行命令的责任分割开，委派给不同的对象。

设计模式

模板方法(Template Method)模式

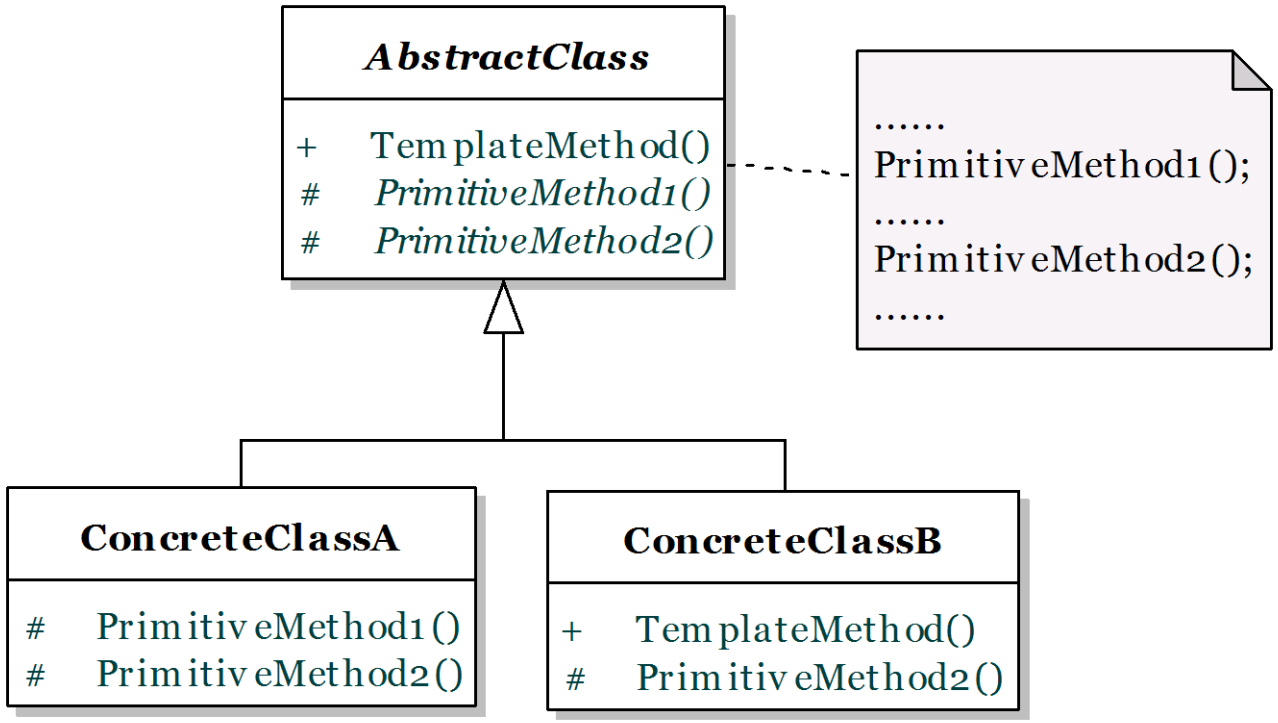
本章目标

- 理解模板方法模式的结构
- 理解模板方法模式的本质
- 理解模板方法模式在项目中的应用

模板方法模式的本质

- 准备一个抽象类，将部分逻辑以具体方法以及具体构造函数的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。这就是模板方法模式的用意。
- 模板方法模式需要开发抽象类和具体子类的设计师之间的协作。一个设计师负责给出一个算法的轮廓和骨架，另一些设计师则负责给出这个算法的各个逻辑步骤。代表这些具体逻辑步骤的方法称做基本方法（primitive method）；而将这些基本法方法总汇起来的方法叫做模版方法（template method），这个设计模式的名字就是从此而来。

模板方法模式的结构



区别模板方法模式与策略模式

- 模板方法模式与策略模式有何异同之处？

模板方法模式与策略模式的区别

- 两者有其共同之处，那就是将某种策略或行为进行了抽象；
- 不同之处在于，模板方法模式封装的统一行为是具体的，而将实现该行为的各个细节进行了抽象；而策略模式则是对整个行为进行了抽象；
- 从结构上讲，模板方法的抽象通常用抽象类来体现；而策略模式则用接口来体现。

培训系统的邮件发送

- 在开发一个培训系统时，需要制订多种不同的Action，发送邮件通知不同的员工。
  - NotifyManagerAction：发送普通邮件给被提名者的Manager；
  - NotifyNomineeAction：发送Meeting Request给被提名者；
- 在不同的场景，会调用不同的Action。

引入模板方法模式

- 考虑Action的操作，不管是哪一种Action，都具有两个相同类型的操作：
  - 获得收邮件的员工信息；
  - 发送邮件；
- 对于调用者而言，并不关心这两个操作，只需要执行Action既可。此时，可引入模板方法模式。Action的执行方法就是一个模版方法，而上面分析的两种操作，就是我们需要抽象的抽象方法。

引入模板方法模式

```
public abstract class ActionBase
{
    public void Execute()  //模版方法
    {
        IList<Employee> employees = GetReceivers();
        foreach (var employee in employees)
        {
            SendMail(employee)
        }
    }
    protected abstract IList<Employee> GetReceivers();
    protected abstract void SendEmail();
}
```

引入模板方法模式

```
public class NotifyManagerAction:ActionBase
{
    protected override IList<Employee> GetReceivers()
    { //获得被提名者对应的Manager信息; }
    protected override void SendMail(Employee emp)
    { //发送普通邮件; }
}
public class NotifyNomineeAction:ActionBase
{
    protected override IList<Employee> GetReceivers()
    { //获得被提名者信息; }
    protected override void SendMail(Employee emp)
    { //发送Meeting Request; }
}
```

总结

- 模板方法模式是行为模式中的一种。
- 模板方法模式利用抽象类，将部分逻辑以具体方法以及具体构造函数的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。
- 模板方法可以看做是策略模式的一种特殊实现，即子类实现的算法或策略实现有一部分共同的实现逻辑。

设计模式

观察者(Observer)模式

本章目标

本章目标

- 理解观察者模式的结构
- 理解观察者模式的本质
- 理解观察者模式在项目中的应用

观察者模式

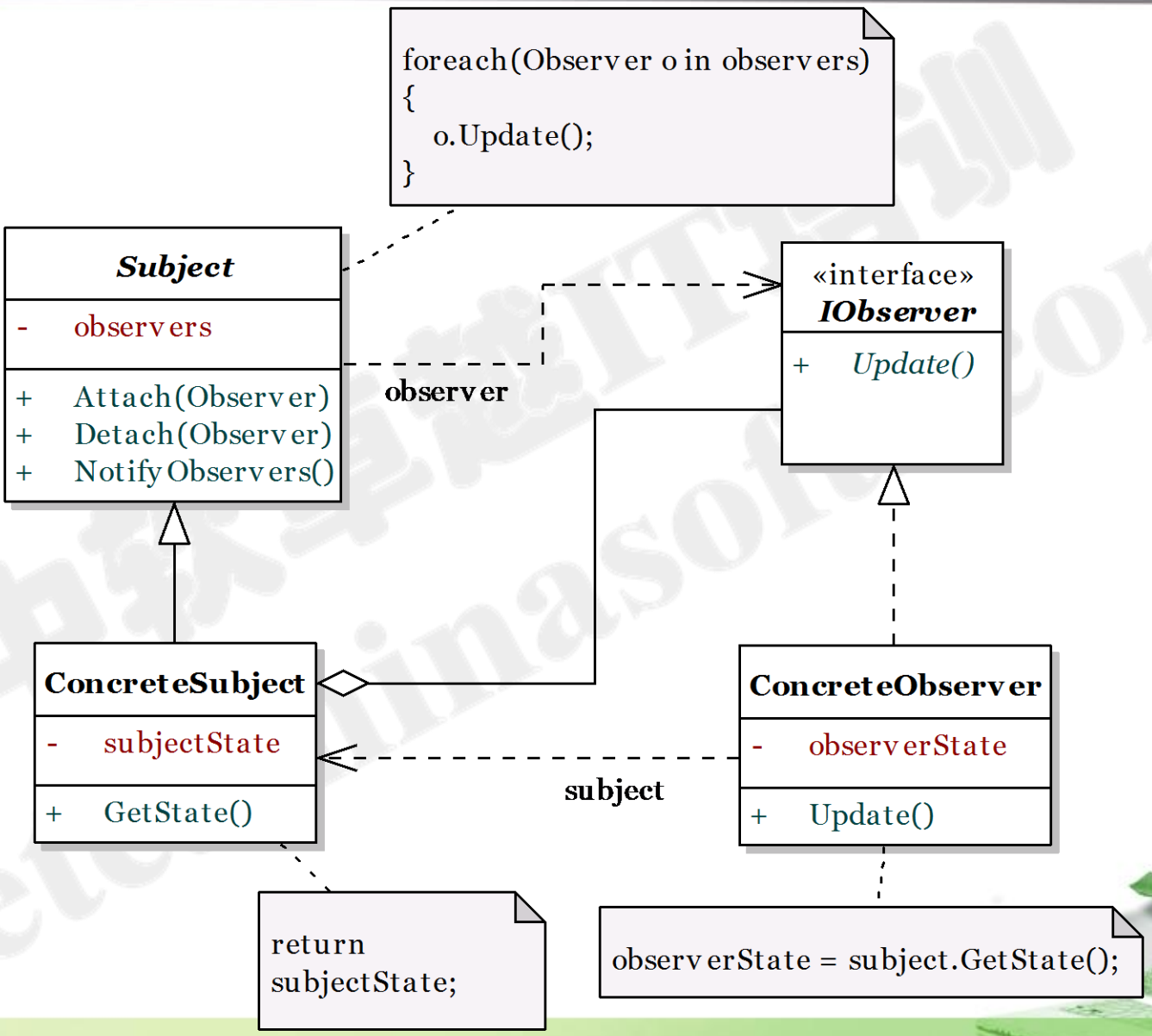
- 理解观察者模式的本质
- 观察者模式在项目中的应用

观察者模式的本质

说明

- 观察者模式的关注重心不是对象的行为，而是两个或多个相互协作类之间的依赖关系。之所以被称为是行为模式，原因是它通过了某种行为来控制这种依赖关系，并产生消息通知进而达到修改被依赖的类的行为或状态的目的。
- 观察者模式的意图：“定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。”
- 观察者模式中最关键的一点是需要对观察者（Observer）角色进行抽象，以解除观察者与被观察者之间的依赖关系。此外，被观察者即主体（Subject）角色，还需要维护一个集合对象，它是一个观察者对象的列表集合，在消息通知的时候，被观察者将遍历该集合内的观察者对象，并调用它们的相关方法。

观察者模式的结构


foreach(Observer o in observers)
{
o.Update();
}
Subject
- observers
+ Attach(Observer)
+ Detach(Observer)
+ NotifyObservers()
observer
«interface»
IObserver
+ Update()
ConcreteSubject
- subjectState
+ GetState()
subject
ConcreteObserver
- observerState
+ Update()
return
subjectState;
observerState = subject.GetState();

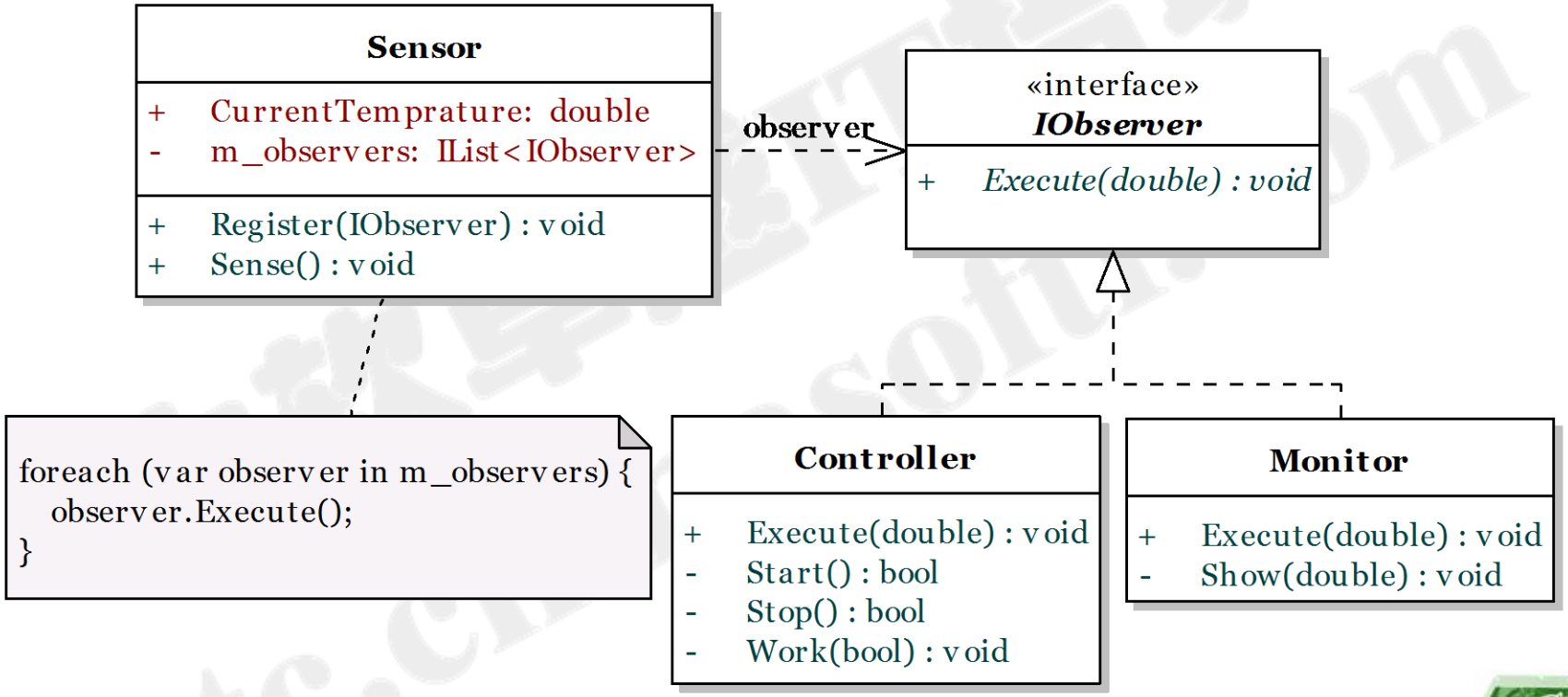
烤箱的设计

- 在烤箱（Oven）中有一个传感器Sensor和一个控制器Controller。Sensor根据加热器以获得烤箱的温度，而控制器则根据温度的值控制烤箱的操作，是应该继续加热，还是停止加热。
- 如果还有一个监视器(Monitor)，当温度发生变化时，就在监视器上显示最新的温度。

- 要求：传感器不公开其获得的烤箱温度值。

烤箱的设计

一案例


Sensor
+ CurrentTemprature: double
- m_observers: IList<IObserver>
+ Register(IObserver) : void
+ Sense() : void
observer
«interface»
IObserver
+ Execute(double) : void
foreach (var observer in m_observers) {
observer.Execute();
}
Controller
+ Execute(double) : void
- Start() : bool
- Stop() : bool
- Work(bool) : void
Monitor
+ Execute(double) : void
- Show(double) : void

利用委托和事件

说明

- 在.NET中，还可以利用委托与事件来实现观察者模式。
- 此时，委托相当于是观察者抽象的接口。定义和发布事件的对象相当于模式的主题（Subject）角色。订阅事件的对象相当于模式的观察者（Observer）角色。
- 正因为此，观察者模式又时有被称为发布者（Publisher）/订阅者（Subsriber）模式。

作业

- 将案例中烤箱的实现改用委托的方式实现。

火箭发射

- 考虑火箭的发射，目前我国的火箭是三级火力推射，当第一级火力燃料耗尽后，会脱落，然后启动第二级火力，依次类推。
- 负责火力推射的是火箭的引擎。每一级引擎都会在燃油耗尽的瞬间通知下一级引擎。此时，上级引擎就扮演了被观察者的角色，而下一级引擎则扮演了观察者的角色。
- 试完成程序模拟火箭运行的状态。

总结

- 观察者模式是行为模式中的一种。
- 观察者模式的关注重心不是对象的行为，而是两个或多个相互协作类之间的依赖关系。
- 观察者模式中最关键的一点是需要对观察者（Observer）角色进行抽象，以解除观察者与被观察者之间的依赖关系。

设计模式总结

- 设计模式是面向对象软件设计经验的总结；
- 设计模式的本质仍然是面向对象设计，其精髓就是封装变化。
- 设计模式的基本原则：
  - 针对接口编程，而不是针对实现编程。
  - 优先使用对象组合，而不是类继承。
- 设计模式的分类：
  - 创建型模式
  - 结构型模式
  - 行为型模式

创建型模式总结

- 创建型模式要处理的是对象的创建。在具体的实例中，我们会遇到这样的情况，就是我们要创建的对象可能会根据需求的不同，而发生变化。此时，我们认为，创建是变化的。创建型模式的目的就是封装创建的变化。
- 常用的创建型模式：
  - 工厂方法模式
  - 抽象工厂模式
  - 建造者模式
  - 单例模式

结构型模式总结

- 结构型模式关注的是对象之间组合的方式。本质上说，如果对象结构可能存在变化，主要在于其依赖关系的改变。当然对于结构型模式来说，处理变化的方式不仅仅是封装与抽象那么简单，还要合理地利用继承与聚合的方法，灵活地表达对象之间的依赖关系。
- 常用的结构型模式：
  - 适配器模式
  - 装饰器模式
  - 合成模式
  - 代理模式
  - 桥接模式

行为模式总结

- 行为模式关注的是对象的行为。该类型的模式需要做的是对变化的行为进行抽象，通过封装达到整个架构的可扩展性。
- 行为模式所要封装的行为，恰恰是软件架构中最不稳定的部分，其扩展的可能性也最大。将这些行为封装起来，利用抽象的特性，就提供了扩展的可能。
- 常用的行为模式：
  - 策略模式
  - 命令模式
  - 模板方法模式
  - 观察者模式

设计模式的本质

- 封装变化
  - 要点：发现功能需求的变化点，然后进行抽象。
- 面向对象思想
  - 要点：明确抽象、继承、封装、多态、组合等面向对象思想的诸多元素，并合理运用这些设计元素。

设计模式的本质

- 对基本思想和原则的主要运用包括：
  - 灵活运用接口或抽象类（如策略模式）；
  - 合理利用继承与组合（如装饰器模式、桥接模式以及适配器模式）；
  - 合理利用抽象和多态，尤其是虚方法和抽象方法的运用（如模板方法模式和抽象工厂模式）；
  - 合理运用集合（观察者模式）；
  - 合理运用职责的封装与信息的隐藏（如工厂方法模式和单例模式）；
  - 注意职责的分离（如桥接模式）。

卓软人生

ETC与你共成长