哈尔滨工程大学
Harbin Engineering University

软件工程学科导论
Introduction to Software Engineering

第6课 软件设计模式概要

王念滨
软件学院大数据与智能计算课题组
2020年6月

二、课程安排

学习内容

软件工程学科概述

软件开发与软件开发管理

软件项目管理概要

软件需求工程概要

软件系统架构概要

软件设计原则与设计模式概要

软件测试与维护要点

软件工程教育与职业发展

第6课 软件设计原则与设计模式概要

软件工程学科导论-软件设计模式

01 软件设计原则
02 软件设计模式
03 设计模式之适配器模式
04 设计模式之装饰器模式
05 设计模式之外观模式
06 设计模式之观察者模式

软件工程学科导论-软件设计模式

01 软件设计原则
02 软件设计模式
03 设计模式之适配器模式
04 设计模式之装饰器模式
05 设计模式之外观模式
06 设计模式之观察者模式

在软件开发中，为提高软件系统的可复用性、可维护性，增加软件的可扩展性和灵活性。程序员应尽力根据软件设计的7条原则来开发程序，从而提高软件的开发效率，降低软件开发成本和维护成本。

01 软件设计原则

1 开闭原则 The Open Closed Principle（OCP）

2 里氏替换原则 Liskov Substitution Principle（LSP）

3 单一职责原则 Single Responsibility Principle（SRP）

4 依赖倒置原则 Dependence Inversion Principle（DIP）

5 接口隔离原则 Interface Segregation Principle（ISP）

6 迪米特法则 Law of Demeter（LoD）

7 合成复用原则 Composite/Aggregate Reuse Principle （CARP）

1 开闭原则　The Open Closed Principle（OCP）

什么是软件开发过程中最不稳定的因素？

——答案是**需求**！

需求在软件开发过程中时时刻刻都可能发生变化。那么，如何灵活应对变化是软件结构设计中最重要也是最困难的一个问题。好的设计带来了极大了灵活性，不好的设计则充斥着僵化的臭味。所以我们要遵循——**开放封闭原则OCP**

Bertrand Meyer，面向对象技术大师，发明了Eiffel 语言和按契约设计（Design by Contract）的思想，名著《面向对象软件构造》的作者，法国工程院院士。

1 开闭原则　The Open Closed Principle（OCP）

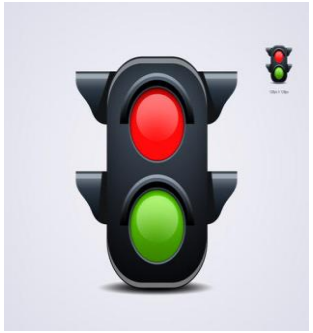
开放封闭原则（OCP，Open Closed Principle）是所有面向对象原则的核心。软件设计本身所追求的目标就是**封装变化、降低耦合。**

开放封闭原则正是对这一目标的最直接体现。其他的设计原则，很多时候是为实现这一目标服务的，例如后面将介绍的Liskov替换原则实现最佳的、正确的继承层次，就能保证不会违反开放封闭原则。

OCP核心的思想是：

软件实体应该是可扩展，而不可修改的。
对扩展是开放的，而对修改是封闭的。

1 开闭原则 The Open Closed Principle（OCP）



软件实体（类、模块、函数等）应该是可扩展的，但是不可修改的。

OCP有两大特征：

●对于扩展是开放的（Open for extension）

模块的行为可以扩展，当应用的需求改变时，可以对模块进行扩展，以满足新的需求。

●对于更改是封闭的（Closed for modification）

对模块行为扩展时，不必改动模块的源代码或二进制代码。

1 开闭原则　The Open Closed Principle（OCP）

在许多方面，OCP是面向对象设计的核心所在。遵循这个原则可以带来面向对象技术所声称的巨大好处（也就是：灵活性、可重用性以及可维护性）。然而，并 不是说只要使用一种面向对象语言就是遵循了这个原则。对于应用程序中的每个部分都肆意地进行抽象同样不是一个好主意。正确的做法是，开发人员应该仅仅对程 序中呈现出频繁变化的那些部分做出抽象。拒绝不成熟的抽象和抽象本身一样重要。

2 里氏替换原则
Liskov Substitution Principle（LSP）

定义

“若对于类型S的任一对象o_1，均有类型T的对象o_2存在，使得在T定义的所有程序P中，用o_1替换o_2之后，程序的行为不变，则S是T的子类型”。

所有引用基类的地方必须能透明底使用其子类的对象。

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

如果在任何情况下，子类（或子类型）或实现类与基类都是可以互换的，那么继承的使用就是合适的。为了达到这一目标，**子类不能添加任何父类没有的附加约束** “子类对象必须可以替换父类对象”

2 里氏替换原则　Liskov Substitution Principle（LSP）

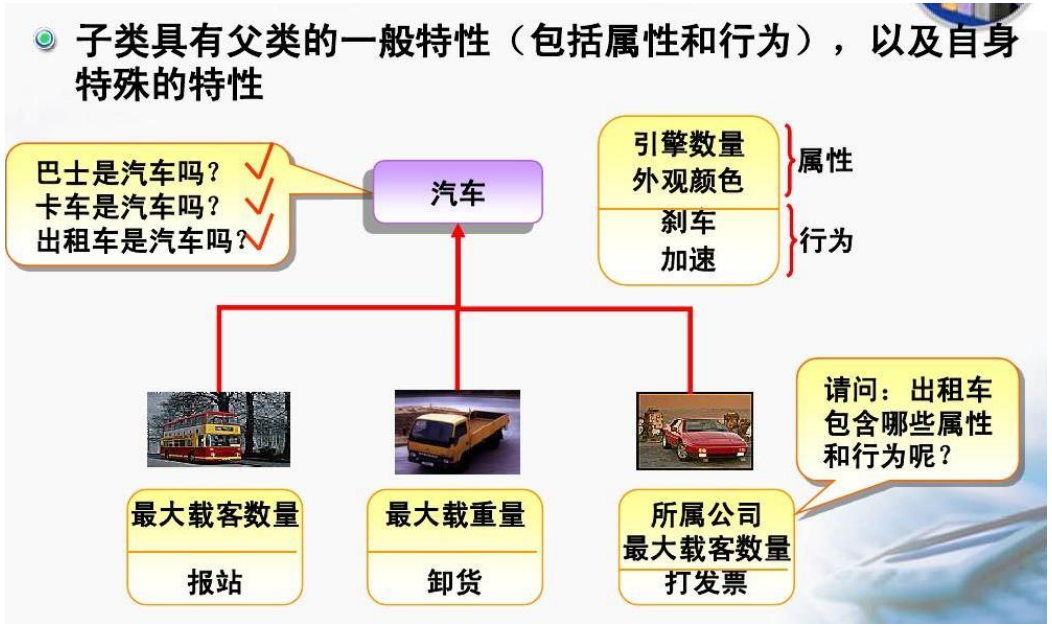
问题由来：有一个功能P1，由类A完成，现在对功能P1扩展，扩展后新功能P由原功能P1与新功能P2组成。新功能P由类A的子类B来完成，则子类B在完成新功能P2时，有可能导致原有功能P1发生故障。

解决方案：使用继承是，遵循LSP。类B继承类A时，除添加新的方法完成新新增功能P2外，尽量不要重写父类A的方法，也尽量不要重载父类A的方法。

LSP含义：父类中凡是已经实现好的方法（相对于抽象方法而言），实际上是在设定一系列的规范和契约。虽然它并未强求所有的子类必须遵守这些契约，但若子类对这些非抽象方法任意修改，则会对整个继承体系造成破坏。

继承的问题：带来侵入性。子类必须继承父类的所有属性和方法。增加了对象间的耦合性，降低了程序的可移植性。

2 里氏替换原则　Liskov Substitution Principle（LSP）



- 长方形与正方形
  - 假如我们有一个类：长方形（Rectangle）
  - 我们需要一个新的类，正方形（Square）
  - 问：可否直接继承长方形？

没问题，因为数学上正方形就是长方形的子类！

3 依赖倒置原则　Dependence Inversion Principle（DIP）

定义

高层模块不应该依赖于低层模块，二者应该依赖于抽象；抽象不应该依赖于细节。细节应该依赖于抽象。

High level modules should not depend upon low level modules.Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions.

解释 抽象：指接口或抽象类（对开发而言）；　　细节：指具体的实现类。

核心思想　　要面向接口编程，而不是要面向实现编程。

作用　降低类之间的耦合性，提高系统的稳定性。减少开发引起的风险。提高代码的可读性，可维护性

3 依赖倒置原则　Dependence Inversion Principle（DIP）

如何满足此原则

每个类尽量要提供接口或抽象类，或者两者都具备；

变量的声明类型尽量用接口或抽象类，而不要去用具体的实现类；

任何类都不应该从具体类派生；

使用继承是尽量遵循里氏替换原则。

3 依赖倒置原则　Dependence Inversion Principle（DIP）

分析

所谓“倒置”是相对于传统的开发方法（例如结构化方法）中总是倾向于让高层模块依赖于低层模块而言的软件结构而言的。高层包含应用程序的策略和业务模型，而低层包含更多的实现细节，平台相关细节等。高层依赖低层将导致：

（1）难以复用。通常改变一个软硬件平台将导致一些具体的实现发生变化，如果高层依赖低层，这种变化将导致逐层的更改；

（2）难以维护，低层通常是易变的

4 单一职责原则　Single Responsibility Principle（SRP）

定义：就一个类而言，应该只有一个导致其变化的原因。

There should never be more than one reason for a class to change.

单一职责包含两层含义：

（1）一个模块只完成一个功能；　（2）一个功能只由一个模块完成

存在的问题

如果有多个原因引起类的变化，则类应当被拆分。对象不应该承担太多的职责。如果一个类承担太多的职责，则存在职责的变化可能削弱或抑制该类实现其他职责的能力。

另外，当某个客户端需要该对象的某一个职能时，不得不将不需要的其他职责也包含进来，从而造成代码浪费。

4 单一职责原则　Single Responsibility Principle（SRP）

使用单一职责原则的好处

（1）降低类的复杂度。一个类只负责一个职责，其逻辑比负责多个职责要简单得多；

（2）对于提高代码的可读性，降低复杂性，可读性自然提高。提高代码的可维护性。

（3）变更引起的风险降低，变更时必然的，若SRP遵守得好，则修改一个功能时可以显著降低对其他功能的影响。

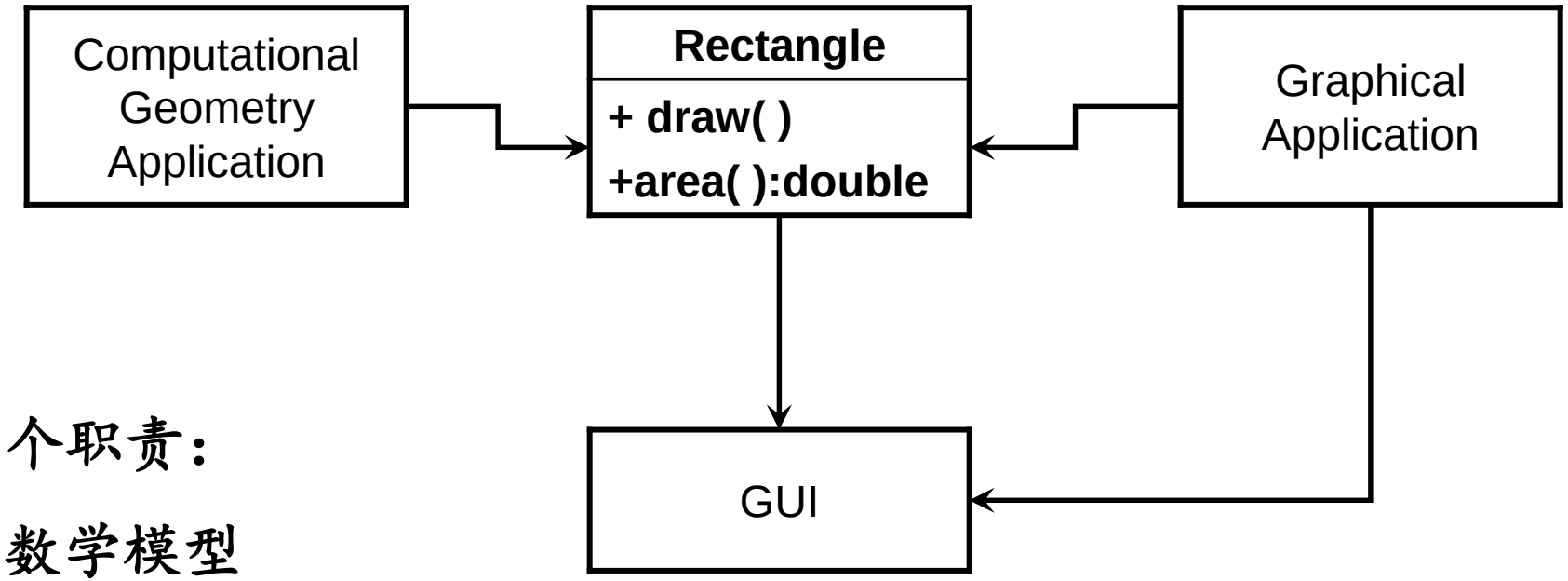
体会

SRP是最简单地，但却是应用中最难处理和应用的一条法则。由于需要设计者发现类的不同职责，并结合实际考虑划分，经验与分析都需要。

调查表明，好的程序员，即使并未学习过SRP，但在实际编程中，也会不自觉地使用这一原则。

经验，凡是存在if....then...的地方，都是可以考虑动手的地方。

4 单一职责原则　Single Responsibility Principle（SRP）



Rectangle类具有两个职责：

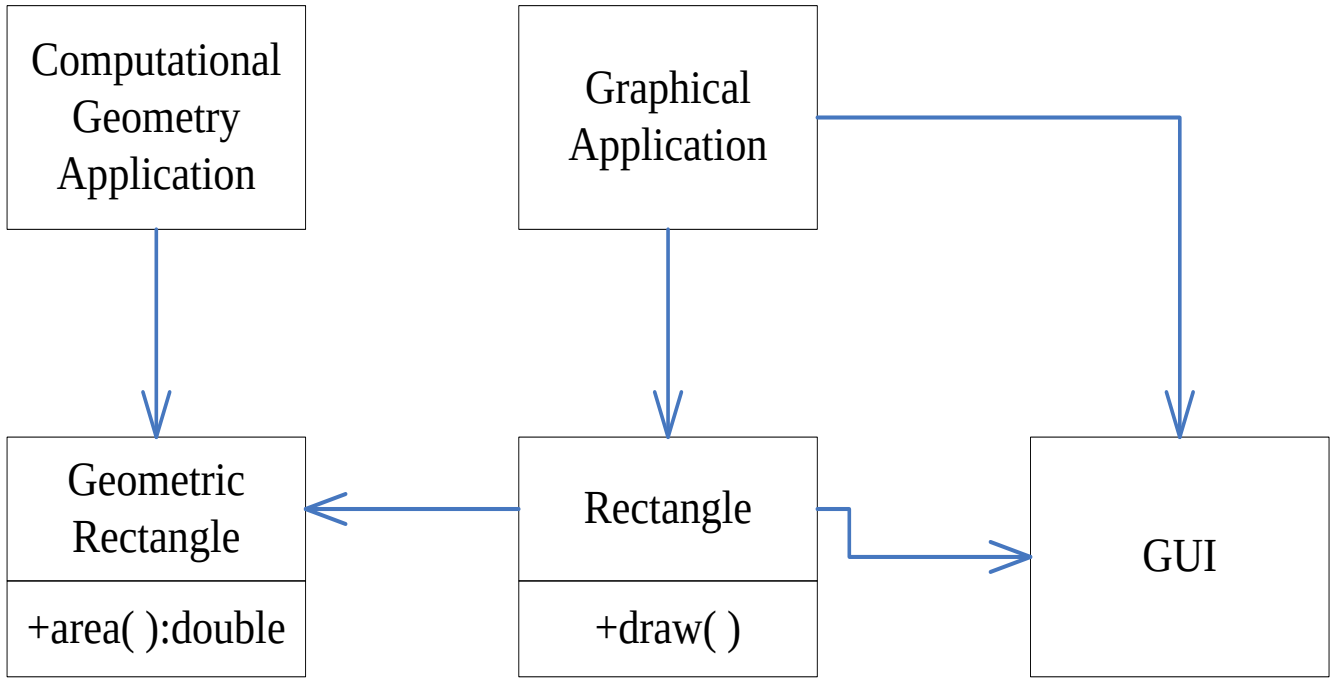
1. 计算矩形面积的数学模型
2. 将矩形在一个图形设备上描述出来

4 单一职责原则　Single Responsibility Principle（SRP）

- Rectangle类违反了SRP，具有两个职能——计算面积和绘制矩形
- 这种对SRP的违反将导致两个方面的问题：
  - 包含不必要的代码
    - 一个应用可能希望使用Retangle类计算矩形的面积，但是却被迫将绘制矩形相关的代码也包含进来
  - 一些逻辑上毫无关联的原因可能导致应用失败
    - 如果GraphicalApplication的需求发生了变化，从而对Rectangle类进行了修改。但是这样的变化居然会要求我们重新构建、测试以及部署ComputationalGeometryApplication，否则其将莫名其妙的失败。

4 单一职责原则 Single Responsibility Principle（SRP）

基于SRP原则的修改



5 接口隔离原则　Interface Segregation Principle（ISP）

定义：

- 不应该强迫客户依赖于他们不用的方法
- 一个类的不内聚的“胖接口”应该被分解成多组方法，每一组方法都服务于一组不同的客户程序。

5 接口隔离原则　Interface Segregation Principle（ISP）

```
class Door{
    public:
        virtual void Lock( )=0;
        virtual void Unlock( )=0;
        virtual bool
    IsDoorOpen( )=0;
};
```

- Door可以加锁、解锁、而且可以感知自己是开还是关；
- Door是抽象基类，客户程序可以依赖于抽象而不是具体的实现

增加功能

如果门打开时间过长，它就会报警。（比如宾馆客房的门）

5 接口隔离原则　Interface Segregation Principle（ISP）

为了实现上述新增功能，我们要求Door与一个已有的Timer对象进行交互

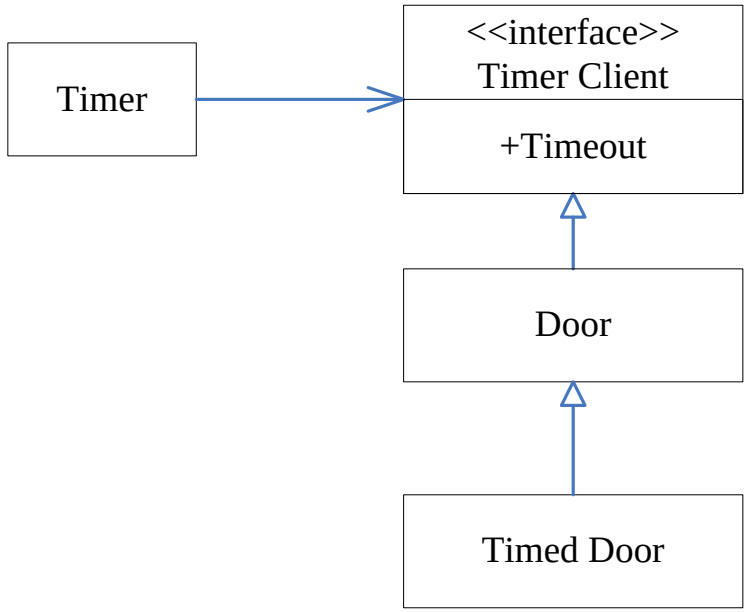
```
class Timer{
public:
    void Register(int timeout,TimerClient* client);
};

class TimerClient{
public:
    virtual void TimerOut( );
};
```

- 如果一个对象希望得到超时通知，它可以调用Timer的Register函数。
- 该函数有两个参数，一个是超时时间，另一个是指向TimerClient对象的指针，此对象的TimerOut函数会在超时时被调用

问题

我们如何将TimerClient和TimedDoor联系起来？

5 接口隔离原则 Interface Segregation Principle（ISP）



- 问题——接口污染

在Door接口中加入新的方法（Timeout），而这个方法仅仅只为它的一个子类带来好处。

——如果每次子类需要一个新方法时它都被加到基类接口中，基类接口将很快变胖。

胖接口将导致SRP，LSP被违反，从而导致脆弱、僵化

5 接口隔离原则　Interface Segregation Principle（ISP）

- 客户的反作用力：
  - 通常接口的变化将导致client的改变
  - 但是很多时候，接口之所以变化是因为客户需要他们变化

→ Client对interface具有反作用力！

TimedDoor的多个超时请求问题，导致Timer接口做出下面的调整：

```
class Timer{
public:
        void Register(int timeout, int timeOutId, TimerClient* client);
};

class TimerClient{
public:
        virtual void TimerOut(int timeOutId );
};
```

TimedDoor对Timer接口的影响会传递到Door接口，从而导致所有Door都受到此影响，而且这一影响还会影响到Door的所有Clients——牵一发动全身

6 迪米特法则　　Law of Demeter（LoD）

- 迪米特原则定义

  迪米特原则(Law of Demeter, LoD)又称为最少知识原则(Least Knowledge Principle, LKP)，它有多种定义方法，其中几种典型定义如下：

  1、不要和“陌生人”说话。

  2、只与你的直接朋友通信。

  3、每一个软件单位对其他的单位都只有最少的知识，而且局限于那些与本单位密切相关的软件单位。

6 迪米特法则　　Law of Demeter（LoD）

迪米特法则特点

- 简单地说，迪米特原则就是指一个软件实体应当尽可能少的与其他实体发生相互作用。这样，当一个模块修改时，就会尽量少的影响其他的模块，扩展会相对容易，这是对软件实体之间通信的限制，它要求限制软件实体之间通信的宽度和深度。
- 在迪米特原则中，对于一个对象，其朋友包括以下几类：

  1、当前对象本身(this)；

  2、以参数形式传入到当前对象方法中的对象；

  3、当前对象的成员对象；

  4、如果当前对象的成员对象是一个集合，那么集合中的元素也都是朋友；

  5、当前对象所创建的对象。
- 任何一个对象，如果满足上面的条件之一，就是当前对象的“朋友”，否则就是“陌生人”。

6 迪米特法则　　Law of Demeter（LoD）

迪米特法则特点

- 狭义的迪米特原则：可以降低类之间的耦合，但是会在系统中增加大量的小方法并散落在系统的各个角落，它可以使一个系统的局部设计简化，因为每一个局部都不会和远距离的对象有直接的关联，但是也会造成系统的不同模块之间的通信效率降低，使得系统的不同模块之间不容易协调。

- 广义的迪米特原则：指对对象之间的信息流量、流向以及信息的影响的控制，主要是对信息隐藏的控制。信息的隐藏可以使各个子系统之间脱耦，从而允许它们独立地被开发、优化、使用和修改，同时可以促进软件的复用，由于每一个模块都不依赖于其他模块而存在，因此每一个模块都可以独立地在其他的地方使用。一个系统的规模越大，信息的隐藏就越重要，而信息隐藏的重要性也就越明显。

6 迪米特法则　　Law of Demeter（LoD）

迪米特法则特点

- 迪米特原则的主要用途在于控制信息的过载：

  1、在类的划分上，应当尽量创建松耦合的类，类之间的耦合度越低，就越有利于复用，一个处在松耦合中的类一旦被修改，不会对关联的类造成太大波及；

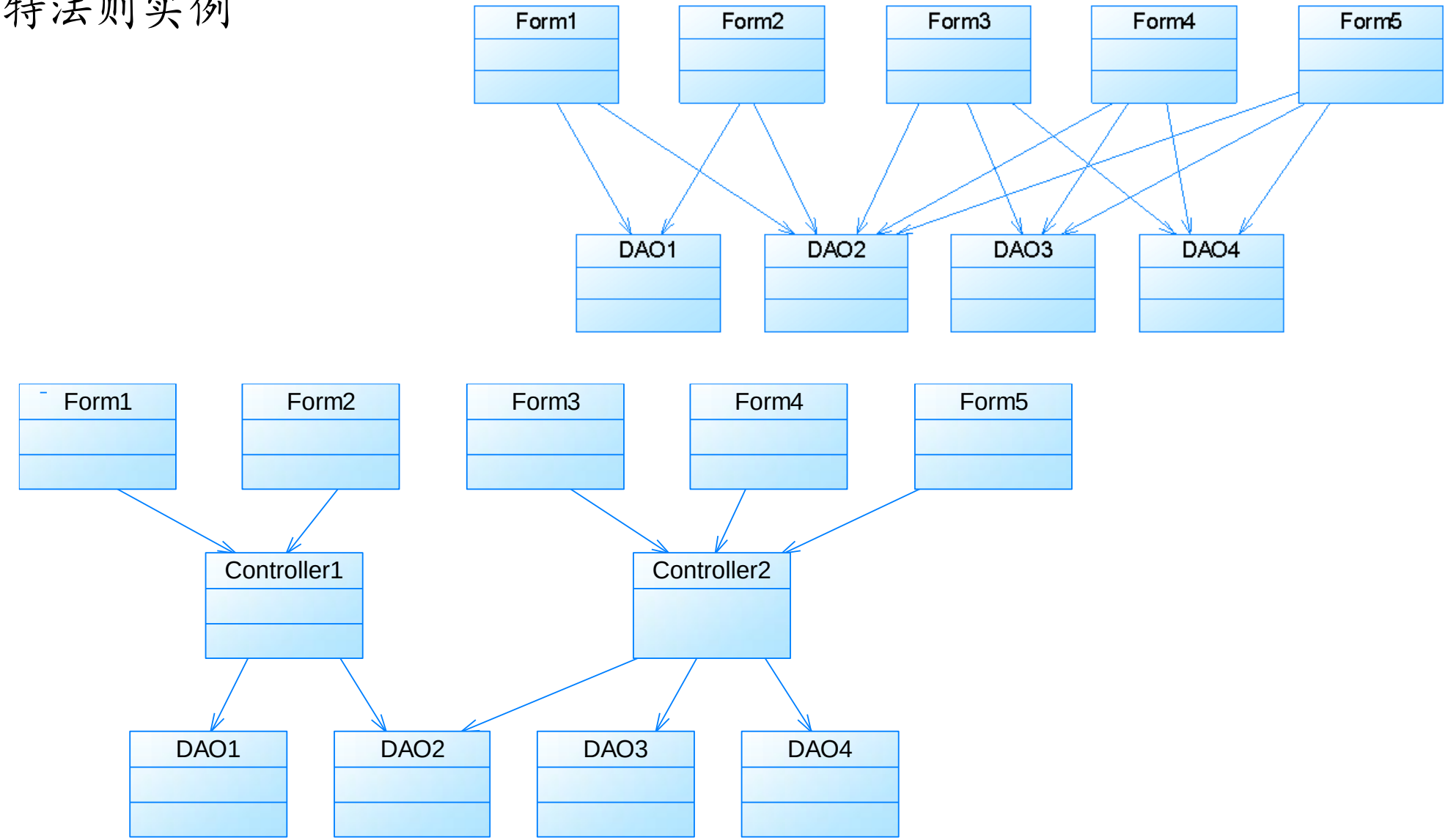
  2、在类的结构设计上，每一个类都应当尽量降低其成员变量和成员函数的访问权限；

  3、在类的设计上，只要有可能，一个类型应当设计成不变类；

  4、在对其他类的引用上，一个对象对其他对象的引用应当降到最低。

6 迪米特法则 Law of Demeter（LoD）

迪米特法则实例



7 合成复用原则 Composite/Aggregate Reuse Principle （CARP）

- 合成复用原则定义

  合成复用原则(Composite Reuse Principle, CRP)又称为组合/聚合复用原则(Composition/ Aggregate Reuse Principle, CARP)，其定义如下：

  尽量使用对象组合，而不是继承来达到复用的目的。

合同复用原则特点

- 合成复用原则就是指在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用其已有功能的目的。简言之：要尽量使用组合/聚合关系，少用继承。

- 在面向对象设计中，可以通过两种基本方法在不同的环境中复用已有的设计和实现，即通过组合/聚合关系或通过继承。

  1、继承复用：实现简单，易于扩展。破坏系统的封装性；从基类继承而来的实现是静态的，不可能在运行时发生改变，没有足够的灵活性；只能在有限的环境中使用。（“白箱”复用 ）

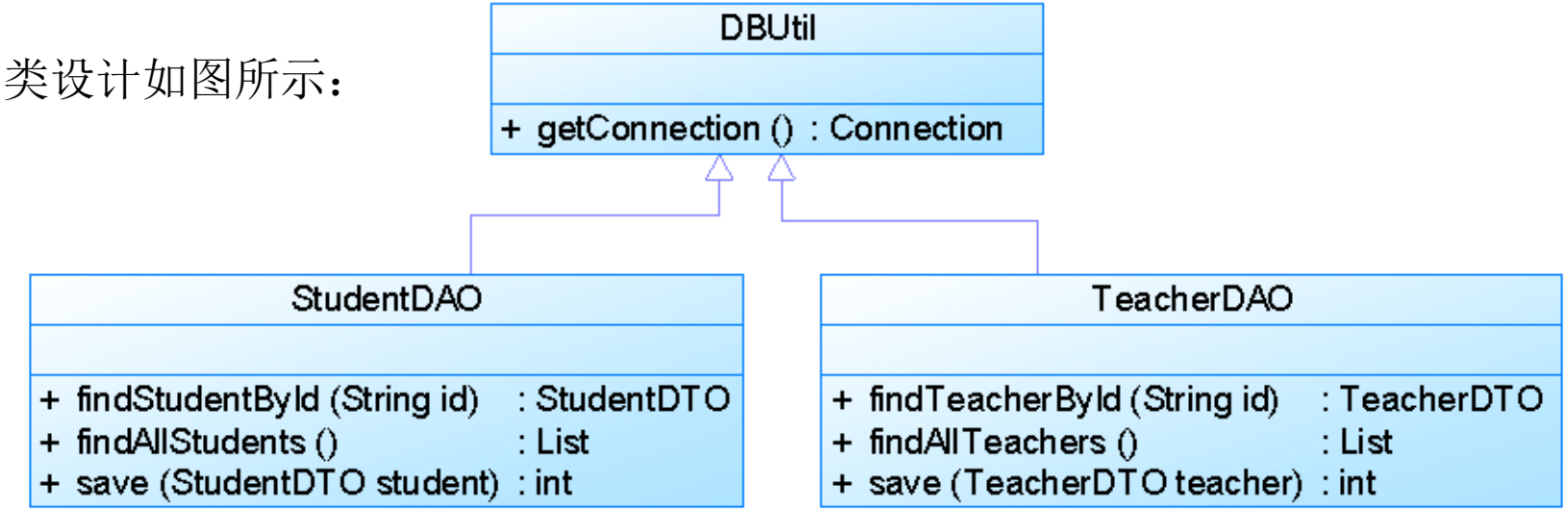
  2、组合/聚合复用：耦合度相对较低，选择性地调用成员对象的操作；可以在运行时动态进行。（“黑箱”复用 ）

7 合成复用原则 Composite/Aggregate Reuse Principle （CARP）

- 组合/聚合可以使系统更加灵活，类与类之间的耦合度降低，一个类的变化对其他类造成的影响相对较少，因此一般首选使用组合/聚合来实现复用；其次才考虑继承，在使用继承时，需要严格遵循里氏代换原则，有效使用继承会有助于对问题的理解，降低复杂度，而滥用继承反而会增加系统构建和维护的难度以及系统的复杂度，因此需要慎重使用继承复用。

7 合成复用原则 Composite/Aggregate Reuse Principle （CARP）

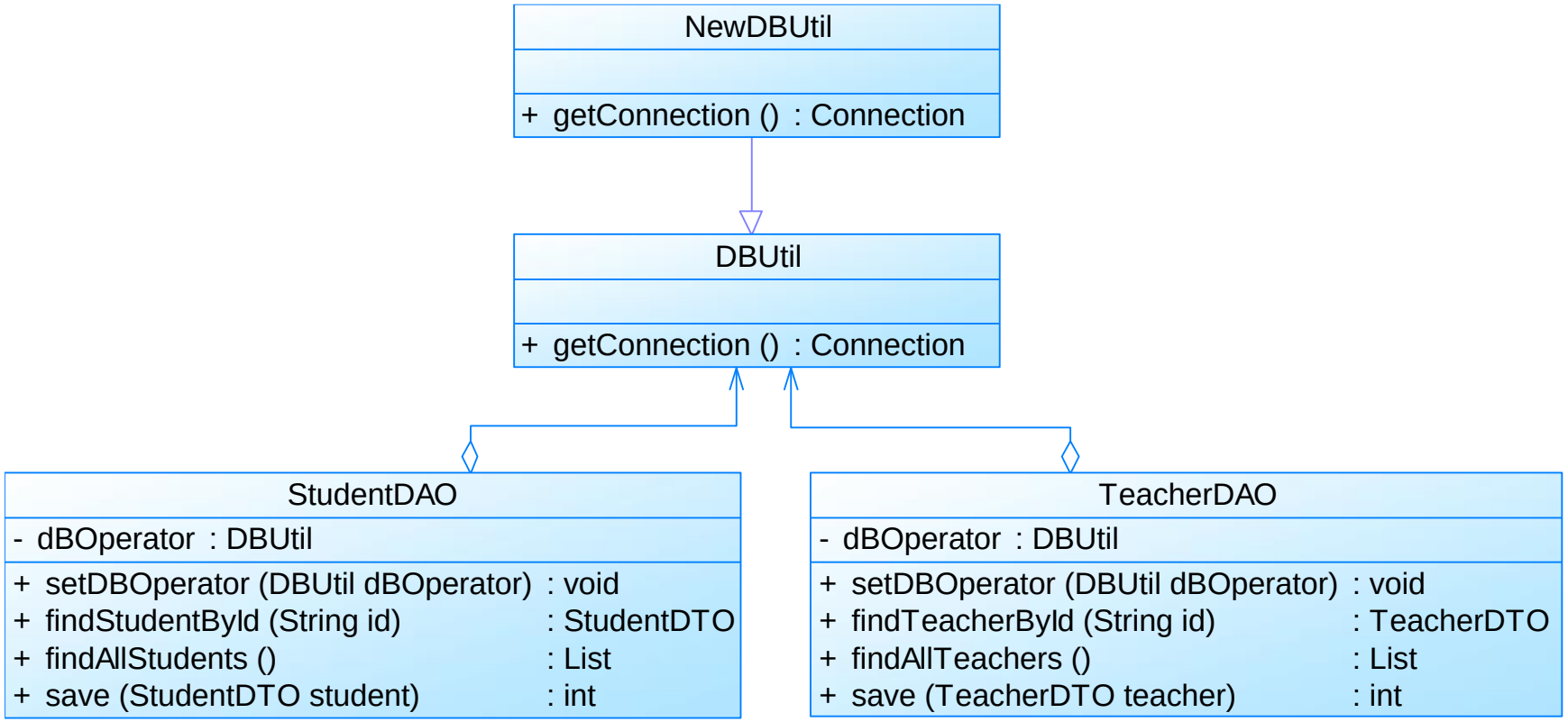
- 某教学管理系统部分数据库访问类设计如图所示：



- 需求说明
- 如果需要更换数据库连接方式，如原来采用JDBC连接数据库，现在采用数据库连接池连接，则需要修改DBUtil类源代码。如果StudentDAO采用JDBC连接，但是TeacherDAO采用连接池连接，则需要增加一个新的DBUtil类，并修改StudentDAO或TeacherDAO的源代码，使之继承新的数据库连接类，这将违背开闭原则，系统扩展性较差。

- 现使用合成复用原则对其进行重构。

7 合成复用原则 Composite/Aggregate Reuse Principle （CARP）



这些原则并不是孤立存在的，它们相互依赖，相互补充。

开闭原则是总纲，它告诉我们要对开展开放，对修改关闭。

里氏替换原则，告诉我们不要破坏继承体系。

依赖倒置原则，告诉我们要面向接口编程。

单一职责原则，告诉我们类的职责要单一。

接口隔离原则，告诉我们要在设计接口时精而简单。

迪米特法则，告诉我们要降低耦合度。

合成复用原则，告诉我们要多用组合或聚合关系，少用继承复用关系。

面向对象设计7大原则

01 软件设计原则

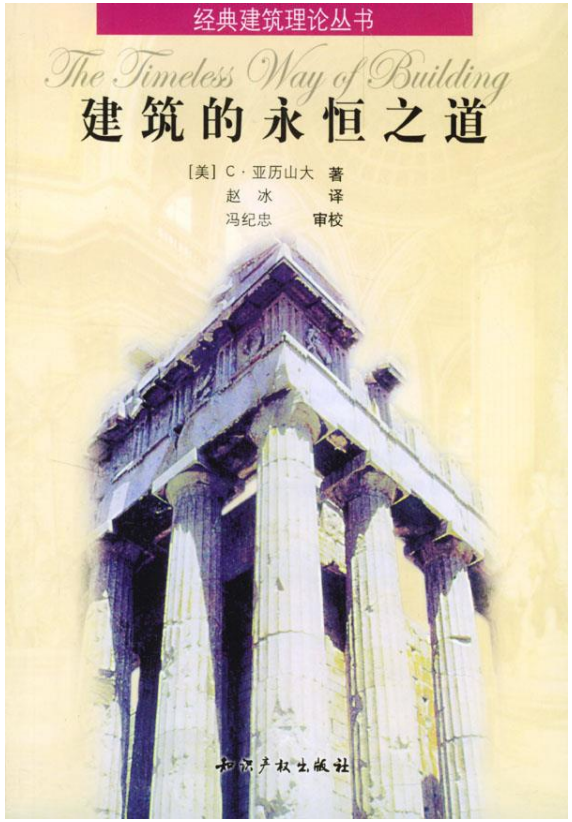
这些原则并不是孤立存在的，它们相互依赖，相互补充。

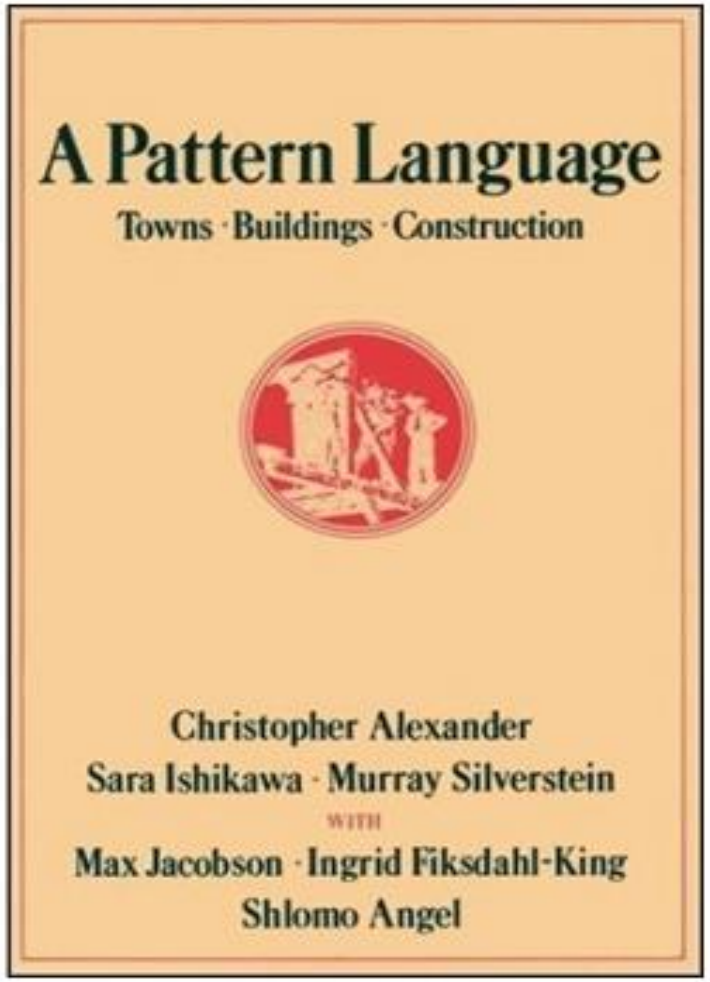
设计原则名称	设计原则简介	重要性
开闭原则 (Open-Closed Principle, OCP)	软件实体对扩展是开放的，但对修改是关闭的，即在不修改一个软件实体的基础上去扩展其功能。	★★★★★
里氏替换原则 (Liskov Substitution Principle, LSP)	在软件系统中，一个可以接受基类对象的地方必然可以接受一个子类对象	★★★★☆
依赖倒置原则 (Dependency Inversion Principle, DIP)	要针对抽象层编程，而不要针对具体类编程	★★★★★
单一职责原则 (Single Responsibility Principle, SRP)	类的职责要单一，不能将太多的职责放在一个类中	★★★★☆
接口隔离原则 (Interface Segregation Principle, ISP)	使用多个专门的接口来取代一个统一的接口	★★☆☆☆
迪米特法则 (Law of Demeter, LoD)	一个软件实体对其他实体的引用越少越好，或者说如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用，而是通过引入一个第三者发生间接交互	★★★☆☆
合成复用原则 (Composite Reuse Principle, CRP)	在系统中应该尽量多使用组合和聚合关联关系，尽量少使用甚至不使用继承关系	★★★★☆

软件工程学科导论-软件设计模式

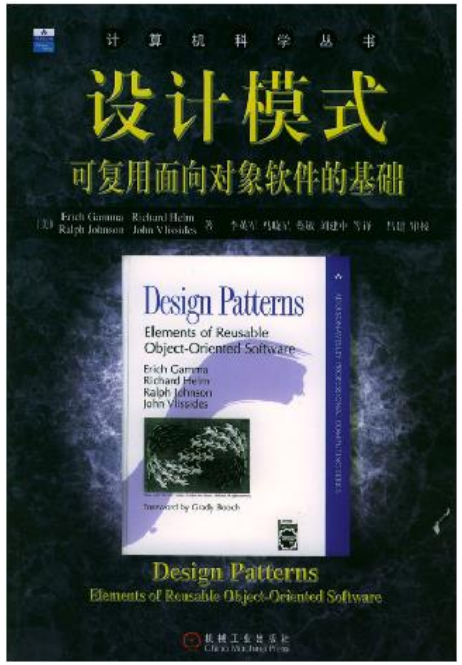
01 软件设计原则
02 软件设计模式
03 设计模式之适配器模式
04 设计模式之装饰器模式
05 设计模式之外观模式
06 设计模式之观察者模式

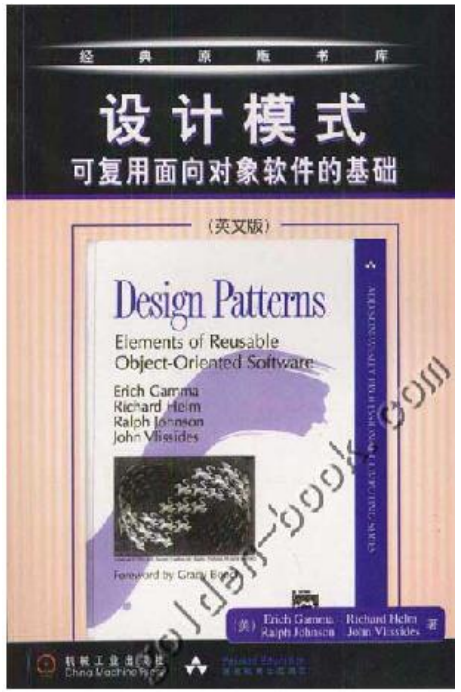
设计模式的诞生与发展





Christopher Alexander





设计模式的诞生与发展

- 模式的诞生与定义
  - 模式起源于建筑业而非软件业
  - 模式(pattern)之父——美国加利佛尼亚大学环境结构中心研究所所长Christopher Alexander博士
  - 《A Pattern Language: Towns, Buildings, Construction》——253个建筑和城市规划模式
  - 模式
    - Context（模式可适用的前提条件）
    - Theme或Problem（在特定条件下要解决的目标问题）
    - Solution（对目标问题求解过程中各种物理关系的记述）

设计模式的诞生与发展

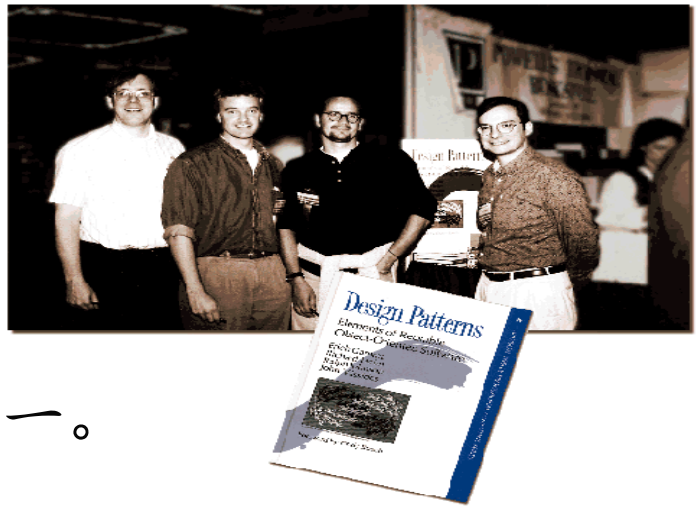
- 模式的诞生与定义
  - ✓Alexander给出了关于模式的经典定义：每个模式都描述了一个在我们的环境中不断出现的问题，然后描述了该问题的解决方案的核心，通过这种方式，我们可以无数次地重用那些已有的解决方案，无需再重复相同的工作。
  - ✓A pattern is a solution to a problem in a context
  - ✓模式是在特定环境中解决问题的一种方案

设计模式的诞生与发展

- 软件模式
  - ✓1990年，软件工程界开始关注Christopher Alexander等在这一住宅、公共建筑与城市规划领域的重大突破，最早将该模式的思想引入软件工程方法学的是1991-1992年以“四人组(Gang of Four，GoF，分别是Erich Gamma，Richard Helm，Ralph Johnson和John Vlissides)”自称的四位著名软件工程学者，他们在1994年归纳发表了23种在软件开发中使用频率较高的设计模式，旨在用模式来统一沟通面向对象方法在分析、设计和实现间的鸿沟。

设计模式的诞生与发展

Gang of Four



Erich Gamma

苏黎世大学计算机科学博士，
Eclipse项目主要技术负责人之一。

Richard Helm

墨尔本大学计算机科学博士，
IBM 研究员。

Ralph Johnson

康奈尔大学计算机科学博士，
伊利诺伊大学教授。

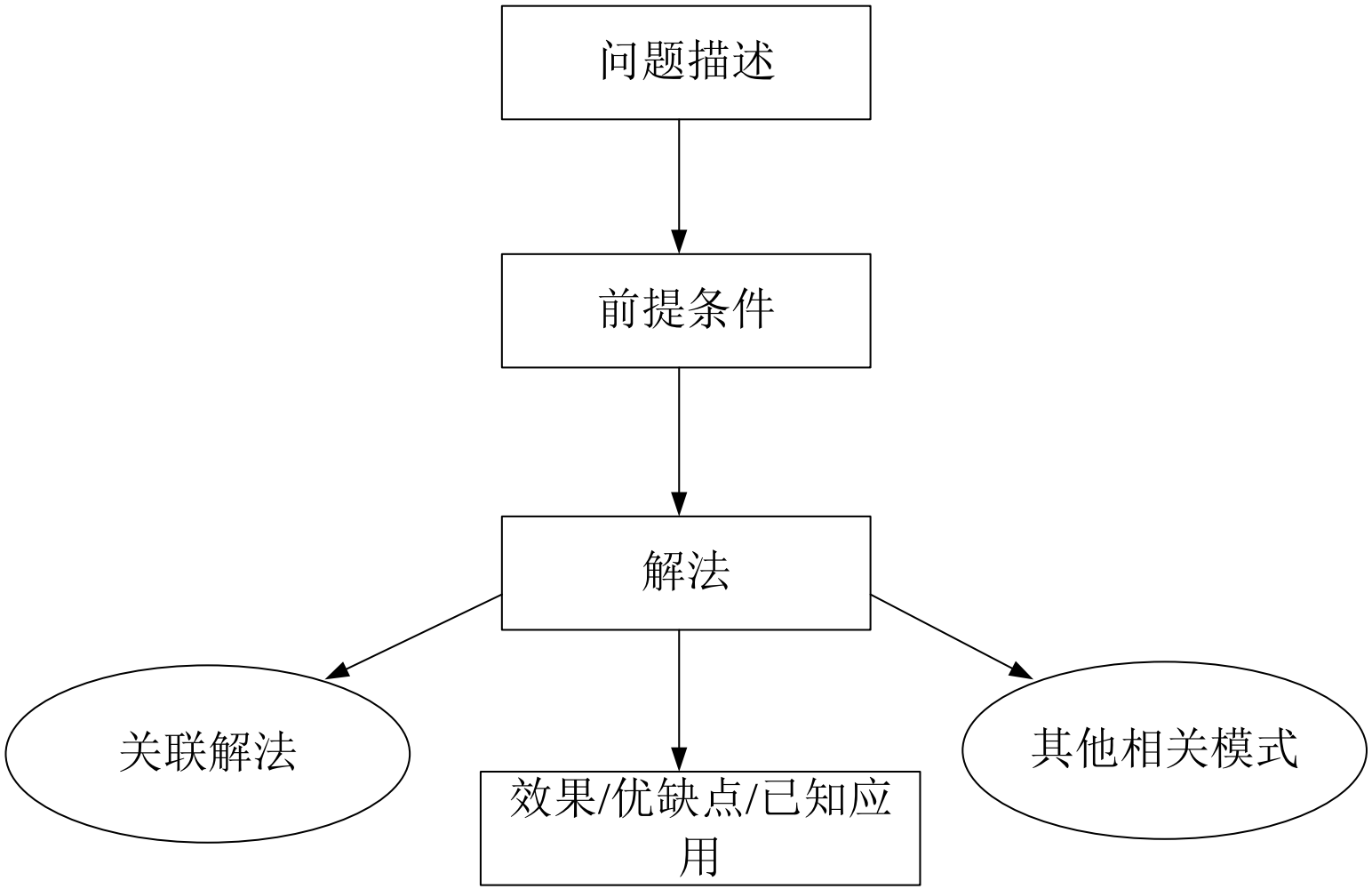
John Vlissides

斯坦福大学计算机科学博士，
原IBM研究员。

设计模式的诞生与发展

- 软件模式
  - 软件模式是将模式的一般概念应用于软件开发领域，即软件开发的总体指导思路或参照样板。软件模式并非仅限于设计模式，还包括架构模式、分析模式和过程模式等，实际上，在软件生存期的每一个阶段都存在着一些被认同的模式。
  - 软件模式可以认为是对软件开发这一特定“问题”的“解法”的某种统一表示，它和Alexander所描述的模式定义完全相同，即软件模式=一定条件下的{问题+解法}。软件模式的基础结构由4个部分构成：问题描述、前提条件（环境或约束条件）、解法和效果。

设计模式的诞生与发展



设计模式的诞生与发展

- 软件模式
  - 软件模式与具体的应用领域无关，在模式发现过程中需要遵循大三律(Rule of Three)，即只有经过三个以上不同类型（或不同领域）的系统的校验，一个解决方案才能从候选模式升格为模式。

设计模式的诞生与发展

设计模式的发展

- 1987年，Kent Beck和Ward Cunningham借鉴Alexander的模式思想在程序开发中开始应用一些模式，在OOPSLA会议上发表了他们的成果。
- 1990年，OOPSLA与ECOOP联合举办，Erich Gamma和Richard Helm等人开始讨论有关模式的话题(Bruce Anderson主持)，“四人组” 正式成立，并开始着手进行设计模式的分类整理工作。
- 1991 年，OOPSLA，Bruce Anderson主持了首次针对设计模式的研讨会 。
- 1992 年，OOPSLA ，Anderson再度主持研讨会，模式已经逐渐成为人们讨论的话题。
- 注：OOPSLA (Object-Oriented Programming, Systems, Languages & Applications, 面向对象编程、系统、语言和应用大会)，编程语言及软件工程国际顶级会议，2010年改为SPLASH --- Systems, Programming, Languages and Applications: Software for Humanity

设计模式的诞生与发展

设计模式的发展

- 1993年，Kent Beck 和 Grady Booch 赞助了第一次关于设计模式的会议，这个设计模式研究组织发展成为著名的Hillside Group研究组。
- 1994 年，由Hillside Group发起，在美国伊利诺伊州(Illinois)的Allerton Park召开了第1届关于面向对象模式的世界性会议，名为PLoP(Pattern Languages of Programs，编程语言模式会议)，简称PLoP ‘94。
- 1995年，PLoP ‘95 仍在伊利诺伊州的Allerton Park举行 ，“四人组”出版了《设计模式：可复用面向对象软件的基础》(Design Patterns: Elements of Reusable Object-Oriented Software)一书，本书成为1995年最抢手的面向对象书籍，也成为设计模式的经典书籍。

设计模式的诞生与发展

设计模式的发展

- 从1995年至今，设计模式在软件开发中得以广泛应用，在Sun的Java SE/Java EE平台和Microsoft的.net平台设计中就应用了大量的设计模式。
- 诞生了越来越多的与设计模式相关的书籍和网站，设计模式也作为一门独立的课程或作为软件体系结构等课程的重要组成部分出现在国内外研究生和大学教育的课堂上。

设计模式定义

Christopher Alexander给出的经典定义是：每个模式都描述了一个在我们的环境中不断出现的问题，然后描述了该问题的解决方案的核心。通过这种方式，你可以无数次地使用那些已有的解决方案，无需再重复相同的工作。

- ✓ 广义讲，软件设计模式是可解决一类软件问题并能重复使用的软件设计方案
- ✓ 狭义讲，设计模式是对被用来在特定场景下解决一般设计问题的类和相互通信的对象的描述。是在类和对象的层次描述的可重复使用的软件设计问题的解决方案
- ✓ 模式体现的是程序整体的构思，所以有时候它也会出现在分析或者是概要设计阶段
- ✓ 模式的核心思想是通过增加抽象层，把变化部分从那些不变部分里分离出来

设计模式定义

- 设计模式的定义
  - ✓ 设计模式**(Design Pattern)**是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结，使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。

设计模式的基本要素

设计模式一般有如下几个基本要素：模式名称，问题，目的，解决方案，效果，实例代码和相关设计模式，其中的关键元素包括以下四个方面：

- 模式名称（pattern name）
- 问题（problem）
- 解决方案（solution）
- 效果（consequences）

模式的基本要素

模式名称（Pattern Name）：每个模式都有一个独一无二的名称，通过名称来鉴别不同的模式。

问题(Problem)：描述应该在何时使用模式。解释了设计问题和问题存在的前因后果,可能还描述模式必须满足的先决条件

解决方案(Solution)：描述了设计的组成成分、相互关系及各自的职责和协作方式。模式就像一个模板，可应用于多种场合，所以解决方案并不描述一个具体的设计或实现，而是提供设计问题的抽象描述和解决问题所采用的元素组合（类和对象）

效果(consequences)：描述模式的应用效果及使用模式应权衡的问题

参与者和协作者：模式所包含的实体。

模式的学习步骤

设计模式学习步骤

一般按照以下次序来学习设计模式：

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

如何描述设计模式？

我们怎样来描述设计模式呢？采用图形概念吗？图形概念固然很重要、很有用，但这还远远不够。因为它们仅仅简单的表达了类和对象之间的关系，这是设计过程的最终产品。为了设计重用，我们还必须记录决策过程、替代方案等。

一般我们用统一的格式来描述设计模式，每一种模式都按以下的模版分成多个部分。每个部分的模版使用统一的信息结构，便于设计模式的学习、比较和使用。

常用的描述模式的格式大致可分为以下部分：

如何描述设计模式？

模式名和分类（Pattern Name and Classification）：模式名简洁的表达了模式的本质。好的命名非常重要，因为它将会成为你的部分设计术语（词汇）。

意图（Intent）：主要描述设计模式的作用？其基本原理和目的是什么？它针对哪些特殊的设计问题？

别名（Byname）：如果某个模式有其它的名称，那么该模版部分就指出了该模式的这个名称。

动机（Motivation）：指出可能存在的设计问题以及怎样使用该模式中的类和对象来解决该问题的情景。这个情景能帮助你理解对该模式更高层的抽象描述。

适用性（Applicability）：指出模式适用于哪些情况？该模式可用于对那些不良的设计进行改进，以及怎么才能识别这种情况？

如何描述设计模式？

结构（Structure）：指出基于对象模型技术（OMT）对该模式的图形表示，以及使用交互图（对象间请求流程的示意图）来表示对象的请求顺序和协作。

参与者（Participants）：指出参与该模式的类和对象以及各自的职责。

协作（Collaboration）：指出参与者为了完成各自的职责应该如何协作。

结果（Consequences）：指出模式达成目标的程度、应用该模式的结果和费用，以及系统结构是否允许你改变其中的某个或某些方面，具体是哪些方面？

实现（Implementation）：指出在实现该模式时，应当具备的前提和技术，以及该模式有什么缺陷？是否具有与语言无关的特性？

如何描述设计模式？

例程（Sample Code）：指出如何使用编程语言来实现该模式。

已知应用（Known Uses）：指出实际系统中已经使用了该模式的例子，一般至少包括两个不同领域的例子。

相关模式（Related Patterns）：指出哪些设计模式与该模式紧密相关？有什么重要的不同？以及该模式应当与哪个或哪些模式一起应用。

应用设计模式解决问题

对于面向对象设计者经常遇到的一些问题，设计模式可采用多种方法来解决，比如对变化性的封装就是许多设计模式的主题，以下列举了这些问题中的几种。

- 寻找合适的对象
- 决定对象的粒度
- 指定对象的接口
- 描述对象的实现
- 运用复用机制
- 运行时刻和编译时刻的结构
- 设计应支持变化

设计模式可以确保系统能以特定的方式变化，从而避免重新设计。

应用设计模式解决问题

- 寻找合适的对象
- 决定对象的粒度
- 指定对象的接口
- 描述对象的实现
- 运用复用机制
- 运行时刻和编译时刻的结构
- 设计应支持变化

寻找适合的对象

面向对象程序由对象组成，对象包含数据和方法。对象的内部数据和方法是被封装的，不能被直接访问，它的表示对于对象外部是不可见的，客户只能通过对象的请求执行相应的操作。

面向对象设计最困难的部分是将系统分解成对象集合。因为在分解的同时需要考虑许多因素：封装、粒度、依赖关系、灵活性、性能、演化、复用等等，它们都影响着系统的分解，并且这些因素通常还是互相冲突的。而许多对象都来源于现实世界的分析模型，但是所得到的类通常在现实世界中并不存在。此时设计模式就能确定那些并不明显的抽象和描述这些抽象的对象

应用设计模式解决问题

- 寻找合适的对象
- 决定对象的粒度
- 指定对象的接口
- 描述对象的实现
- 运用复用机制
- 运行时刻和编译时刻的结构
- 设计应支持变化

决定对象的粒度

对象在大小和数目上变化极大，能表示任何事物，那么我们怎么来确定一个对象呢？设计模式能很好的处理这个问题

指定对象的接口

对象接口描述了该对象所能接受的全部请求的集合，任何匹配对象接口的请求都可以发送给该对象。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情况，也无法请求对象做任何事情。设计模式就能通过确定接口的主要组成成分以及经接口发送的数据类型，来定义接口。而且设计模式也指定了接口之间的关系。

应用设计模式解决问题

- 寻找合适的对象
- 决定对象的粒度
- 指定对象的接口
- 描述对象的实现
- 运用复用机制
- 运行时刻和编译时刻的结构
- 设计应支持变化

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方法，通过实例化类来创建对象，该对象被称为该类的实例。而在很多时候却不将变量声明为某个特定的具体类的实例对象，而是让它遵循抽象类所定义的接口

抽象类（abstract class）的主要目的是为它的子类定义公共接口。一个抽象类将把它的部分或全部操作的实现延迟到子类中，因此，一个抽象类不能被实例化。

而当不得不在某个地方实例化具体的类时，设计模式就可以帮你，通过抽象对象的创建过程，在实例化时建立接口和实现的透明连接。

应用设计模式解决问题

- 寻找合适的对象
- 决定对象的粒度
- 指定对象的接口
- 描述对象的实现
- 运用复用机制
- 运行时刻和编译时刻的结构
- 设计应支持变化

复用技术主要有：

- 类继承
- 对象组合
- 委托
- 参数化类型

面向对象系统中功能复用的两种最常用技术是类继承和对象组合(object composition)。

委托（delegation）是一种组合方法，它使组合具有与继承同样的复用能力。

参数化类型(parameterized type)是一种不常用到的复用技术

类继承与对象组合

类继承是在编译时刻静态定义的，且可直接使用，可以较方便地改变被复用的实现。但是继承在编译时就已经定义了，所以无法在运行时改变从父类继承的实现，更糟的是，父类通常至少定义了部分子类的具体表示。因为继承对子类揭示了其父类的实现细节，所以继承常被认为“破坏了封装性”。

对象组合是通过获得对其他对象的引用而在运行时刻动态定义的。组合要求对象遵守彼此的接口约定，进而要求更仔细地定义接口，而这些接口并不妨碍你将一个对象和其他对象一起使用。

对象组合对系统设计还有另一个作用，即优先使用对象组合有助于你保持每个类被封装，并被集中在单个任务上。

一般在实际中应该优先使用对象组合，而不是类继承。

委　　托

委托（delegation）是一种组合方法，它使组合具有与继承同样的复用能力。在委托方式下，有两个对象参与处理一个请求，接受请求的对象将操作委托给它的代理者（delegate）。

比如可以在类A中保存一个类B的实例来代理类B的特定操作，这样类A可以复用类B的操作，而不必像类继承那样定义成类B的子类。

委托的主要优点在于它便于运行时刻组合对象操作以及改变这些操作的组合方式。有些设计模式使用了委托，通过改变委托对象来改变委托对象的行为。

注意：委托是对象组合技术的一个特例，它使我们了解到对象组合作为一个代码复用机制也可以替代继承。

参数化类型

参数化类型(parameterized type)也是一种复用技术，它允许在定义一个类型时并不指定该类型所用到的其他所有类型。未经指定的类型在使用时以参数形式提供。

对象组合技术允许在运行时刻改变被组合的行为，但是它存在间接性，效率比较低。继承允许提供操作的缺省实现，并通过子类重定义这些操作。参数化类型允许改变类所用到的类型。但是继承和参数化类型都不能在运行时刻改变。

运行时刻和编译时刻的结构

- 寻找合适的对象
- 决定对象的粒度
- 指定对象的接口
- 描述对象的实现
- 运用复用机制
- 运行时刻和编译时刻的结构
- 设计应支持变化

聚合意味着一个对象包含另一个对象或者是另一个对象的一部分。聚合对象和其所有者具有相同的生命周期。

相识（也称为“关联”或“引用”）意味着一个对象仅仅知道另一个对象。相识的对象可能请求彼此的操作，但是它们不为对方负责。相识是一种比聚合要弱的关系。

是聚合还是相识是由设计者的意图决定的，而不是由显式的语言机制决定的。他们在编译时刻和运行时刻有很大的区别，程序的编译时刻和运行时刻也存在着很大的区别，但代码不能揭示系统工作的全部，而许多设计模式就能显式的记述编译时刻和运行时刻结构的差别。

以下是导致重新设计的几个方面：

- 通过显式地指定一个类来创建对象
- 对特殊操作的依赖
- 对硬件和软件平台的依赖
- 对对象表示或实现的依赖
- 算法依赖
- 紧耦合
- 通过生成子类来扩充功能
- 不能方便地对类进行修改

选择设计模式

以下提供几个选择设计模式的参考方法：

- 考虑设计模式怎样解决设计问题
- 浏览模式的意图部分
- 研究模式怎样互相关联
- 研究目的相似的模式
- 检查重新设计的原因
- 考虑你的设计过程中哪些是可变的

使用设计模式

对于设计模式的使用可以从以下几个方面来考虑：

浏览一遍模式 ，比如模式名、意图、适用性、效果等

研究结构部分、参与者部分和协作部分

查看代码示例部分，分析该模式代码形式的具体例子

选择模式参与者的名字，使它们在应用上下文中有意义

定义类，声明它们的接口，建立它们的继承关系，
定义代表数据和对象引用的实例变量。识别模式
会影响到你的应用中存在的类，做出相应的修改。

定义模式中专用于应用的操作名称

实现执行模式责任和协作的操作

设计模式的优点

设计模式是从许多优秀的软件系统中总结出的成功的、能够实现可维护性复用的设计方案，使用这些方案将避免我们做一些重复性的工作，而且可以设计出高质量的软件系统。具体来说，设计模式的主要优点如下：

（1）设计模式融合了众多专家的经验，并以一种标准的形式供广大开发人员所用，它提供了一套通用的设计词汇和一种通用的语言以方便开发人员之间沟通和交流，使得设计方案更加通俗易懂。对于使用不同编程语言的开发和设计人员可以通过设计模式来交流系统设计方案，每一个模式都对应一个标准的解决方案，设计模式可以降低开发人员理解系统的复杂度。

（2）设计模式使人们可以更加简单方便地复用成功的设计和体系结构，将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。设计模式使得重用成功的设计更加容易，并避免那些导致不可重用的设计方案。

（3）设计模式使得设计方案更加灵活，且易于修改。

（4）设计模式的使用将提高软件系统的开发效率和软件质量，且在一定程度上节约设计成本。

（5）设计模式有助于初学者更深入地理解面向对象思想，一方面可以帮助初学者更加方便地阅读和学习现有类库与其他系统中的源代码，另一方面还可以提高软件的设计水平和代码质量。

常用设计模式分析

常用的基本设计模式有23种，可分为三个类型：

- 创建型设计模式
  - 描述了实例化对象的相关技术，解决了创建对象的有关问题。
- 结构型设计模式
  - 描述了在软件系统中组织类和对象的常用方法，避免了一个类被赋予过多职责而破坏类的封装性和信息的隐藏，以及类之间功能重叠的问题。
- 行为设计模式
  - 负责分配对象的职责，为对象间协作建模提供有效的策略。

主要有以下几种：

Abstract Factory模式 ：为创建一系列相关的或者相互依赖的对象配置接口，而不必指定它们具体的类。

Builder模式 ：将一个复杂的对象构建与它的表示分离开来，使用同样的构建过程可以创建不同的表示。

Factory Method模式 ：定义了一个创建对象的接口，但是却让子类来决定具体实例化哪一个类。

Prototype模式 ：使用原型实例指定创建对象的种类，而通过拷贝这个原型实例来创建新的实例。

Singleton模式 ：保证在运行的应用程序中，一个Class只实例化一次，并提供一个访问它的全局访问点。

在创建型模式中除了Factory Method（工厂方法模式）属于类模式，其他都属于对象模式。

主要有以下几种：

Adapter模式 ：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法一起使用的两个类能够一起使用 。

Bridge模式 ：将一个复杂对象的构建与它的表示分离，使得二者可以独立变化。

Composite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。

Decorator模式 ：动态给一个对象增加功能，这些功能可以再动态地撤消。

Facade模式 ：为一个复杂系统提供了一个简单的接口。即提供一个高层次的接口，使得系统更易于使用。

结构型设计模式

Flyweight模式 ：创建共享对象的一个小型池，以共享的方式高效的支持大量的细粒度对象。

Bridge模式 ：给某一个对象提供一个代理对象，并由代理对象控制对原来对象的引用。

在结构型模式中除了适配器（类Adapter）属于类模式，其他都属于对象模式。

行为设计模式

主要有以下几种：

Chain of Responsibility模式 ：将很多对象连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求为止。

Command模式：把一个请求或者操作封装到一个对象中，允许请求的一方和发送的一方独立开来。

Interpreter模式 ：给定一个语言后，该模式可以定义出其文法的一种表示，并同时提供一个解释器。

Iterator模式 ：分离一个集合对象的遍历行为，抽象出一个迭代器类来负责。

Mediator模式 ：用一个中介对象来封装一系列的对象交互，使得这些对象不必相互明显作用。从而使它们可以松散耦合 。

Memento模式 ：在不破坏封装的前提下，捕获一个对象的内部状态，并外部化，存储起来。

Observer模式 ：定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。

State模式 ：允许一个对象在其内部状态改变时，改变行为。

Strategy模式 ：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。

Template Method模式 ：为一系列的算法先制定一个顶级算法框架，而将算法的细节留给具体的子类去实现。

Visitor模式 ：封装一些作用于某种数据结构元素之上的操作。一旦这些操作需要修改的话，接受这个操作的数据结构可以保持不变。

在结构型模式中除了解释器（Interpreter）模板方法（Template Method）属于类模式，其他都属于对象模式。

设计模式的分类

- ✓ 根据其目的（模式是用来做什么的）可分为**创建型(Creational)**，**结构型(Structural)**和**行为型(Behavioral)**三种：
  - 创建型模式主要用于创建对象。
  - 结构型模式主要用于处理类或对象的组合。
  - 行为型模式主要用于描述对类或对象怎样交互和怎样分配职责。

GoF设计模式简介

- 创建型模式
  - ✓ 抽象工厂模式(Abstract Factory)
  - ✓ 建造者模式(Builder)
  - ✓ 工厂方法模式(Factory Method)
  - ✓ 原型模式(Prototype)
  - ✓ 单例模式(Singleton)

GoF设计模式简介

- 结构型模式
  - ✓ 适配器模式(Adapter)
  - ✓ 桥接模式(Bridge)
  - ✓ 组合模式(Composite)
  - ✓ 装饰模式(Decorator)
  - ✓ 外观模式(Facade)
  - ✓ 享元模式(Flyweight)
  - ✓ 代理模式(Proxy)

GoF设计模式简介

- 行为型模式
  - ✓ 职责链模式(Chain of Responsibility)
  - ✓ 命令模式(Command)
  - ✓ 解释器模式(Interpreter)
  - ✓ 迭代器模式(Iterator)
  - ✓ 中介者模式(Mediator)
  - ✓ 备忘录模式(Memento)
  - ✓ 观察者模式(Observer)
  - ✓ 状态模式(State)
  - ✓ 策略模式(Strategy)
  - ✓ 模板方法模式(Template Method)
  - ✓ 访问者模式(Visitor)

设计模式的优点

- 设计模式是从许多优秀的软件系统中总结出的成功的、能够实现可维护性复用的设计方案，使用这些方案将避免我们做一些重复性的工作，而且可以设计出高质量的软件系统。具体来说，设计模式的主要优点如下：
  - ✓ 设计模式融合了众多专家的经验，并以一种标准的形式供广大开发人员所用，它提供了一套通用的设计词汇和一种通用的语言以方便开发人员之间沟通和交流，使得设计方案更加通俗易懂。对于使用不同编程语言的开发和设计人员可以通过设计模式来交流系统设计方案，每一个模式都对应一个标准的解决方案，设计模式可以降低开发人员理解系统的复杂度。

设计模式的优点

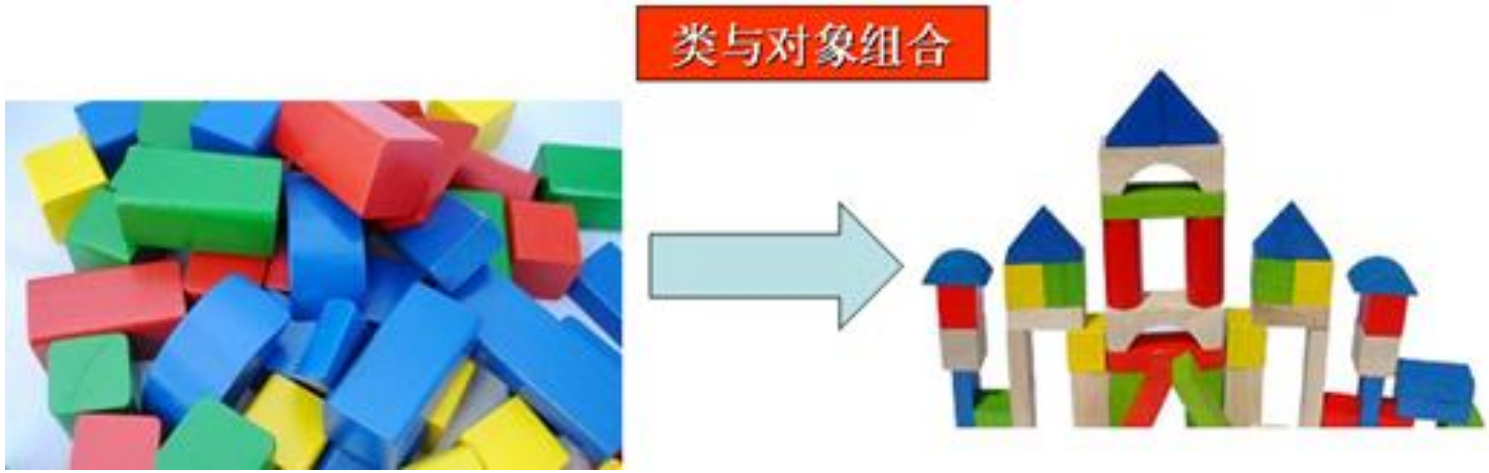
- ✓ 设计模式使人们可以更加简单方便地复用成功的设计和体系结构，将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。设计模式使得重用成功的设计更加容易，并避免那些导致不可重用的设计方案。
- ✓ 设计模式使得设计方案更加灵活，且易于修改。
- ✓ 设计模式的使用将提高软件系统的开发效率和软件质量，且在一定程度上节约设计成本。
- ✓ 设计模式有助于初学者更深入地理解面向对象思想，一方面可以帮助初学者更加方便地阅读和学习现有类库与其他系统中的源代码，另一方面还可以提高软件的设计水平和代码质量。

软件工程学科导论-软件设计模式

01 软件设计原则
02 软件设计模式
03 设计模式之适配器模式
04 设计模式之装饰器模式
05 设计模式之外观模式
06 设计模式之观察者模式

结构型模式

结构型模式(Structural Pattern)
描述如何将类或者对象结合在一起形成更大的结构，就像搭积木，可以通过简单积木的组合形成复杂的、功能更为强大的结构。



结构型模式

结构型模式可以分为类结构型模式和对象结构型模式：

- 类结构型模式关心类的组合，由多个类可以组合成一个更大的系统，在类结构型模式中一般只存在继承关系和实现关系。

- 对象结构型模式关心类与对象的组合，通过关联关系使得在一个类中定义另一个类的实例对象，然后通过该对象调用其方法。根据“合成复用原则”，在系统中尽量使用关联关系来替代继承关系，因此大部分结构型模式都是对象结构型模式。

结构型模式

适配器模式(Adapter)
桥接模式(Bridge)
组合模式(Composite)
装饰模式(Decorator)
外观模式(Facade)
享元模式(Flyweight)
代理模式(Proxy)

结构型模式—适配器模式

模式动机

在软件开发中采用类似于电源适配器的设计和编码技巧被称为适配器模式。

通常情况下，客户端可以通过目标类的接口访问它所提供的服务。有时，现有的类可以满足客户类的功能需要，但是它所提供的接口不一定是客户类所期望的，这可能是因为现有类中方法名与目标类中定义的方法名不一致等原因所导致的。

在这种情况下，现有的接口需要转化为客户类期望的接口，这样保证了对现有类的重用。如果不进行这样的转化，客户类就不能利用现有类所提供的功能，适配器模式可以完成这样的转化。

结构型模式—适配器模式

模式应用意义

图一　说明适配器模式应用领域之一
包含n个企业的大型企业试图以
一个窗口面向企业用户。

图二　说明适配器模式应用领域之二
构建一个涵盖所有通，所有风的物流企业
集成窗口。

结构型模式—适配器模式

模式动机

在适配器模式中可以定义一个包装类，包装不兼容接口的对象，这个包装类指的就是适配器(Adapter)，它所包装的对象就是适配者(Adaptee)，即被适配的类。

适配器提供客户类需要的接口，适配器的实现就是把客户类的请求转化为对适配者的相应接口的调用。也就是说：当客户类调用适配器的方法时，在适配器类的内部将调用适配者类的方法，而这个过程对客户类是透明的，客户类并不直接访问适配者类。因此，适配器可以使由于接口不兼容而不能交互的类可以一起工作。这就是适配器模式的模式动机。

结构型模式—适配器模式

模式定义

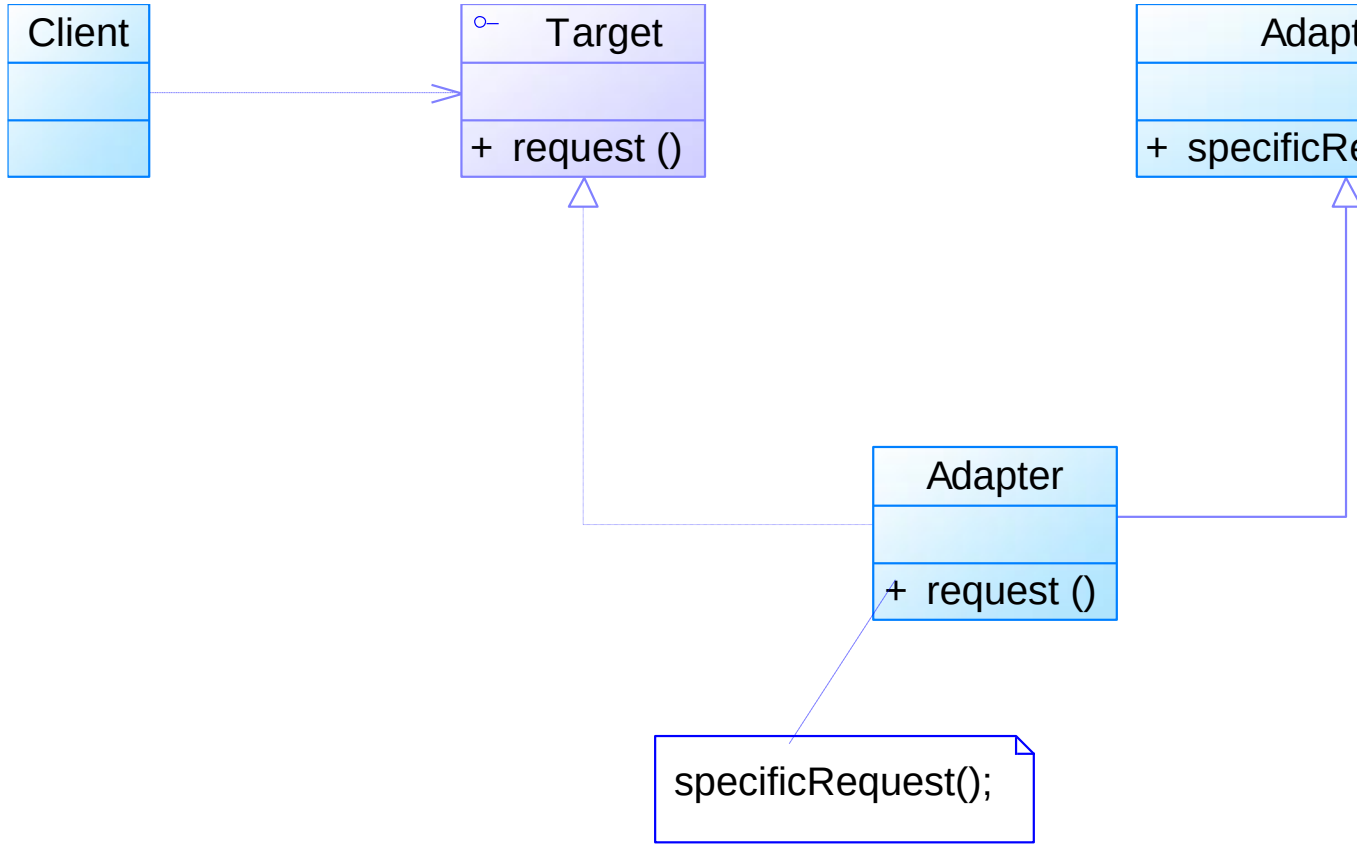
适配器模式(Adapter Pattern)：将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为包装器(Wrapper)。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。

Adapter Pattern: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Frequency of use: medium high

结构型模式—适配器模式

模式结构-类适配器



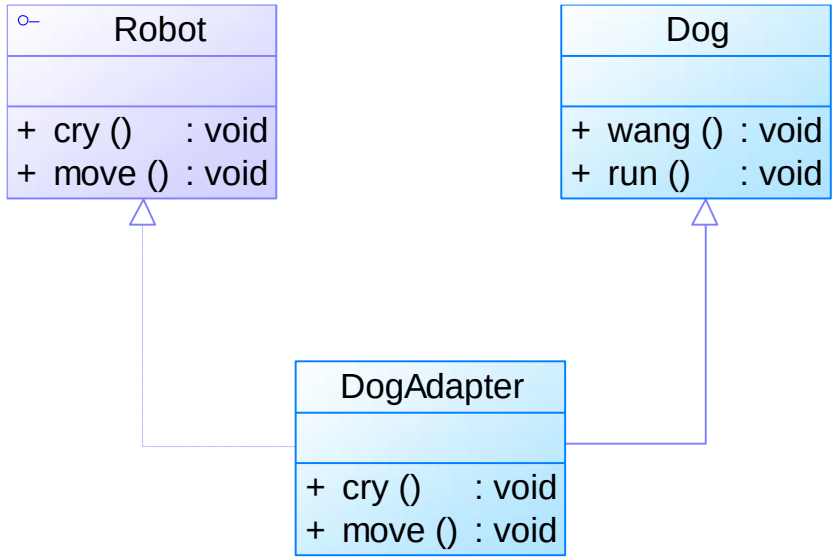
适配器模式包含如下角色：

- Target：目标抽象类
- Adapter：适配器类
- Adaptee：适配者类
- Client：客户类

结构型模式—适配器模式

仿生机器人

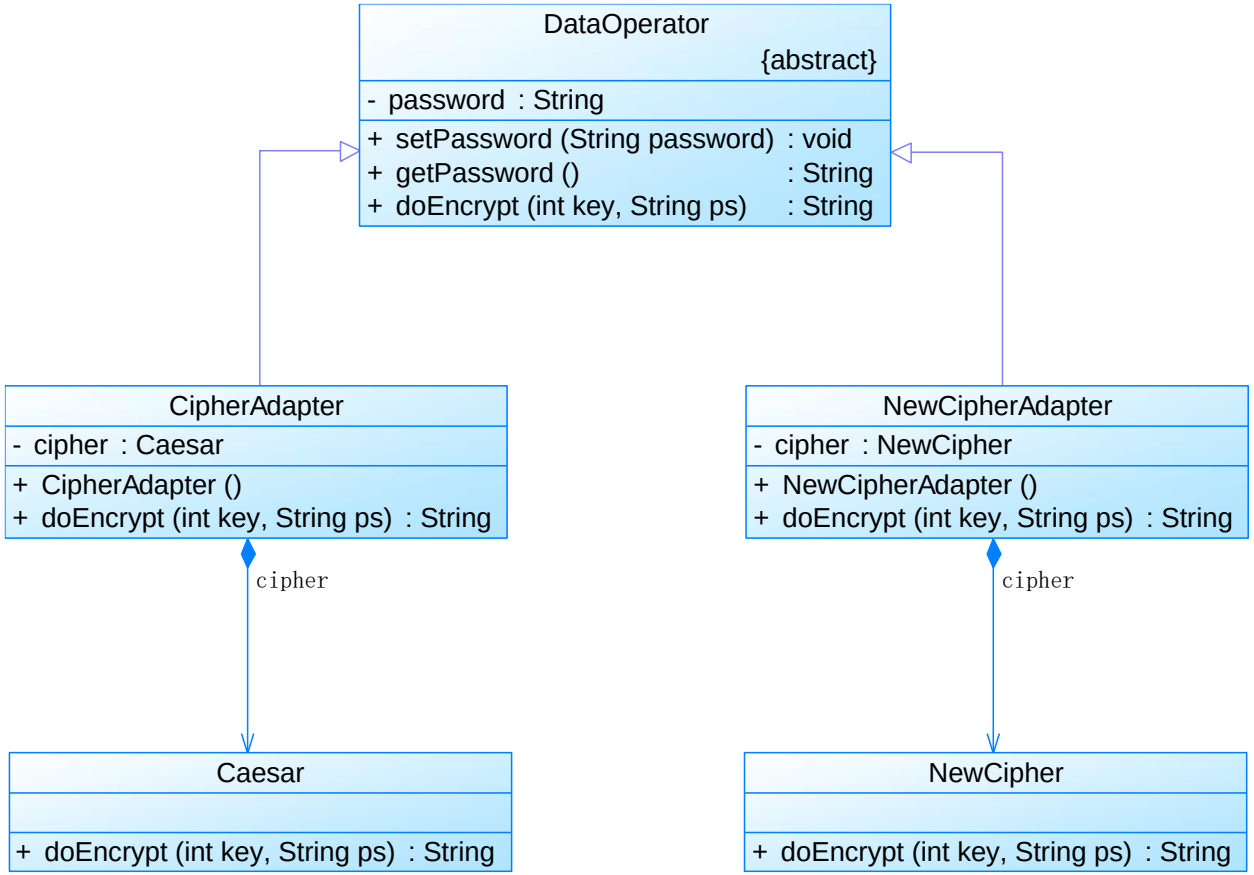
现需要设计一个可以模拟各种动物行为的机器人，在机器人中定义了一系列方法，如机器人叫喊方法cry()、机器人移动方法move()等。如果希望在不修改已有代码的基础上使得机器人能够像狗一样叫，像狗一样跑，使用适配器模式进行系统设计。



结构型模式—适配器模式

加密适配器

某系统需要提供一个加密模块，将用户信息（如密码等机密信息）加密之后再存储在数据库中，系统已经定义好了数据库操作类。为了提高开发效率，现需要重用已有的加密算法，这些算法封装在一些由第三方提供的类中，有些甚至没有源代码。使用适配器模式设计该加密模块，实现在不修改现有类的基础上重用第三方加密方法



结构型模式—适配器模式

适配器模式优缺点

- 将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，而无须修改原有代码。
- 增加了类的透明性和复用性，将具体的实现封装在适配者类中，对于客户端类来说是透明的，而且提高了适配者的复用性。
- 灵活性和扩展性都非常好，通过使用配置文件，可以很方便地更换适配器，也可以在不修改原有代码的基础上增加新的适配器类，完全符合“开闭原则”
- 由于适配器类是适配者类的子类，因此可以在适配器类中置换一些适配者的方法，使得适配器的灵活性更强。

- 对于Java、C#等不支持多重继承的语言，一次最多只能适配一个适配者类，而且目标抽象类只能为抽象类，不能为具体类，其使用有一定的局限性，不能将一个适配者类和它的子类都适配到目标接口。

结构型模式—适配器模式

适配器模式适用环境

- 系统需要使用现有的类，而这些类的接口不符合系统的需要。
- 想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。

适配器模式小结

适配器模式用于将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为包装器。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。

适配器模式包含四个角色：目标抽象类定义客户要用的特定领域的接口；适配器类可以调用另一个接口，作为一个转换器，对适配者和抽象目标类进行适配，它是适配器模式的核心；适配者类是被适配的角色，它定义了一个已经存在的接口，这个接口需要适配；在客户类中针对目标抽象类进行编程，调用在目标抽象类中定义的业务方法。

在类适配器模式中，适配器类实现了目标抽象类接口并继承了适配者类，并在目标抽象类的实现方法中调用所继承的适配者类的方法；在对象适配器模式中，适配器类继承了目标抽象类并定义了一个适配者类的对象实例，在所继承的目标抽象类方法中调用适配者类的相应业务方法。

软件工程学科导论-软件设计模式

01 软件设计原则
02 软件设计模式
03 设计模式之适配器模式
04 设计模式之装饰器模式
05 设计模式之外观模式
06 设计模式之观察者模式

结构型模式—装饰器模式

模式动机

一般有两种方式可以实现给一个类或对象增加行为：

- 继承机制，使用继承机制是给现有类添加功能的一种有效途径，通过继承一个现有类可以使得子类在拥有自身方法的同时还拥有父类的方法。但是这种方法是静态的，用户不能控制增加行为的方式和时机。

- 关联机制，即将一个类的对象嵌入另一个对象中，由另一个对象来决定是否调用嵌入对象的行为以便扩展自己的行为，我们称这个嵌入的对象为装饰器(Decorator)。

结构型模式—装饰器模式

模式动机

装饰模式以对客户透明的方式动态地给一个对象附加上更多的责任，换言之，客户端并不会觉得对象在装饰前和装饰后有什么不同。装饰模式可以在不需要创造更多子类的情况下，将对象的功能加以扩展。这就是装饰模式的模式动机。

结构型模式—装饰器模式

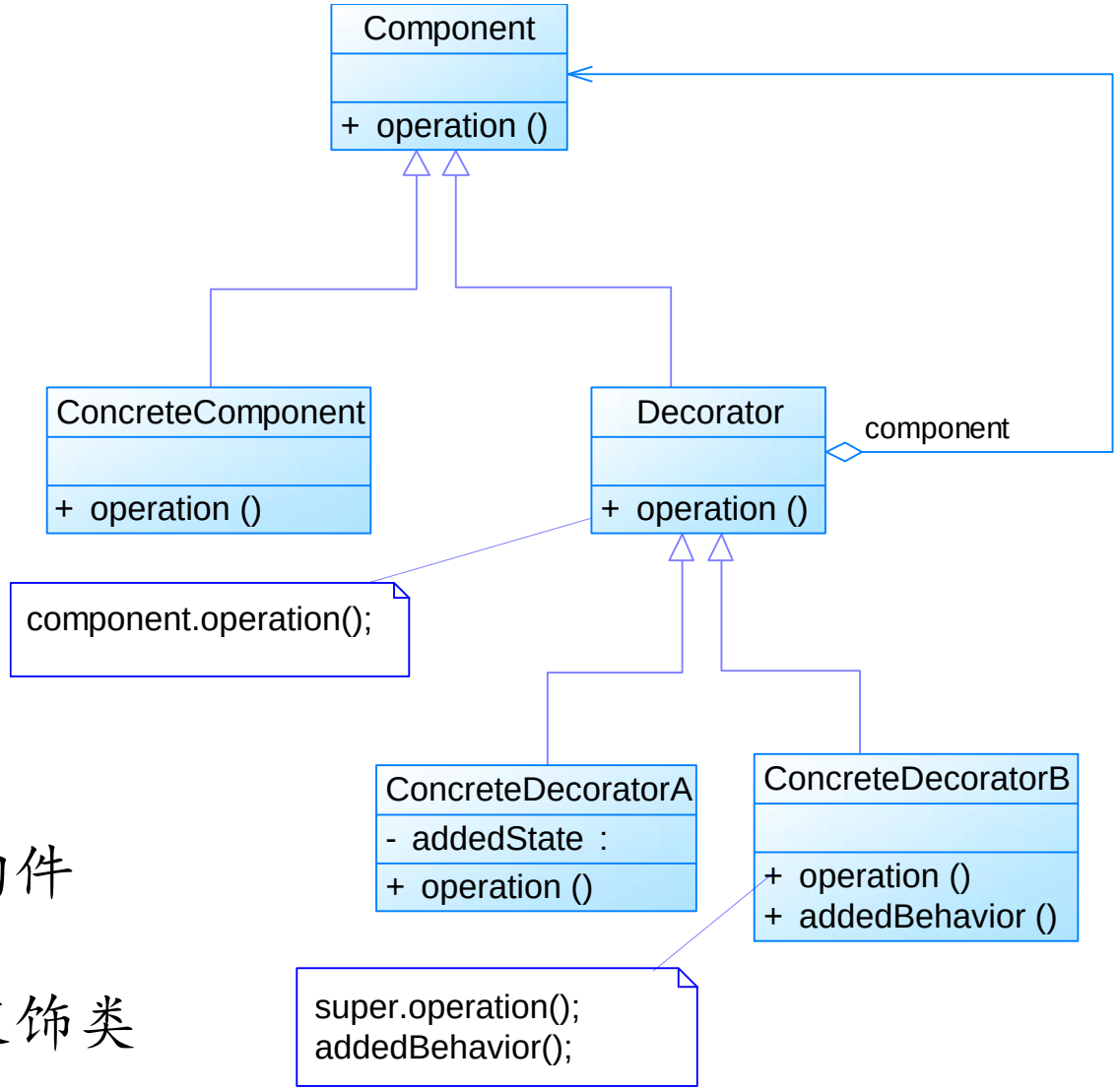
模式定义

装饰模式(Decorator Pattern)：动态地给一个对象增加一些额外的职责(Responsibility)，就增加对象功能来说，装饰模式比生成子类实现更为灵活。其别名也可以称为包装器(Wrapper)，与适配器模式的别名相同，但它们适用于不同的场合。根据翻译的不同，装饰模式也有人称之为“油漆工模式”，它是一种对象结构型模式。

Decorator Pattern: Attach additional responsibilities to an object dynamically.
Decorators provide a flexible alternative to subclassing for extending functionality.
Frequency of use: medium

结构型模式—装饰器模式

模式结构



装饰模式包含如下角色：

- Component：抽象构件
- ConcreteComponent：具体构件
- Decorator：抽象装饰类
- ConcreteDecorator：具体装饰类

结构型模式—装饰器模式

模式分析

与继承关系相比，关联关系的主要优势在于不会破坏类的封装性，而且继承是一种耦合度较大的静态关系，无法在程序运行时动态扩展。在软件开发阶段，关联关系虽然不会比继承关系减少编码量，但是到了软件维护阶段，由于关联关系使系统具有较好的松耦合性，因此使得系统更加容易维护。当然，关联关系的缺点是比继承关系要创建更多的对象。

使用装饰模式来实现扩展比继承更加灵活，它以对客户透明的方式动态地给一个对象附加更多的责任。装饰模式可以在不需要创造更多子类的情况下，将对象的功能加以扩展。

结构型模式—装饰器模式

模式分析-典型的抽象装饰类代码

```
public class Decorator extends Component
{
    private Component component;
    public Decorator(Component component)
    {
        this.component=component;
    }
    public void operation()
    {
        component.operation();
    }
}
```

结构型模式—装饰器模式

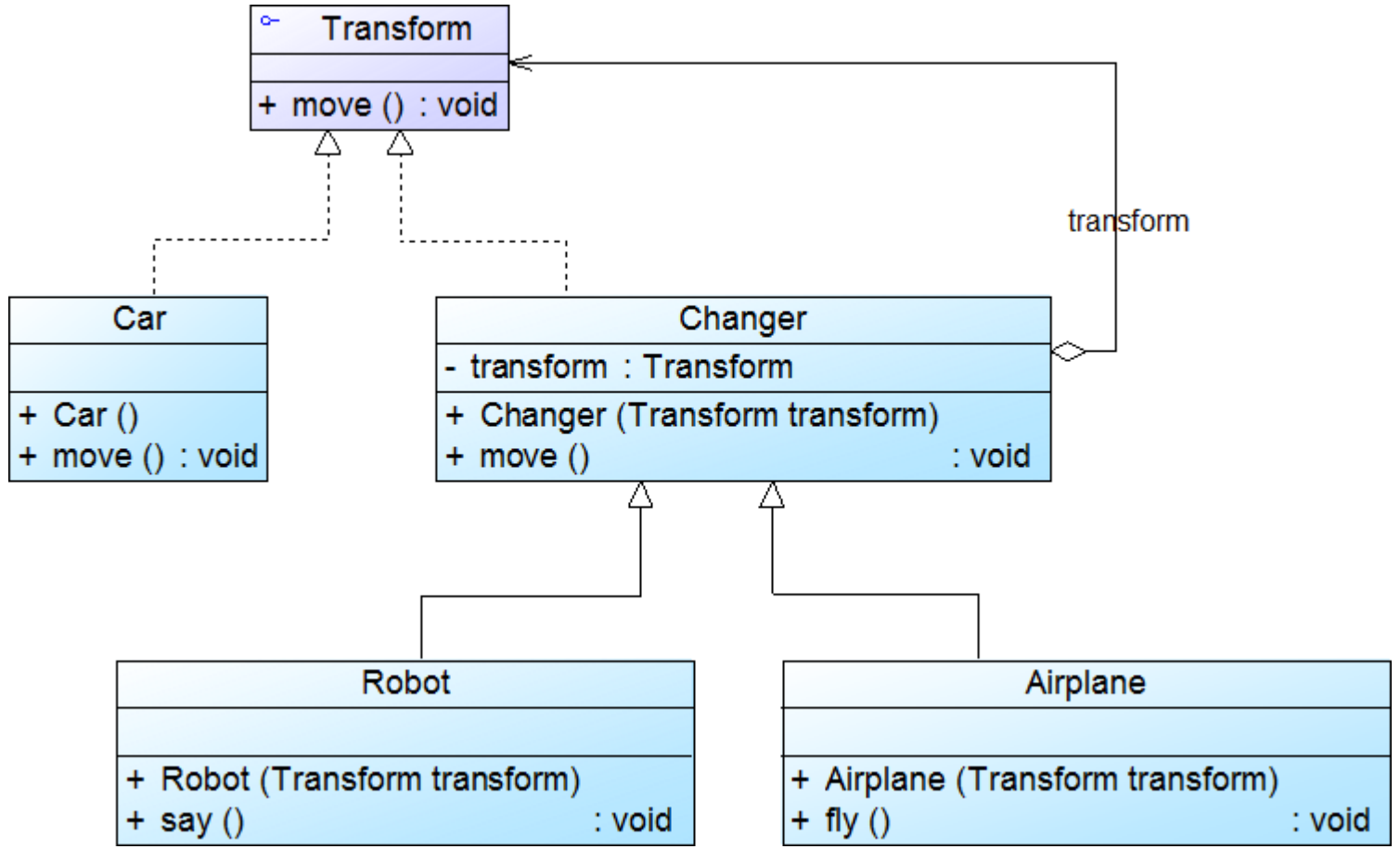
模式分析-典型的实例

实例一：变形金刚

- 变形金刚在变形之前是一辆汽车，它可以在陆地上移动。当它变成机器人之后除了能够在陆地上移动之外，还可以说话；如果需要，它还可以变成飞机，除了在陆地上移动还可以在天空中飞翔。

结构型模式—装饰器模式

模式分析-典型的实例



结构型模式—装饰器模式

模式优缺点

- 装饰模式的优点
  - 装饰模式与继承关系的目的都是要扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。
  - 可以通过一种动态的方式来扩展一个对象的功能，通过配置文件可以在运行时选择不同的装饰器，从而实现不同的行为。
  - 通过使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同行为的组合。可以使用多个具体装饰类来装饰同一对象，得到功能更为强大的对象。
  - 具体构件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类，在使用时再对其进行组合，原有代码无须改变，符合“开闭原则”。

结构型模式—装饰器模式

模式优缺点

装饰模式的缺点

- 使用装饰模式进行系统设计时将产生很多小对象，这些对象的区别在于它们之间相互连接的方式有所不同，而不是它们的类或者属性值有所不同，同时还将产生很多具体装饰类。这些装饰类和小对象的产生将增加系统的复杂度，加大学习与理解的难度。

- 这种比继承更加灵活机动的特性，也同时意味着装饰模式比继承更加易于出错，排错也很困难，对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为烦琐。

结构型模式—装饰器模式

模式适用环境

在以下情况下可以使用装饰模式：

- 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
- 需要动态地给一个对象增加功能，这些功能也可以动态地被撤销。

- 当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展和维护时。不能采用继承的情况主要有两类：第一类是系统中存在大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长；第二类是因为类定义不能继承（如final类）。

结构型模式—装饰器模式

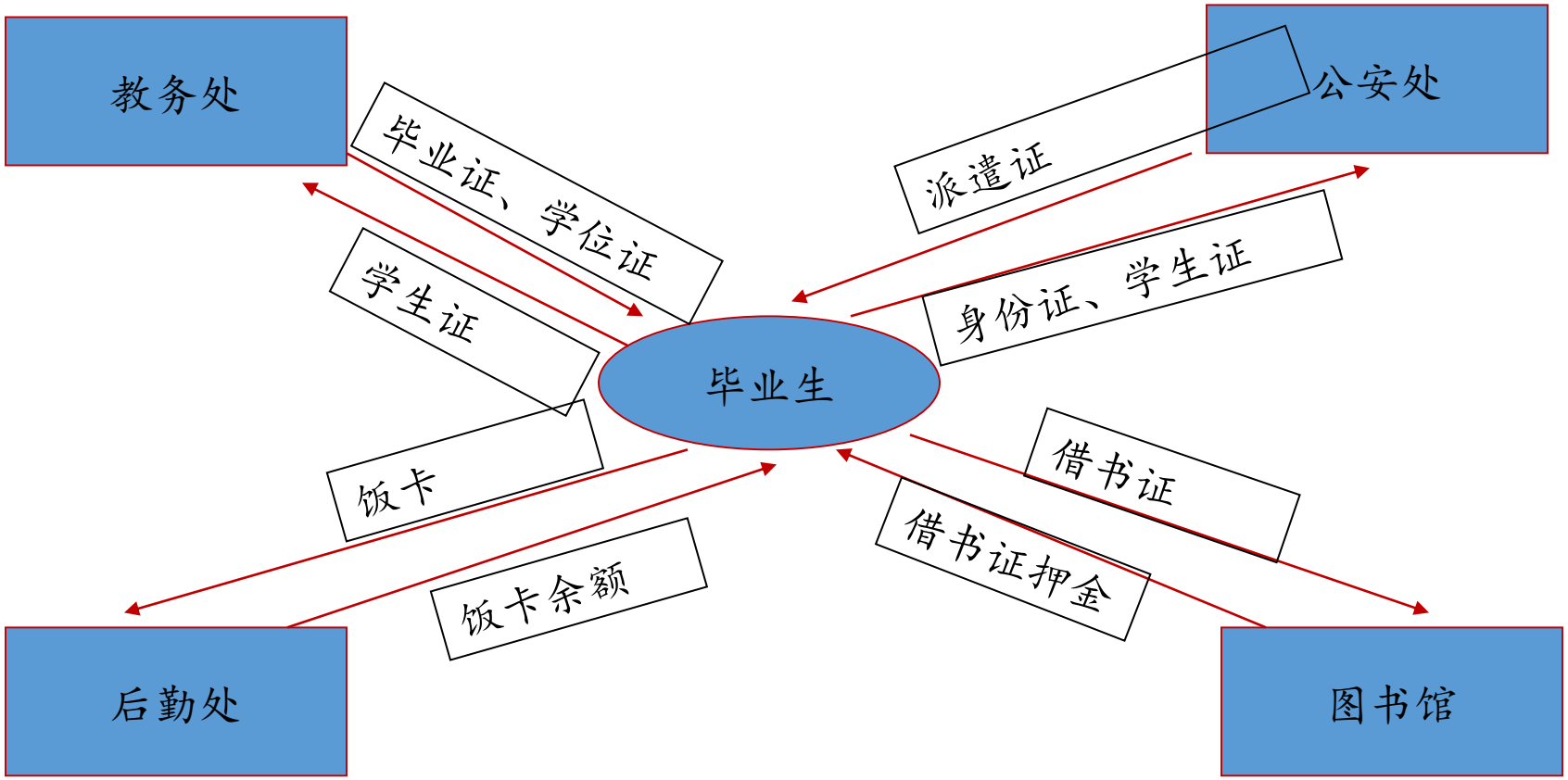
装饰器模式小结

1 装饰模式用于动态地给一个对象增加一些额外的职责，就增加对象功能来说，装饰模式比生成子类实现更为灵活。它是一种对象结构型模式。
2 装饰模式包含四个角色：抽象构件定义了对象的接口，可以给这些对象动态增加职责（方法）；具体构件定义了具体的构件对象，实现了在抽象构件中声明的方法，装饰器可以给它增加额外的职责（方法）；抽象装饰类是抽象构件类的子类，用于给具体构件增加职责，但是具体职责在其子类中实现；具体装饰类是抽象装饰类的子类，负责向构件添加新的职责。
3 使用装饰模式来实现扩展比继承更加灵活，它以对客户透明的方式动态地给一个对象附加更多的责任。装饰模式可以在不需要创造更多子类的情况下，将对象的功能加以扩展。
4 装饰模式的主要优点在于可以提供比继承更多的灵活性，可以通过一种动态的方式来扩展一个对象的功能，并通过使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同行为的组合，而且具体构件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类；其主要缺点在于使用装饰模式进行系统设计时将产生很多小对象，而且装饰模式比继承更加易于出错，排错也很困难，对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为烦琐。
5 装饰模式适用情况包括：在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责；需要动态地给一个对象增加功能，这些功能也可以动态地被撤销；当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展和维护时。
6 装饰模式可分为透明装饰模式和半透明装饰模式：在透明装饰模式中，要求客户端完全针对抽象编程，装饰模式的透明性要求客户端程序不应该声明具体构件类型和具体装饰类型，而应该全部声明为抽象构件类型；半透明装饰模式允许用户在客户端声明具体装饰者类型的对象，调用在具体装饰者中新增的方法。

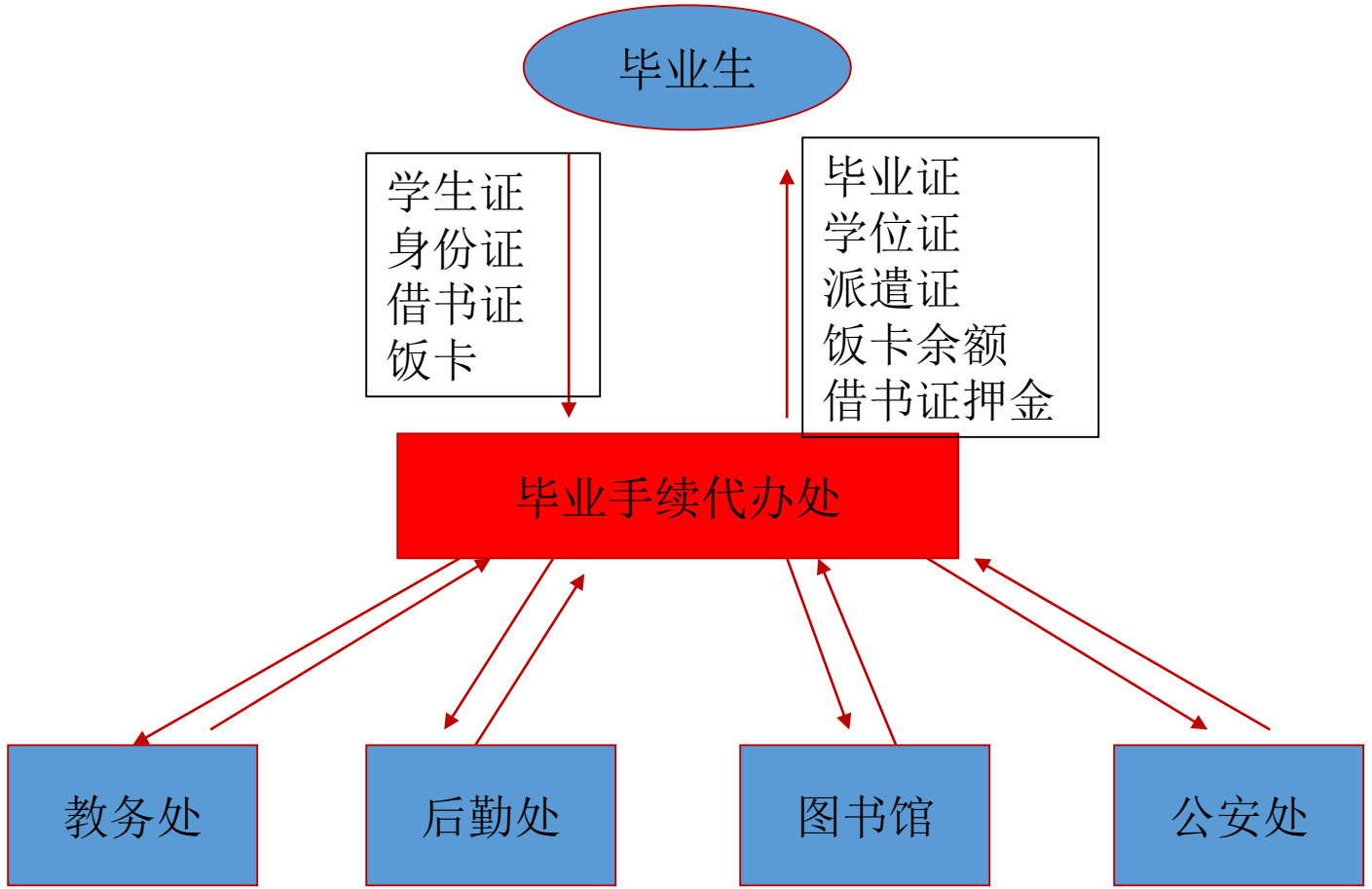
软件工程学科导论-软件设计模式

01 软件设计原则
02 软件设计模式
03 设计模式之适配器模式
04 设计模式之装饰器模式
05 设计模式之外观模式
06 设计模式之观察者模式

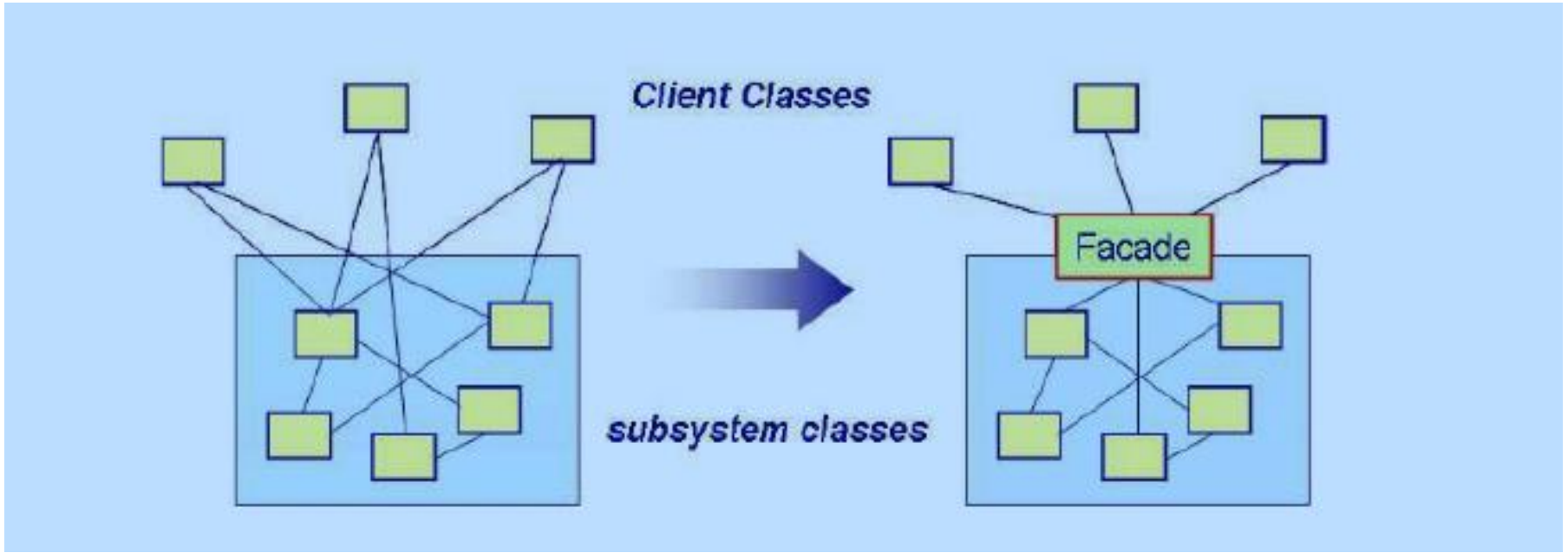
05 软件设计模式-外观模式



05 软件设计模式-外观模式



05 软件设计模式-外观模式



Facade（外观）模式定义

外观模式，要求外部与一个子系统的通信必须通过一个统一的外观对象进行，为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。外观模式属于对象结构型模式。

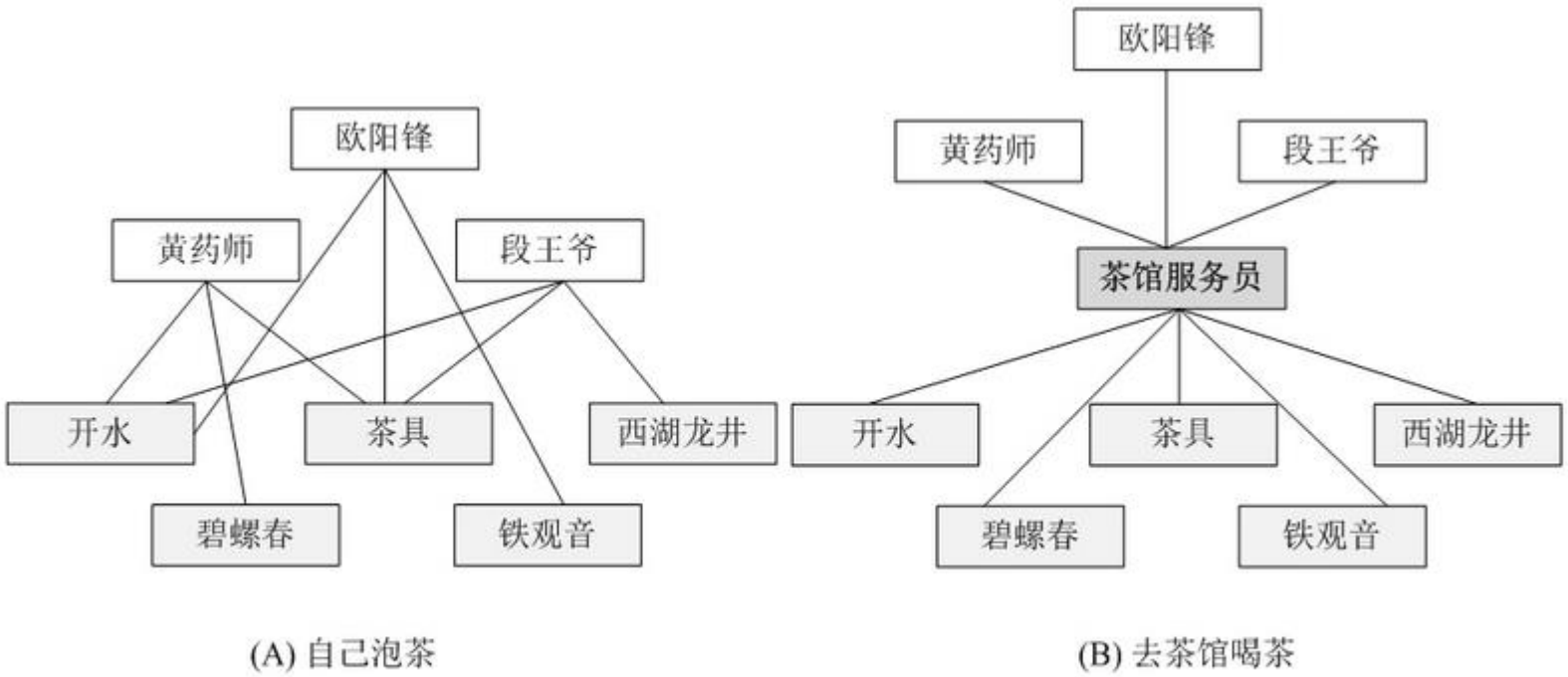


图1 两种喝茶方式示意图

Facade（外观）模式的结构

- Facade表示外观角色，客户端可以调用这个角色的方法，此角色知道相关的子系统的功能和责任，正常情况下，将客户端发来的请求交付给相应的子系统去完成。

- 子系统并不知道外观类的存在，子系统仅仅把它当做另一个客户端。

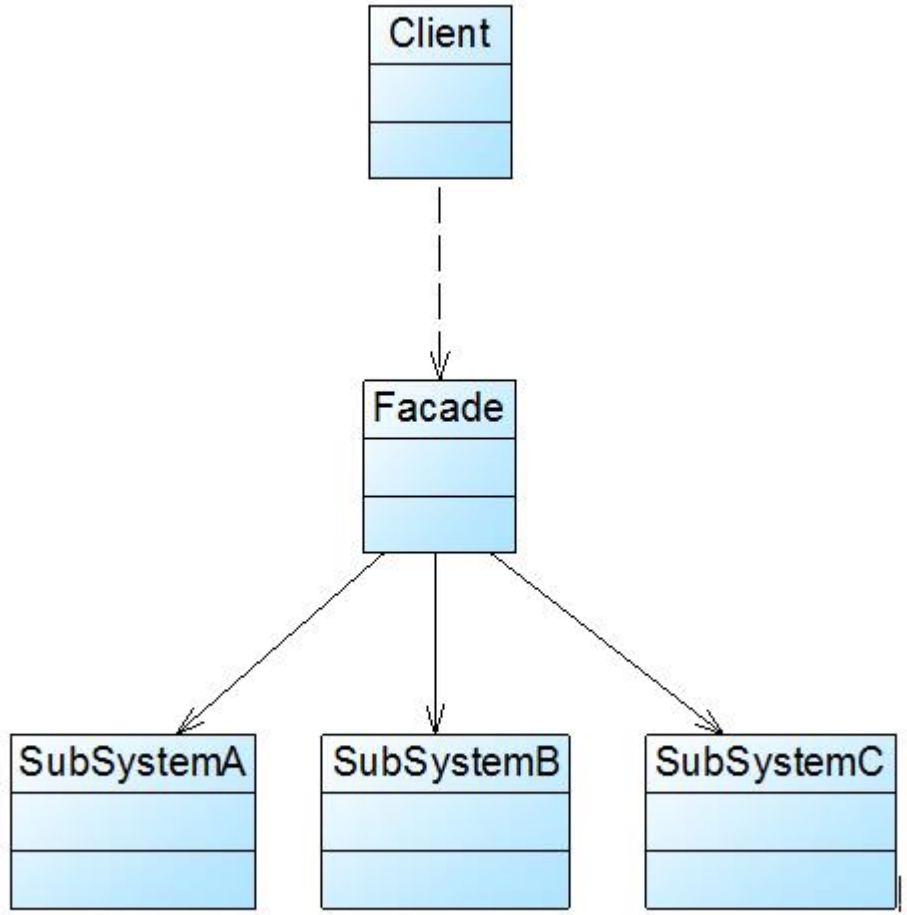


图3 外观模式结构图

Facade（外观）模式的结构

外观模式包括如下角色：

- **Facade**　知道哪些子系统的类能完成用户请求的功能。可被客户调用。
  将客户的请求委派给适当的子系统对象来完成。

- **Subsystem**
  每个子系统都是多个类的集合。可被客户或Facade调用。
  处理由Facade对象或客户对象指派的任务。
  不知道Facade对象或客户对象的存在。

Facade（外观）模式适用场景

（1）当要为访问一系列复杂的子系统提供一个简单入口时可以使用外观模式。

（2）客户端程序与多个子系统之间存在很大的依赖性。引入外观类可以将子系统与客户端解耦，从而提高子系统的独立性和可移植性。

（3）在层次化结构中，可以使用外观模式定义系统中每一层的入口，层与层之间不直接产生联系，而通过外观类建立联系，降低层之间的耦合度

Facade（外观）模式具体应用

- 绝大多数系统都有一个首页或者导航页面，同时也提供了菜单或者工具栏，在这里，首页和导航页面、菜单和工具栏就是外观角色，通过它们用户可以快速访问子系统，降低了系统的复杂程度。

Facade（外观）模式具体应用

外观模式在其他的领域也有许多应用：

- 文件安全传送模块开发：存在三个模块，读取文件内容模块，对文件内容进行加密模块和写入文件模块。提供一个外观，对这些模块进行处理。

- 生活中的外观：找电脑组装公司组装电脑； 通过中介来处理房屋的买卖；开学需要办理各种手续流程，可以找人代为处理；公司需要开发某款产品，市场部门负责市场调研、采购部门负责采购配件，开发部门负责开发，这个时候，由总监代为处理各个部门的活动。

Facade（外观）模式实例

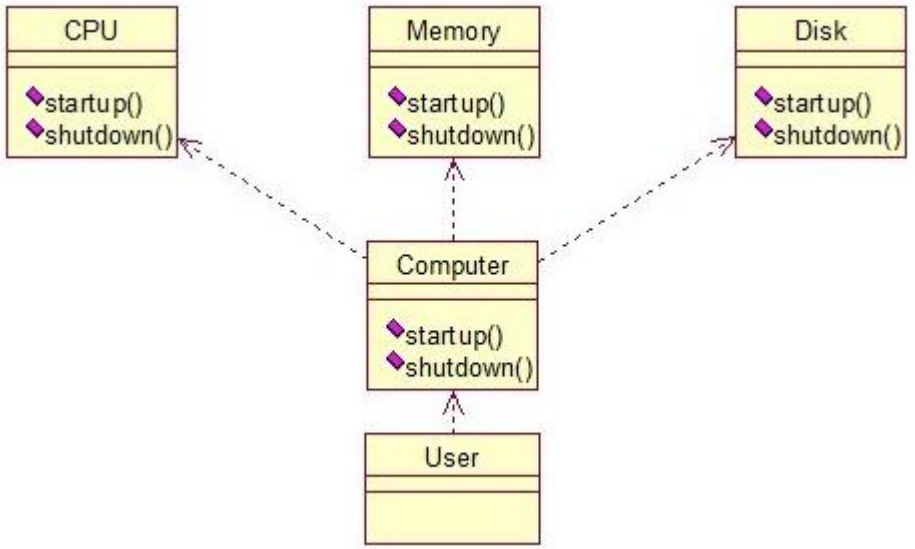
- 我有一个专业的相机，我就喜欢自己手动调光圈、快门，这样照出来的照片才专业，但MM可不懂这些，教了半天也不会。幸好相机有Facade设计模式，把相机调整到自动档，只要对准目标按快门就行了，一切由相机自动调整，这样MM也可以用这个相机给我拍张照片了。

- 在这里相机就是外观，其他的调光圈、快门等操作就是子系统。

Facade（外观）模式实例-电脑开机

我们以一个计算机的启动过程为例

计算机启动，需要开启CPU，内存和硬盘，这三个操作相对独立，我们可以通过点击开机按钮，直接让电脑自动启动CPU，内存和硬盘。电脑就充当外观类，开启CPU、内存和硬盘的操作就是各个子系统。



Facade（外观）模式优缺点

优点：

1）对客户屏蔽子系统组件，减少了客户处理的对象数目并使得子系统使用起来更加容易。通过引入外观模式，客户代码将变得很简单，与之关联的对象也很少。

2）实现了子系统与客户之间的松耦合关系，这使得子系统的组件变化不会影响到调用它的客户类，只需要调整外观类即可。

3）降低了大型软件系统中的编译依赖性，并简化了系统在不同平台之间的移植过程，因为编译一个子系统一般不需要编译所有其他的子系统。一个子系统的修改对其他子系统没有任何影响，而且子系统内部变化也不会影响到外观对象。

4）只是提供了一个访问子系统的统一入口，并不影响用户直接使用子系统类。

Facade（外观）模式优缺点

缺点：

1） 不能很好地限制客户使用子系统类，如果对客户访问子系统类做太多的限制则减少了可变性和灵活性。

2） 在不引入抽象外观类的情况下，增加新的子系统可能需要修改外观类或客户端的源代码，违背了“开闭原则”（对扩展开放，对修改关闭）。

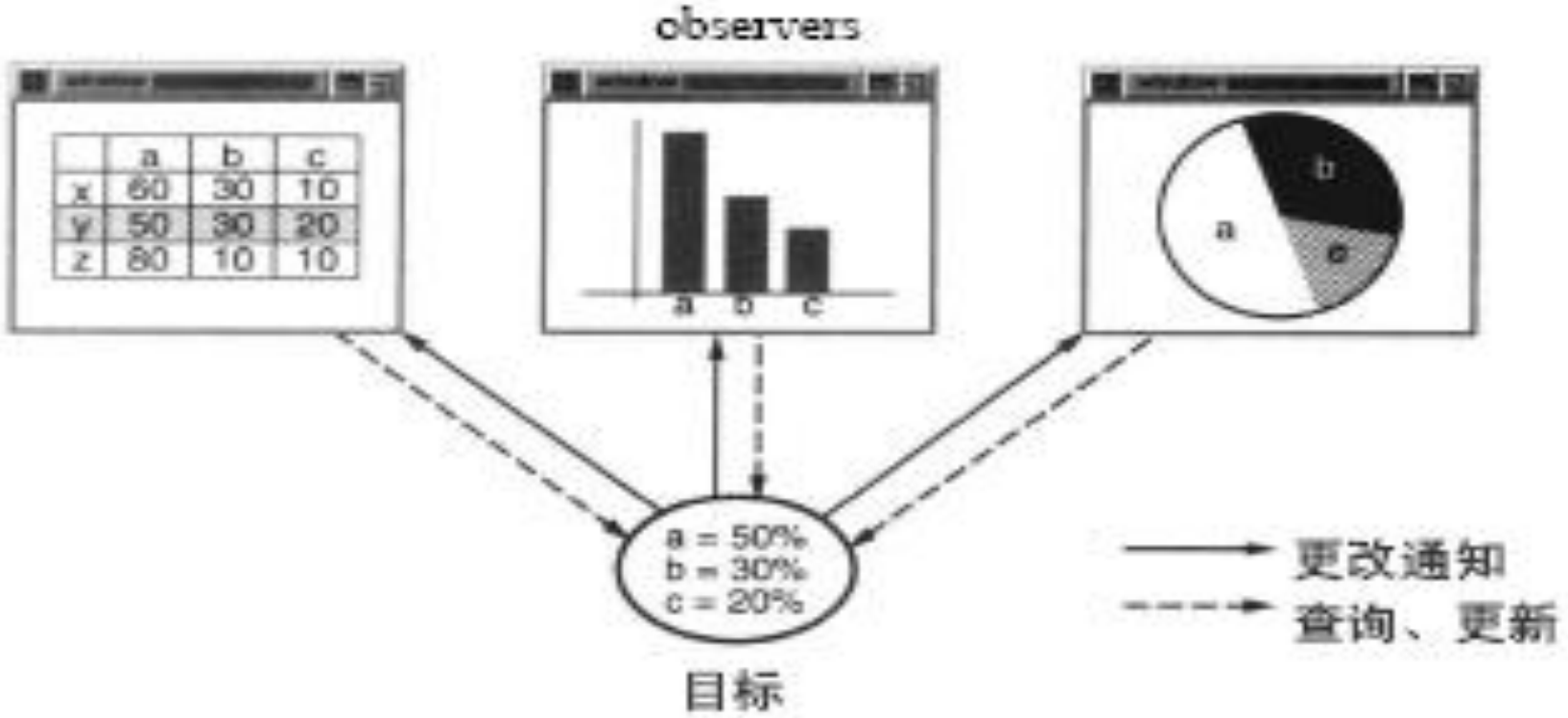
软件工程学科导论-软件设计模式

01 软件设计原则
02 软件设计模式
03 设计模式之适配器模式
04 设计模式之装饰器模式
05 设计模式之外观模式
06 设计模式之观察者模式

观察者模式的由来

- 将系统分割成一系列相互协作的类有也存在一些不足：
  - 需要维护相关对象间的一致性
  - 为维持一致性而使各类紧密耦合，可能降低其可重用性
- 没有理由限定依赖于某数据对象的对象数目， 对相同的数据对象可以有任意数目的不同用户界面

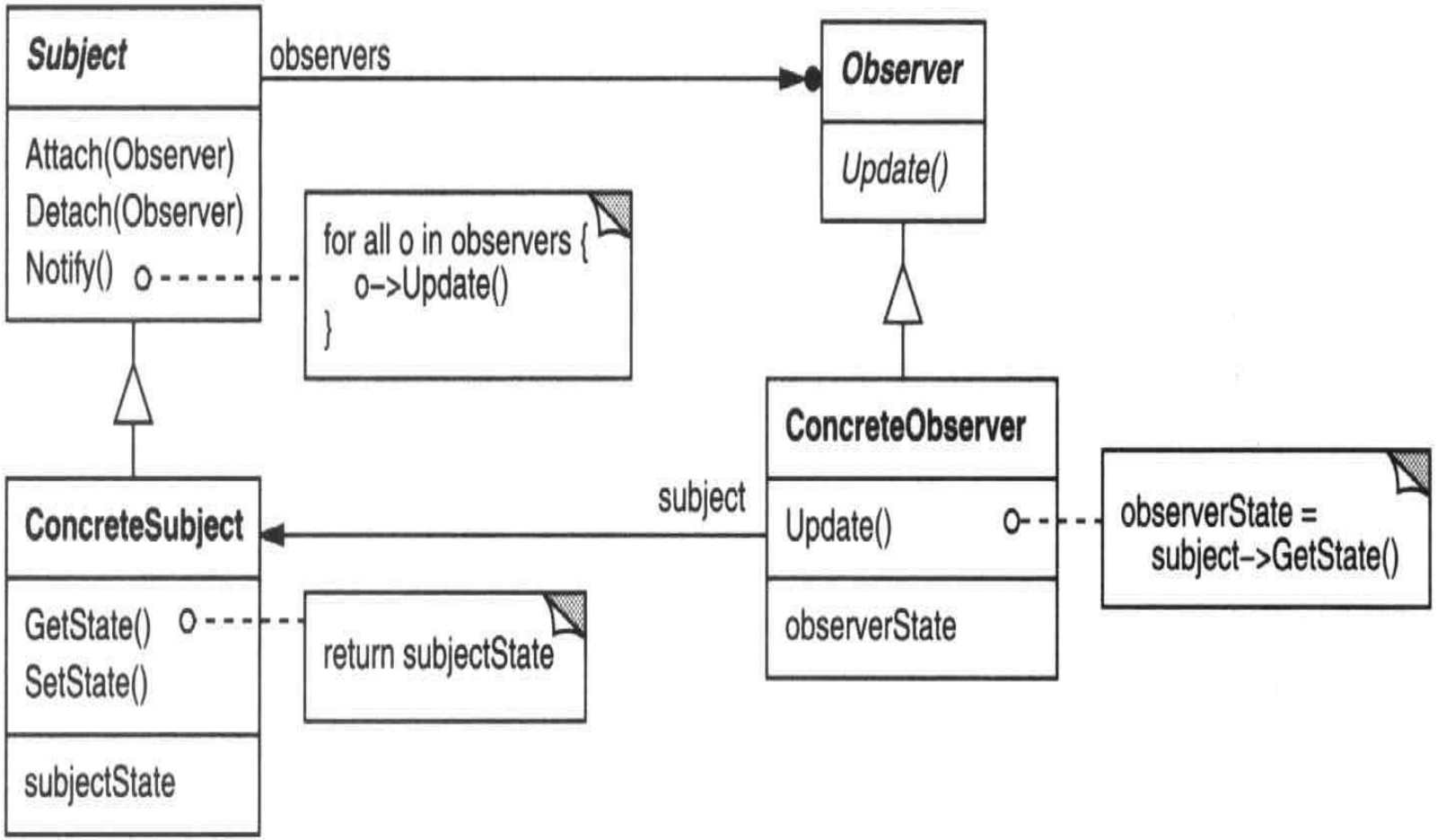
观察者模式的由来


observers
a b c
x 60 30 10
y 50 30 20
z 80 10 10
a b c
a
b
c
a = 50%
b = 30%
c = 20%
更改通知
查询、更新
目标

观察者模式的意图和适用性

- **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
- **适用场合**
  - 当一个抽象模型有两个方面，一方面依赖于另一方面。将二者封装在独立对象中以使它们可以独立被改变和复用
  - 一个对象的改变需要同时改变其它对象，但不知道具体有多少对象有待改变
  - 当一个对象必须通知其它对象，而它又不能假定其它对象是谁。换言之，不希望这些对象是紧密耦合的

观察者模式的结构


Subject
observers
Observer
Attach(Observer)
Detach(Observer)
Notify()
Update()
for all o in observers {
o->Update()
}
ConcreteObserver
subject
ConcreteSubject
Update()
observerState =
subject->GetState()
GetState()
SetState()
observerState
return subjectState
subjectState

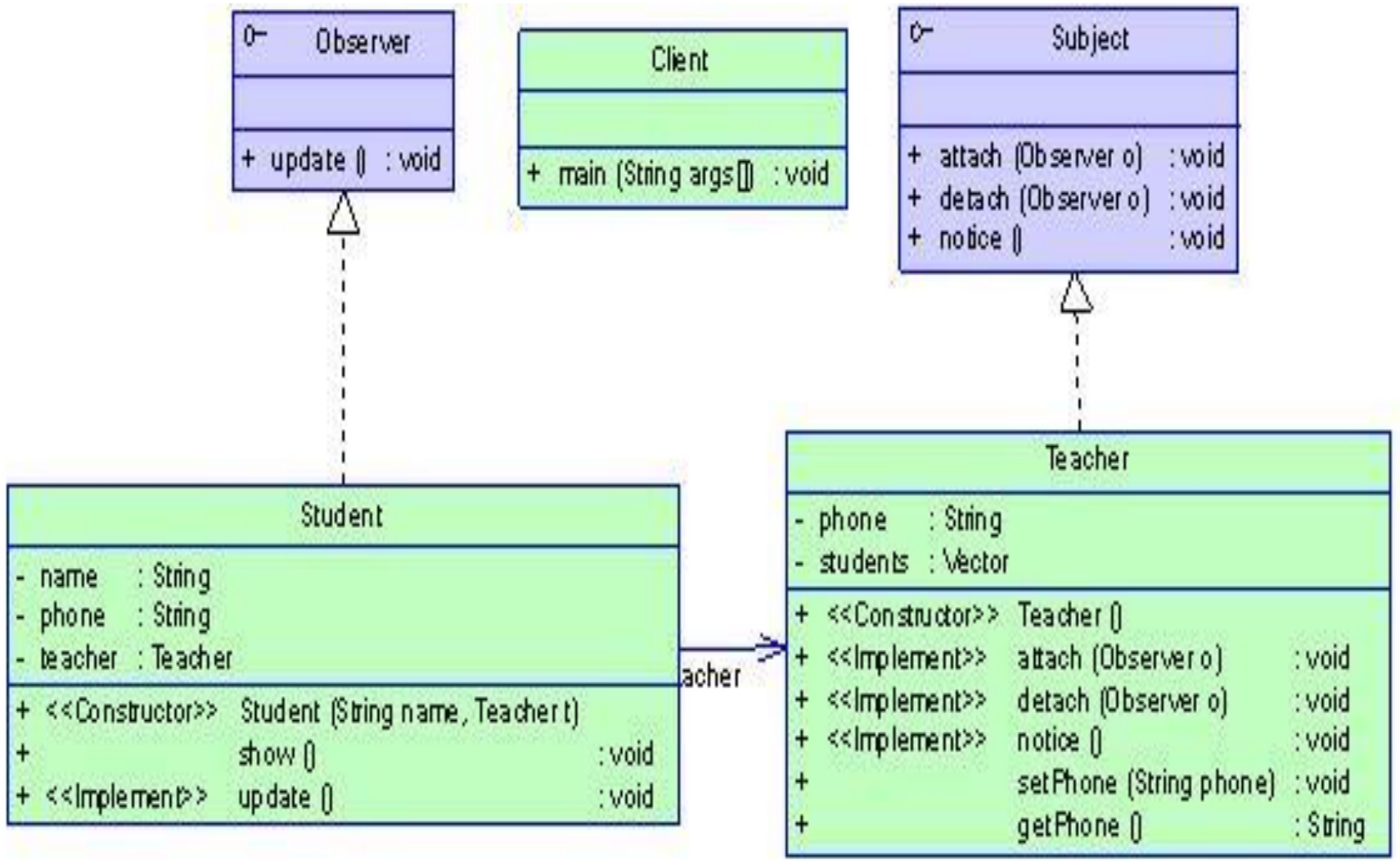
观察者模式的参与者

- Subject：抽象的主题，即被观察的对象
- Observer：抽象的观察者
- Concrete Subject：具体被观察对象
- Concrete Observer：具体的观察者
- 注意：在观察者模式中，Subject通过Attach和Detach方法添加或删除所关联的观察者，并通过Notify进行更新，让每个观察者观察到最新的状态

观察者模式的应用示例

- 班主任老师有电话号码，学生需要知道班主任老师的电话号码以便于在合适时的时候拨打
- 在这样的组合中，老师是一个**被观察者（Subject）**，学生就是需要知道信息的观察者
- 当老师的电话号码发生改变时，学生得到通知，并更新相应的电话记录

观察者模式的应用示例


Observer
+ update () : void
Client
+ main (String args[]) : void
Subject
+ attach (Observer o) : void
+ detach (Observer o) : void
+ notice () : void
Student
- name : String
- phone : String
- teacher : Teacher
+ <<Constructor>> Student (String name, Teacher t)
+ show () : void
+ <<Implement>> update () : void
acher
Teacher
- phone : String
- students : Vector
+ <<Constructor>> Teacher ()
+ <<Implement>> attach (Observer o) : void
+ <<Implement>> detach (Observer o) : void
+ <<Implement>> notice () : void
+ setPhone (String phone) : void
+ getPhone () : String

观察者模式分析

- 应用场景
  - 对一个对象状态的更新，需要其他对象同步更新，而且其他对象的数量动态可变
  - 对象仅需要将自己的更新通知给其他对象而不需要知道其他对象细节
- 优点
  - Subject和Observer之间是松偶合的，可以各自独立改变
  - Subject在发送广播通知时，无须指定具体的Observer，Observer可以自己决定是否要订阅Subject的通知
  - 高内聚、低偶合
- 缺陷
  - 松偶合导致代码关系不明显，有时可能难以理解
  - 如果一个Subject被大量Observer订阅的话，在广播通知的时候可能会有效率问题

哈尔滨工程大学
Harbin Engineering University

王念滨
计算机科学与技术学院
大数据与智能计算课题组
2019年3月