



# SOFT130006

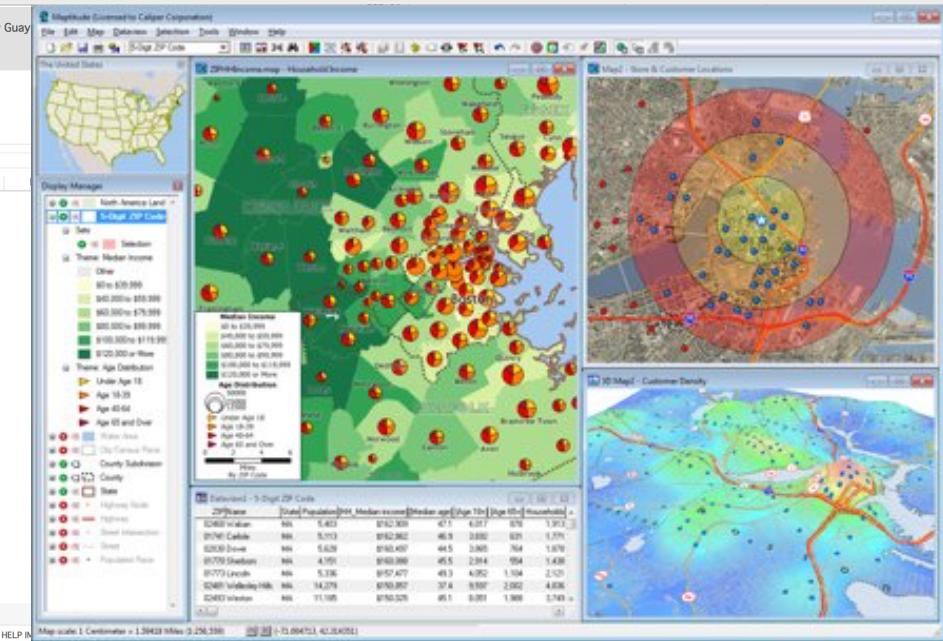
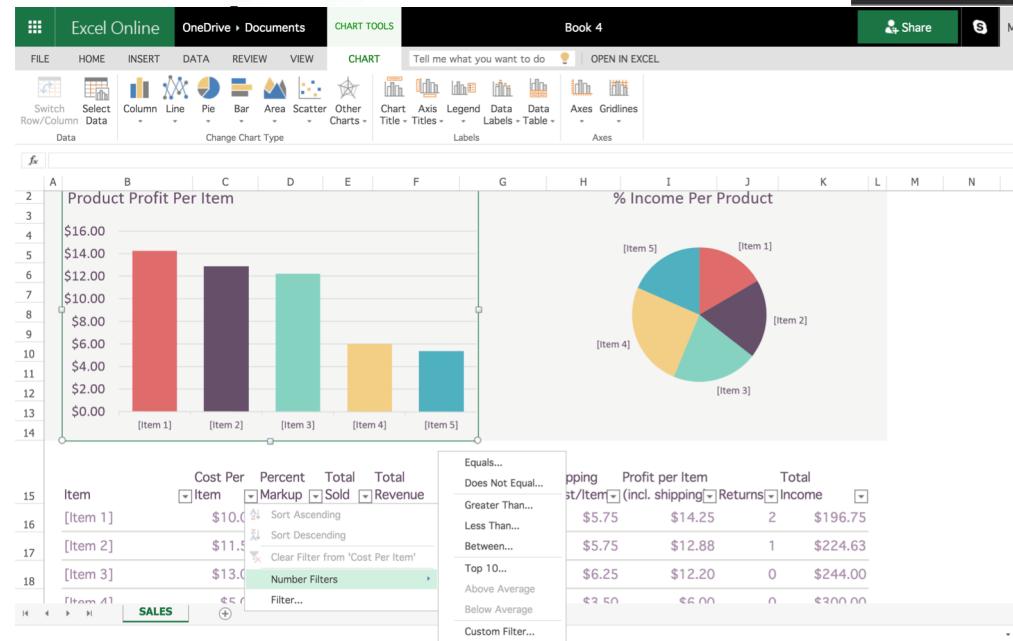
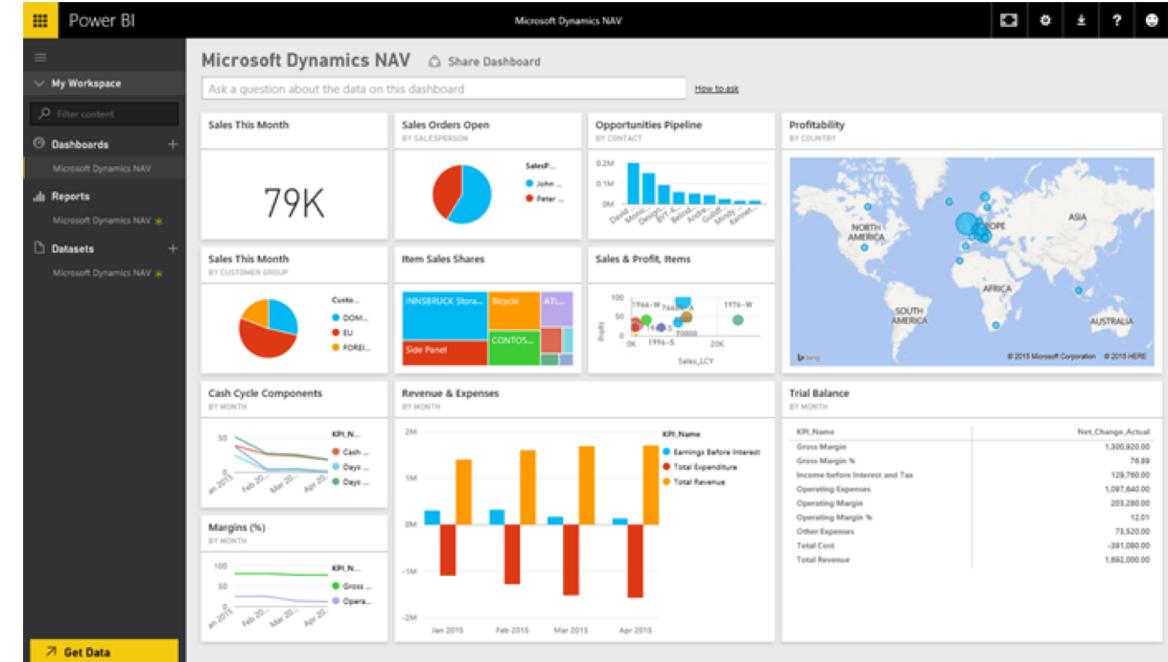
# 软件工程

## 10. 软件设计 —设计模式

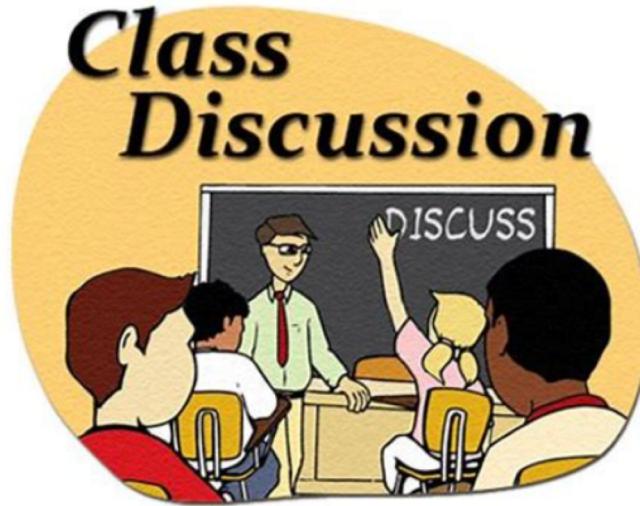


复旦大学软件学院  
彭鑫

# 软件应用实例



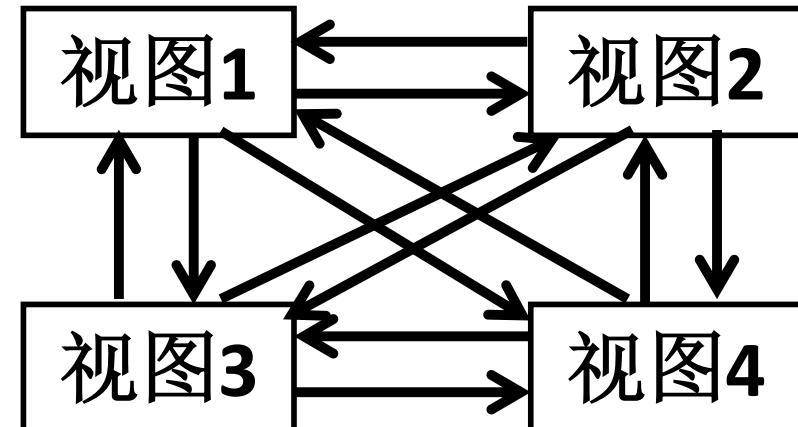
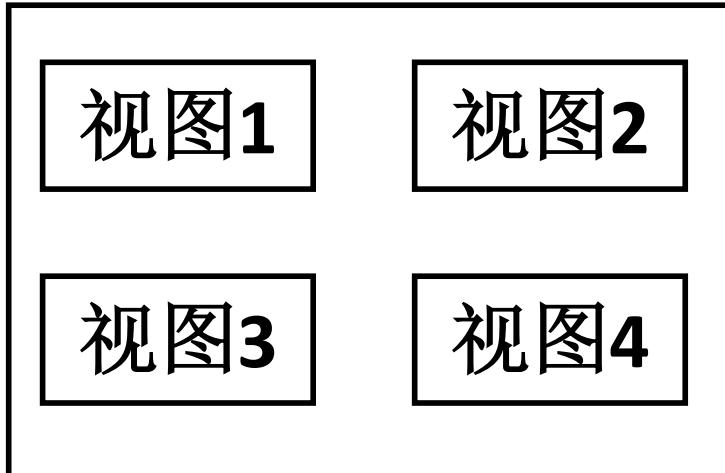
# 课堂讨论：软件设计中的共性



?

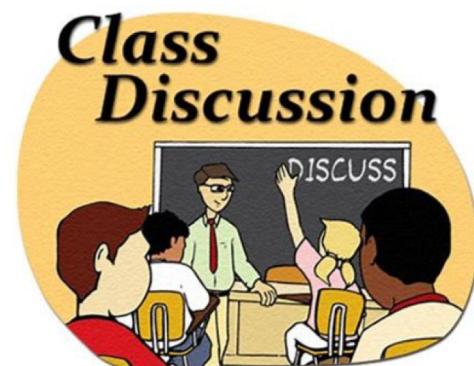
课堂讨论：这几个软件应用有什么共同之处？在设计方案上有什么通用的地方？

# 一个真实的设计方案...

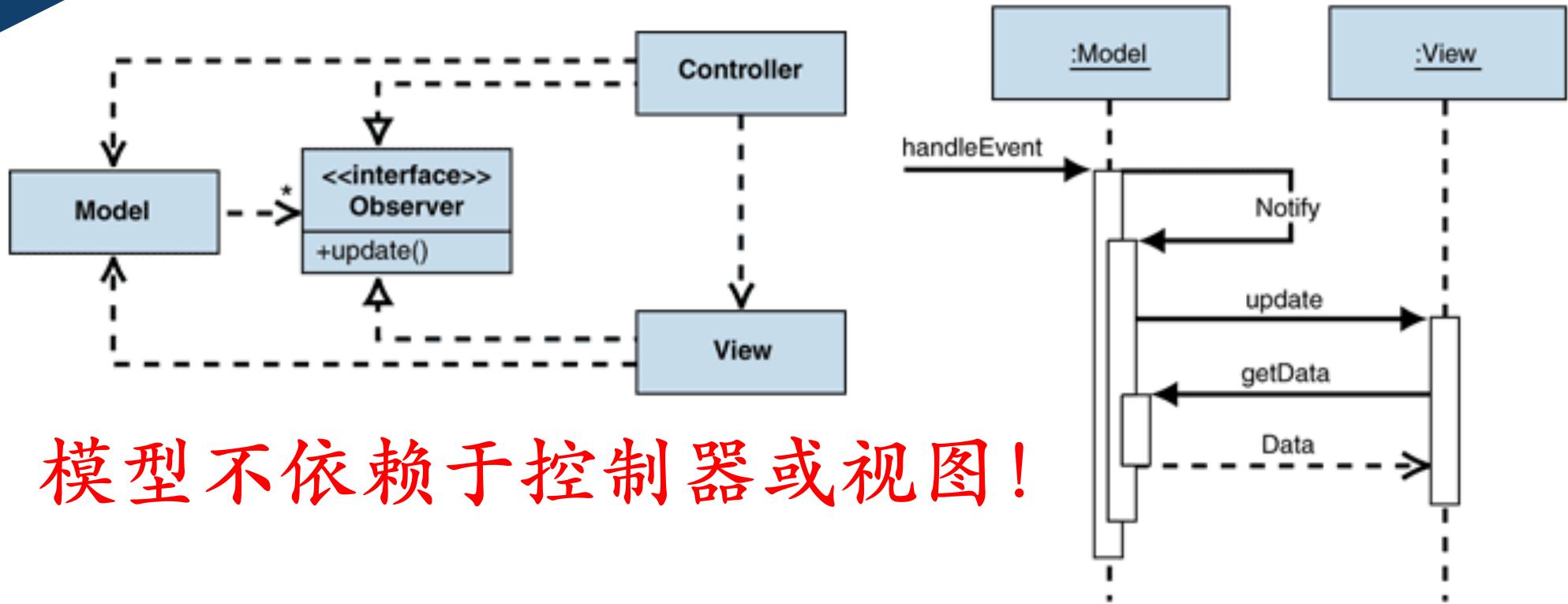


界面由多个可以浏览和修改数据的视图类构成。  
每个视图类都依赖于其他所有视图类，在发生  
数据变化时通知更新。这个设计有什么问题？

?



# MVC(Model-View-Controller)模式



模型不依赖于控制器或视图！

模型、视图、控制器三者角色分离

- 模型保存需要展示和修改的数据，同时处理数据更新事件，数据发生修改后通知观察者（控制器和视图）
- 视图之间相互独立、不发生关联，收到更新通知后从模型那里读取数据进行视图更新
- 控制器根据模型数据的变化对视图进行控制（如设置显示属性）

# 启示

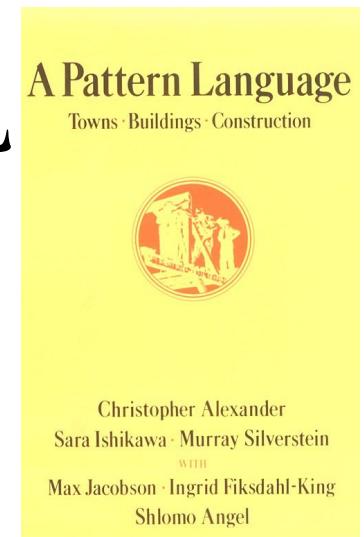
- 软件应用中存在很多共性设计问题
- 这些设计问题的具体形式有所差异，但可抽象为共性问题
- 共性设计问题存在良好的参考设计方案：易理解、易修改、易扩展
  - ✓ 优化职责分配：例如MVC模式中，视图不再具备数据更新或控制职责，更加符合单一职责的原则
  - ✓ 关键是解耦：例如MVC模式中，视图之间解耦、模型不依赖于控制器或视图
  - ✓ 重要手段是引入抽象：例如MVC模式中，观察者接口的引入隔离了模型对控制器和视图的依赖

# 设计模式 (Design Pattern)

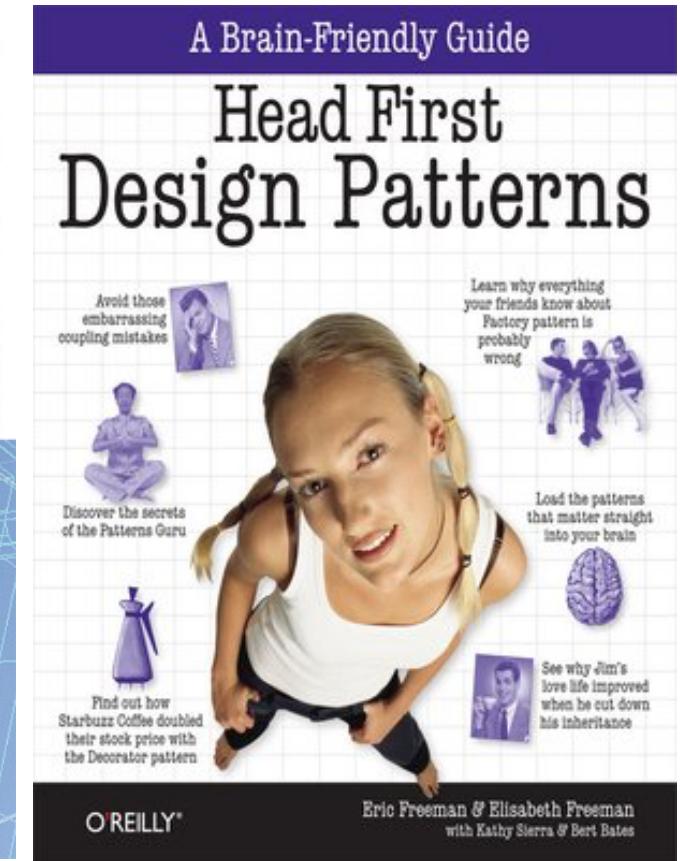
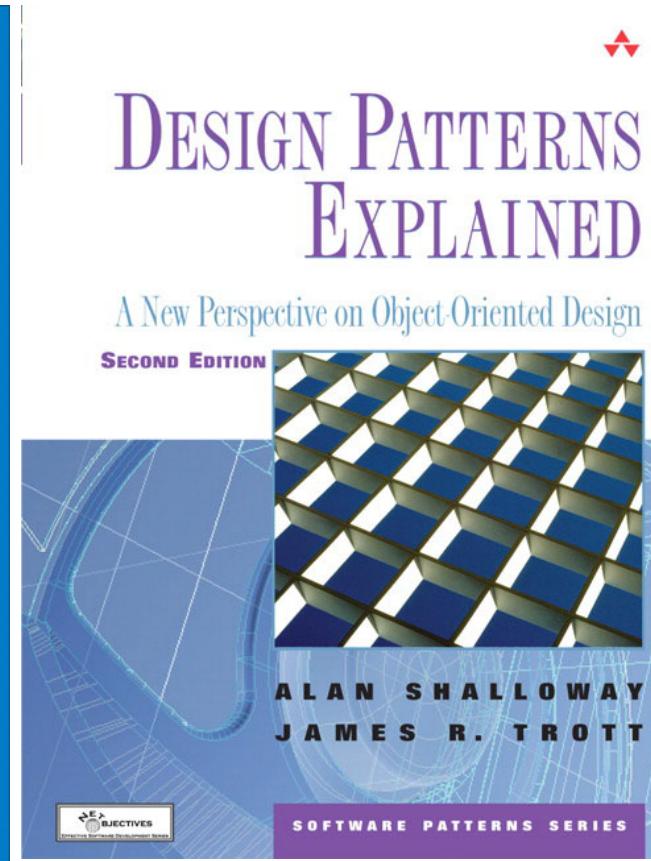
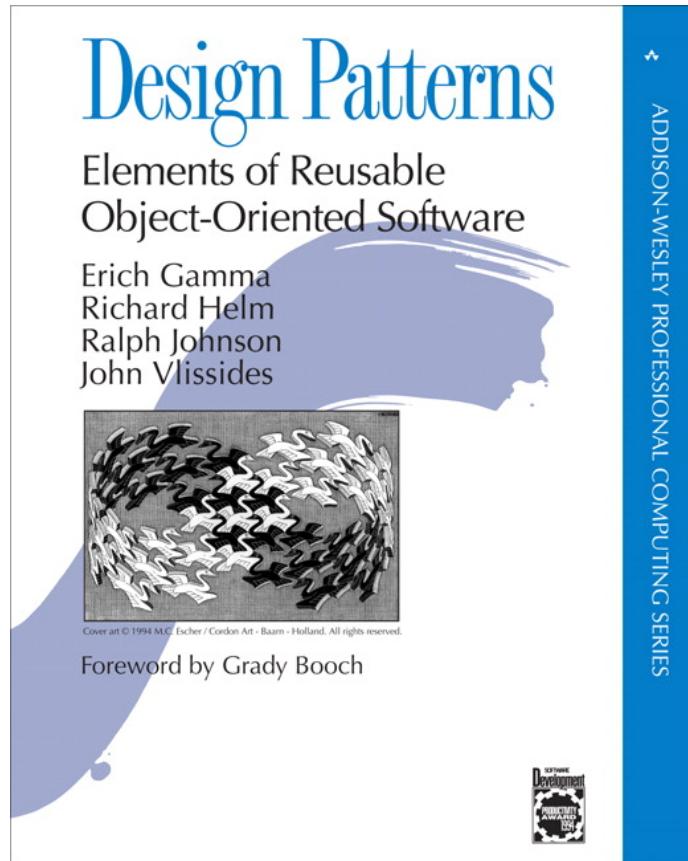
- 针对共性设计问题的一种参考解决方案
- 是对一系列相似设计方案的本质和共性抽象，因此具有通用性
  - ✓ 针对一类设计问题，根据具体问题上下文决定其适用性
  - ✓ 提供的方案经过大量验证，具备良好的性质（如可扩展性）
  - ✓ 明确了构成解决方案的抽象角色、职责分配以及相互之间的关系，例如模型、视图、控制器
  - ✓ 具体应用时需要结合特定问题上下文进行实例化

# 设计模式的起源和意义

- 从Christopher Alexander关于建筑设计模式的思想中派生而来
- Gamma等人（被称为四人组）的著作使得面向对象设计模式正式进入人们的视野
  - ✓ Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA.: Addison-Wesley
- 设计模式的意义
  - ✓ 经过验证的共性设计问题解决方案
  - ✓ 谈论设计的通用词汇表



# 设计模式参考资料



JHotDraw (<http://www.jhotdraw.org>) : 一个包含丰富的设计模式实例的开源Java GUI框架

Hillside小组 ([hillside.net/patterns](http://hillside.net/patterns)) : 设计模式资源汇编

# 设计模式四要素 (Gamma等人)

- 有意义的模式名字
- 解释模式何时适用的问题描述
- 解决方案描述
  - ✓ 包括组成方案的各个部分（角色）、各自的职责、相互之间的关系
  - ✓ 并不是具体的方案描述，而是一个抽象的解决方案模板，可以针对不同的具体情况进行实例化
  - ✓ 通常可以通过图形化的方式表达，展示方案中对象和对象类之间的关系
- 应用该模式的效果以及其中所涉及的权衡决策

# 问题与解决方案描述的细化

## • 问题描述

- ✓ 动机：对于模式为什么有用的一种描述
- ✓ 适用性：对于该模式可以使用的情形的描述

## • 解决方案描述

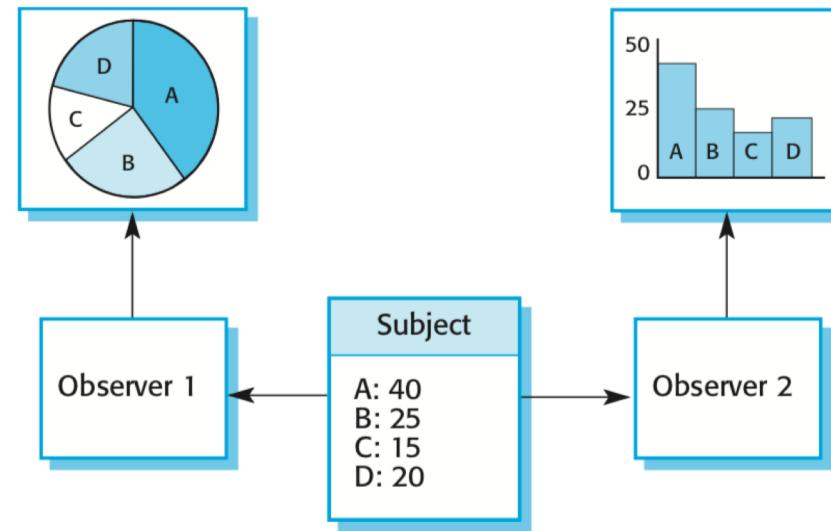
- ✓ 模式结构
- ✓ 参与者
- ✓ 协作
- ✓ 实现

# 观察者模式 (Observer Pattern)

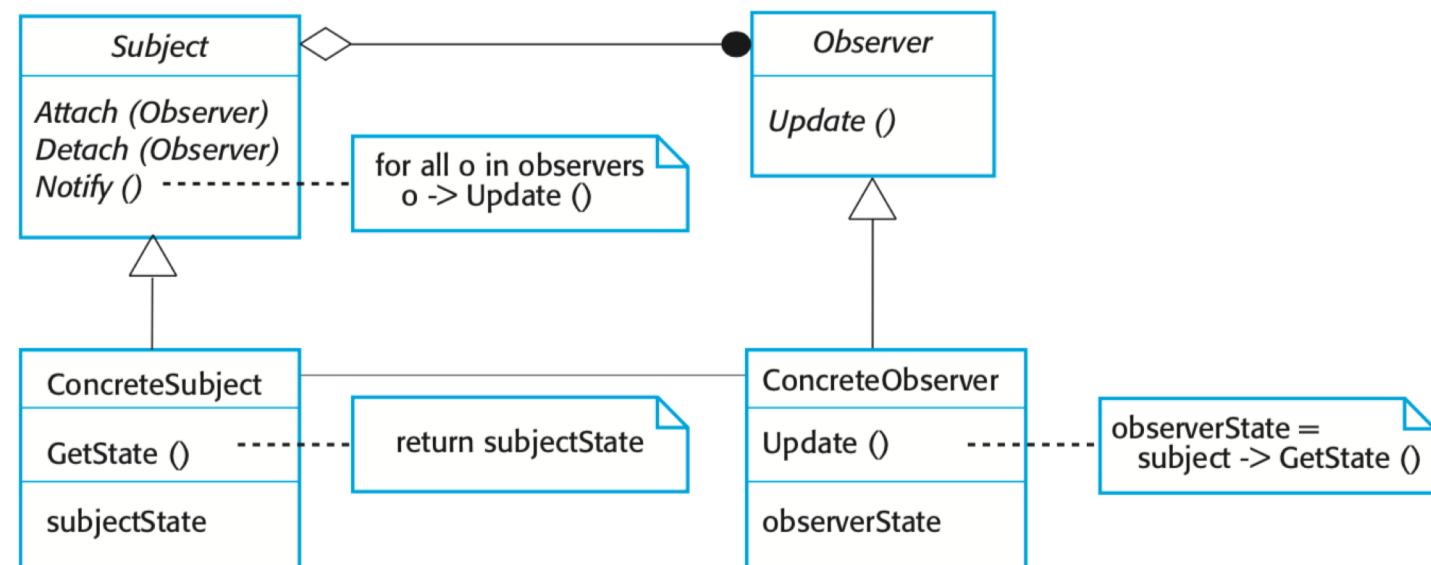
- 向多个其他对象（观察者）告知某个对象（主题，即被观察者）的状态发生了变化
- 观察者（负责显示逻辑）与被观察者（负责数据管理）职责相分离（单一职责原则）
- 被观察者与观察者解耦，观察者可灵活增加（开闭原则）

# 观察者模式

## 实现效果



## 设计结构



《软件工程》7.2节

## 观察者模式的规范描述-1

- 模式名称：观察者
- 描述：将一个对象的状态的呈现与对象本身相分离，允许为对象提供多种呈现方式。当对象状态变化时，所有的呈现都会自动得到通知并进行更新以反映变化。

详见《软件工程》7.2节

## 观察者模式的规范描述-2

- **问题描述：**在很多情况下你都必须为状态信息提供多种呈现方式，例如一个图形化呈现和一个表格呈现。这些呈现方式在对信息进行刻画和定义时并不都是已知的。所有不同的呈现方式都应该支持交互，在状态发生变化时所有呈现都必须更新。这个模式的使用情形是：**状态信息需要不止一种呈现方式，并且维护状态信息的对象不需要知道所使用的特定的呈现格式。**

# 观察者模式的规范描述-3

- **解决方案描述：**解决方案包括两个抽象对象Subject（主题，即被观察者）和Observer（观察者），以及两个继承了相关抽象对象属性的具体对象ConcreteSubject（具体主题）和ConcreteObserver（具体观察者）。抽象对象包括适用于所有情形的通用操作。需要呈现的状态在ConcreteSubject中进行维护，该对象继承了Subject的操作从而允许该对象增加和移除观察者（每个观察者对应一种呈现方式）以及在状态发生变化时通知观察者。ConcreteObserver维护了ConcreteSubject状态的一份拷贝，并且实现了Observer的Update()接口，该接口允许这些拷贝能够被同步更新。ConcreteObserver自动呈现状态并在任何时候当状态发生更新时自动反映变化。该模式的UML模型如图7.12所示。

## 观察者模式的规范描述-4

- **效果：**主题对象只知道抽象的观察者对象，而不知道具体类的细节。因此，这些**对象之间的耦合被最小化了（优点）**。由于缺少这些细节，可能无法对呈现性能进行优化。主题对象的变化可能导致与之相关的针对观察者的更新操作，而其中一些可能并没有必要（缺点）。



# 设计模式背后的原则

- 单一职责原则
- Liskov替换原则 (Liskov Substitution Principle)
- 依赖倒转原则 (Dependence Inversion Principle)
- 接口隔离原则 (Interface Segregation Principle)
- 迪米特 (最少知道) 原则 (Demeter Principle)
- 合成复用原则 (Composite Reuse Principle)

总原则：开闭原则

# 常用设计模式之间的关系

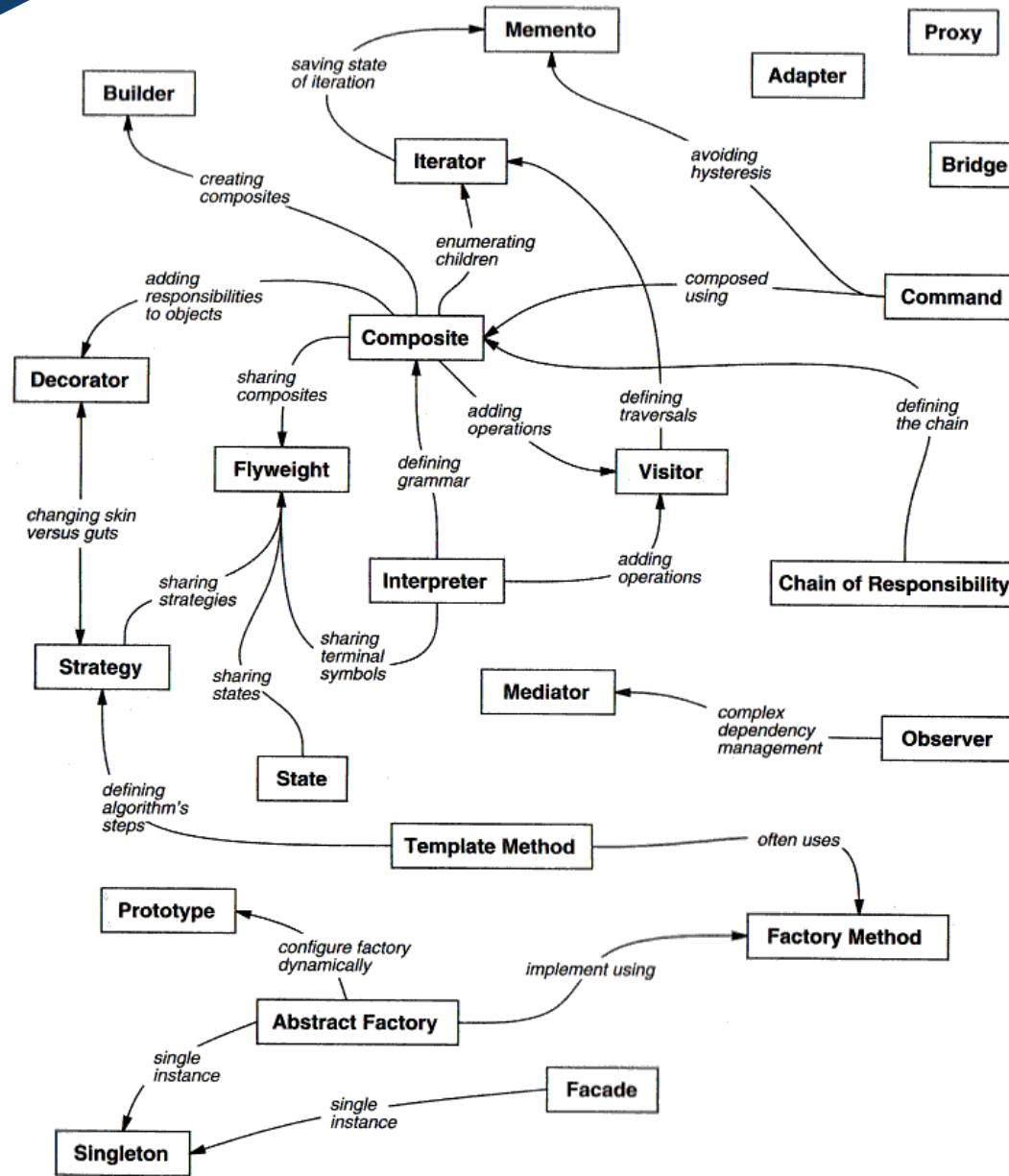
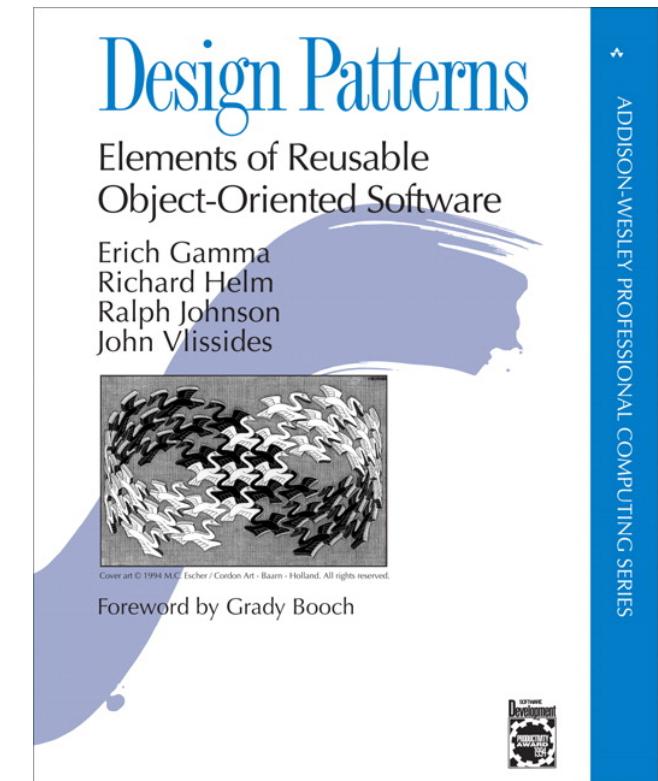


Figure 1.1: Design pattern relationships



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

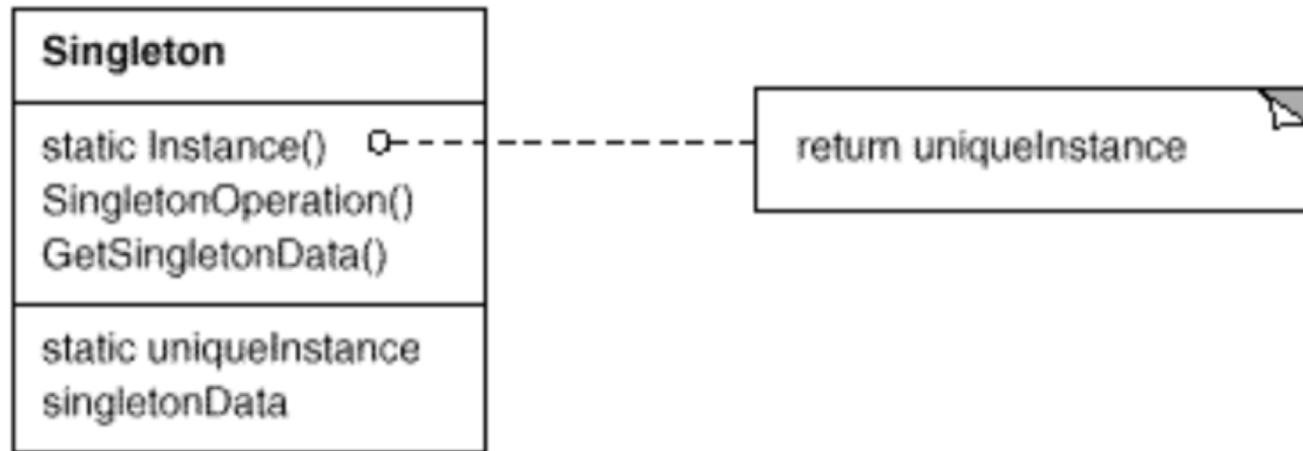
# 常用设计模式列表

- 创建型模式（5种）：工厂方法（Factory Method）、抽象工厂（Abstract Factory）、单例（Singleton）、建造者（Builder）、原型（Prototype）
- 结构型模式（7种）：适配器（Adaptor）、装饰器（Decorator）、代理（Proxy）、外观（Facade）、桥接（Bridge）、组合（Composite）、享元（Flyweight）
- 行为型模式（11种）：策略（Strategy）、模板方法（Template Method）、观察者（Observer）、迭代器（Iterator）、责任链（Chain of Responsibility）、命令（Command）、备忘录（Memento）、状态（State）、访问者（Visitor）、中介者（Mediator）、解释器（Interpreter）

# 常用模式：单例模式 (Singleton)

- 确保一个类只有一个实例并为其提供全局的访问入口
- 保持接口、继承等面向对象特性
- 应用举例
  - ✓ 操作系统的任务管理器和回收站
  - ✓ 网站计数器
  - ✓ 金融交易引擎
  - ✓ 应用程序日志引擎

# 单例模式设计方案



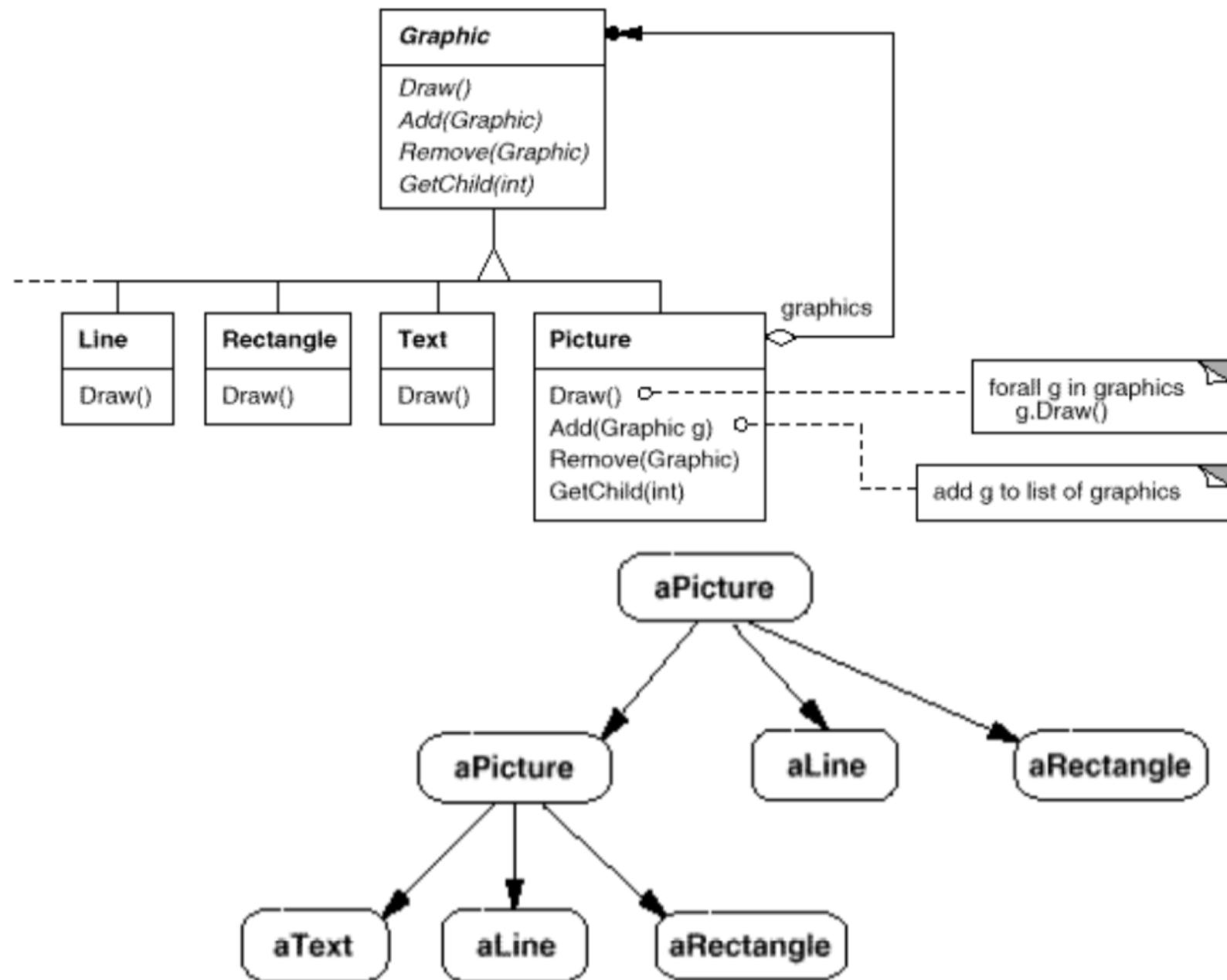
定义一个静态方法为外界返回唯一的类实例

```
1. public class Singleton {  
2.  
3.     /* 私有构造方法, 防止被实例化 */  
4.     private Singleton() {  
5.     }  
6.  
7.     /* 此处使用一个内部类来维护单例 */  
8.     private static class SingletonFactory {  
9.         private static Singleton instance = new Singleton();  
10.    }  
11.  
12.    /* 获取实例 */  
13.    public static Singleton getInstance() {  
14.        return SingletonFactory.instance;  
15.    }  
16.  
17.    /* 如果该对象被用于序列化, 可以保证对象在序列化前后保持一致 */  
18.    public Object readResolve() {  
19.        return getInstance();  
20.    }  
21.}
```

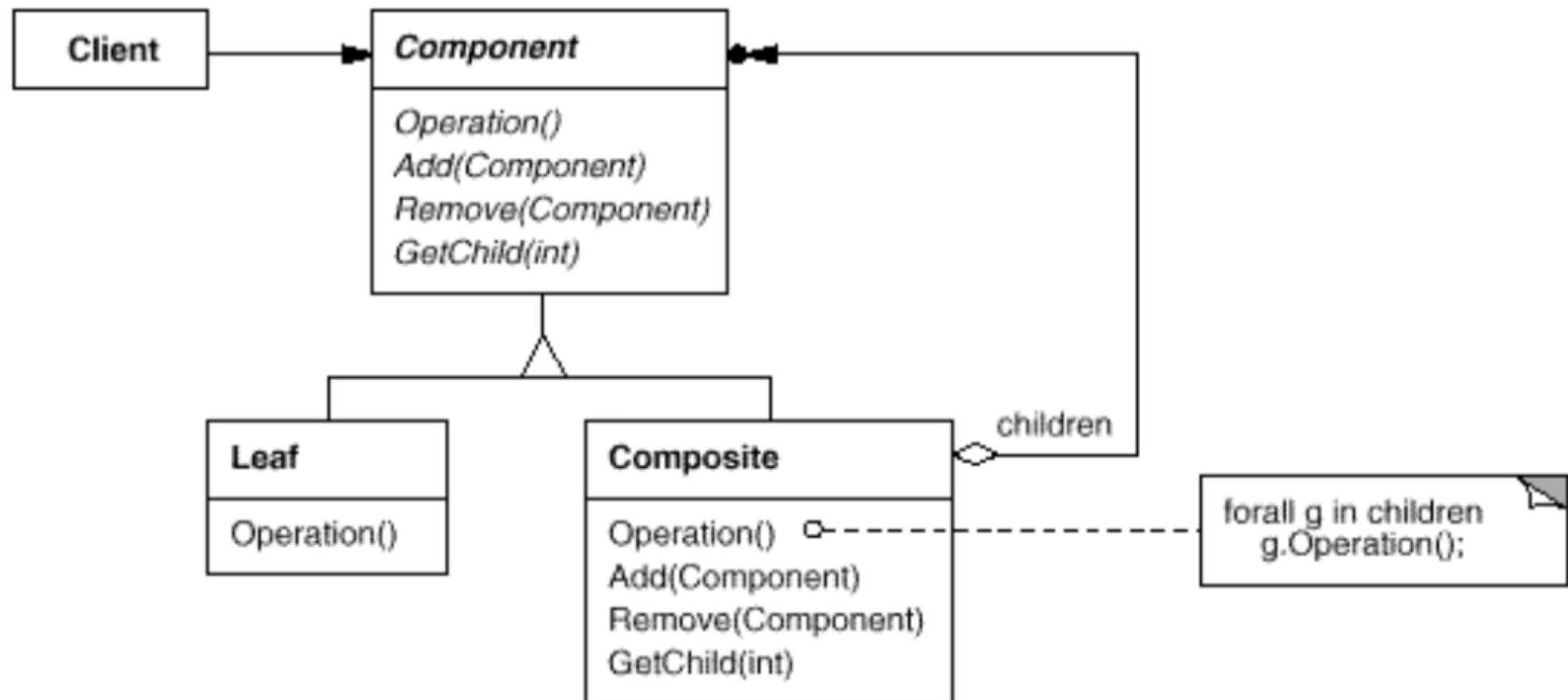
# 常用模式：组合模式 (Composite)

- 将对象组织成表示整体部分层次的树状结构，同时使客户端代码可以统一处理对象以及对象的组合
- 应用举例
  - ✓ 文件系统目录结构
  - ✓ 组织结构
  - ✓ 画图工具中的图形组合

# 组合模式应用场景



# 组合模式设计方案



# 设计模式总结

- 使用设计模式意味着复用设计思想，需要根据具体问题上下文进行实例化
- 学习通用设计模式并领会其设计思想有利于设计模式的合理应用
- 对设计模式的理解和掌握以及对设计模式适用性的判断需要设计经验积累
- 很多时候设计模式都是随着软件演化过程中对设计问题认识的不断深入而逐步引入的



## 阅读建议

- 《软件工程》 7.2
- 《代码大全》 5.5

快速阅读后整理问题  
在微信群中提出并讨论

# SOFT130006

# 软件工程

End

10. 软件设计  
— 设计模式