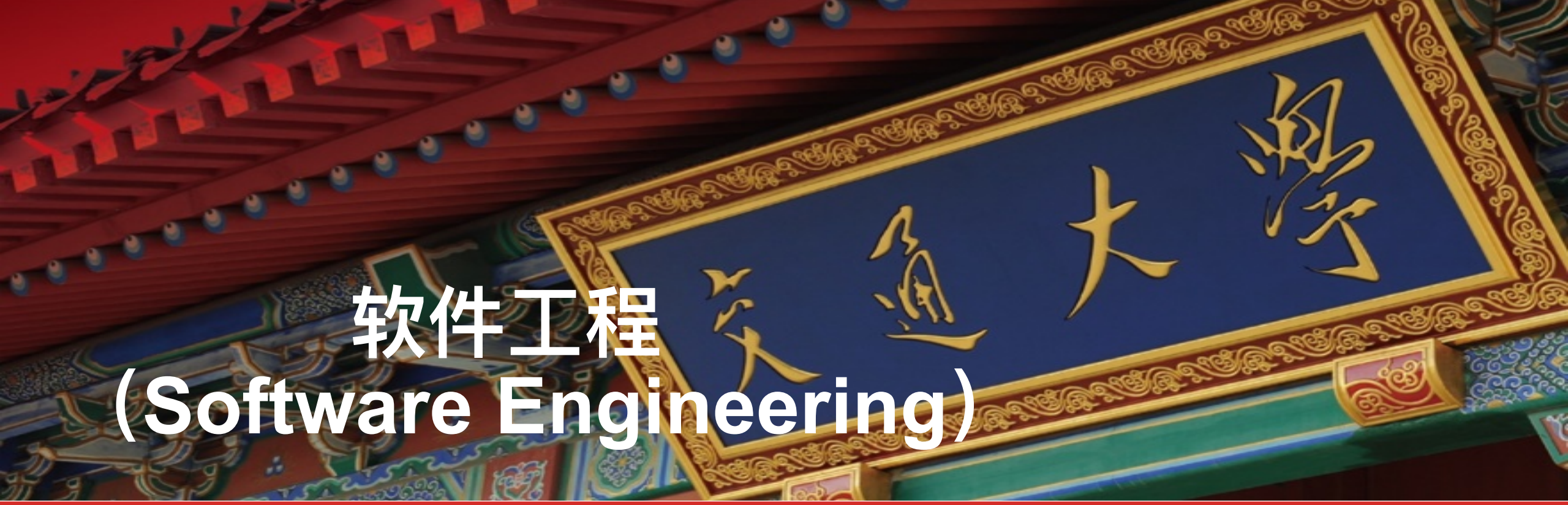


软件工程
（Software Engineering）

5. 体系结构

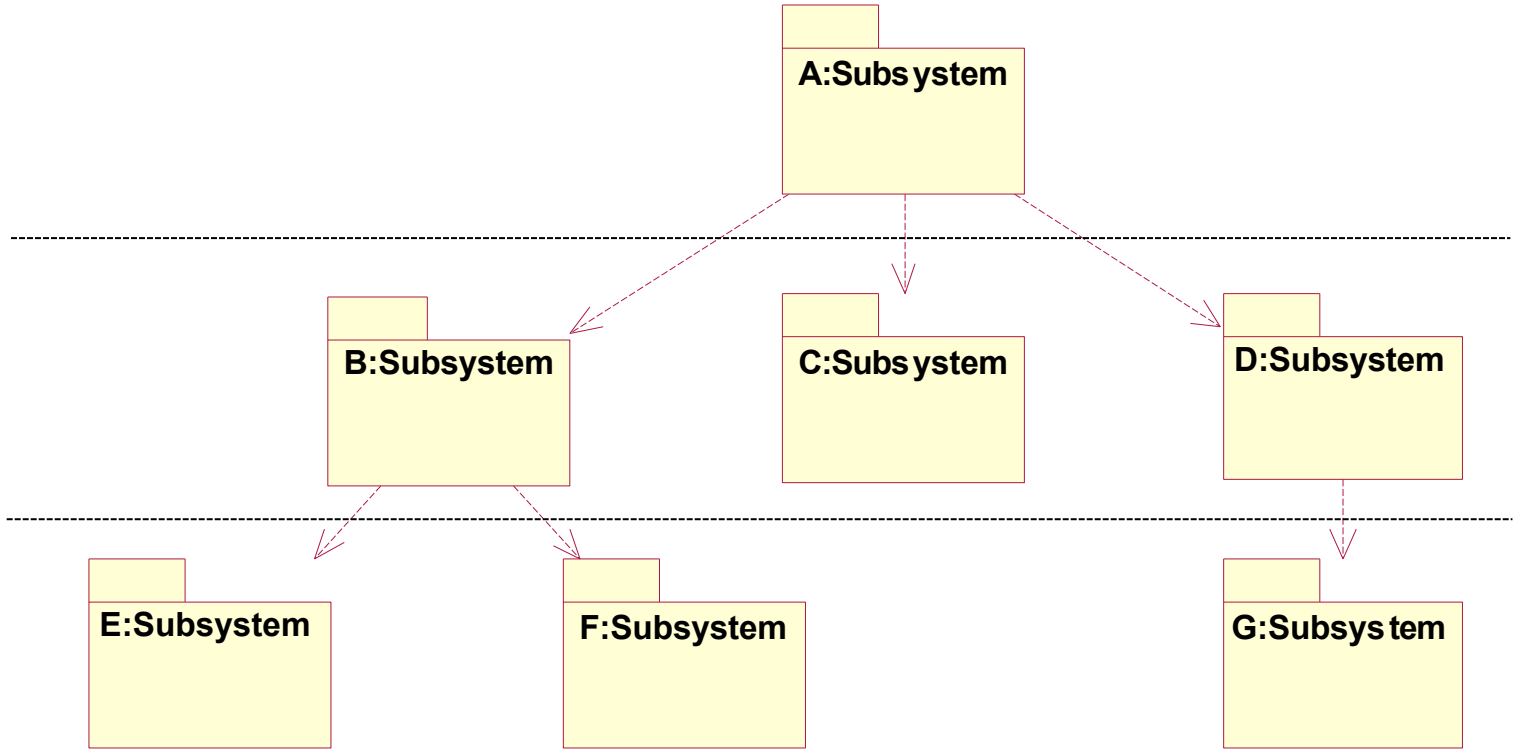
上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

目录 Contents

软件体系结构

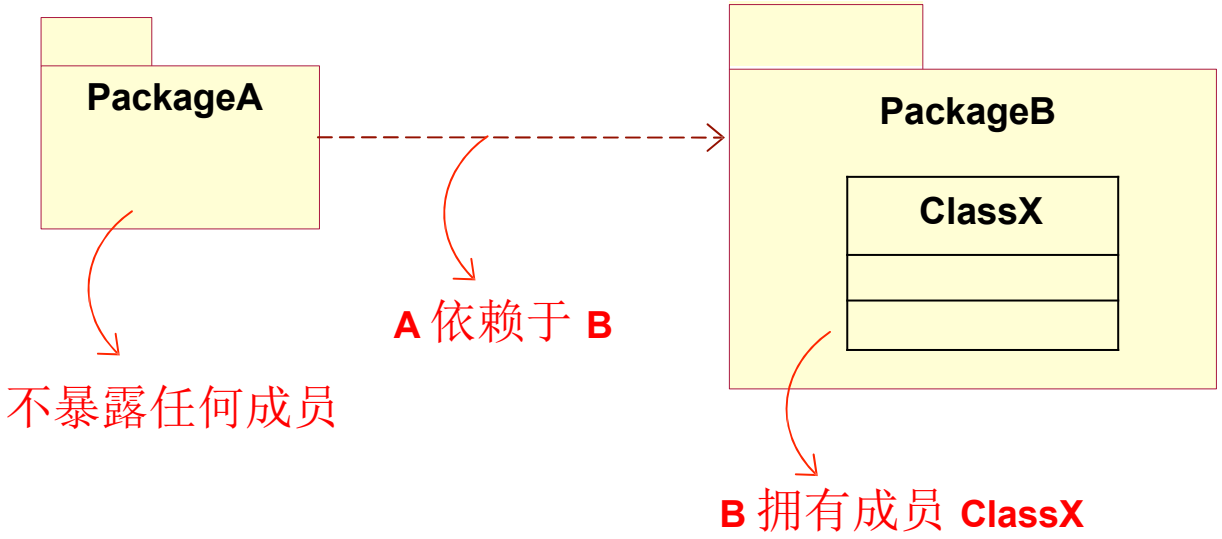
- 软件体系结构包括一组软件部件、软件部件的外部的可见特性及其相互关系，其中软件外部的可见特性是指软件部件提供的服务、性能、特性、错误处理、共享资源使用等。
  - 系统的总体组织结构和全局控制结构
  - 通信、同步和数据访问的协议
  - 设计元素的组成与功能分配
  - 非功能需求
  - 系统的物理部署
  - 备选设计方案的选择

软件体系结构

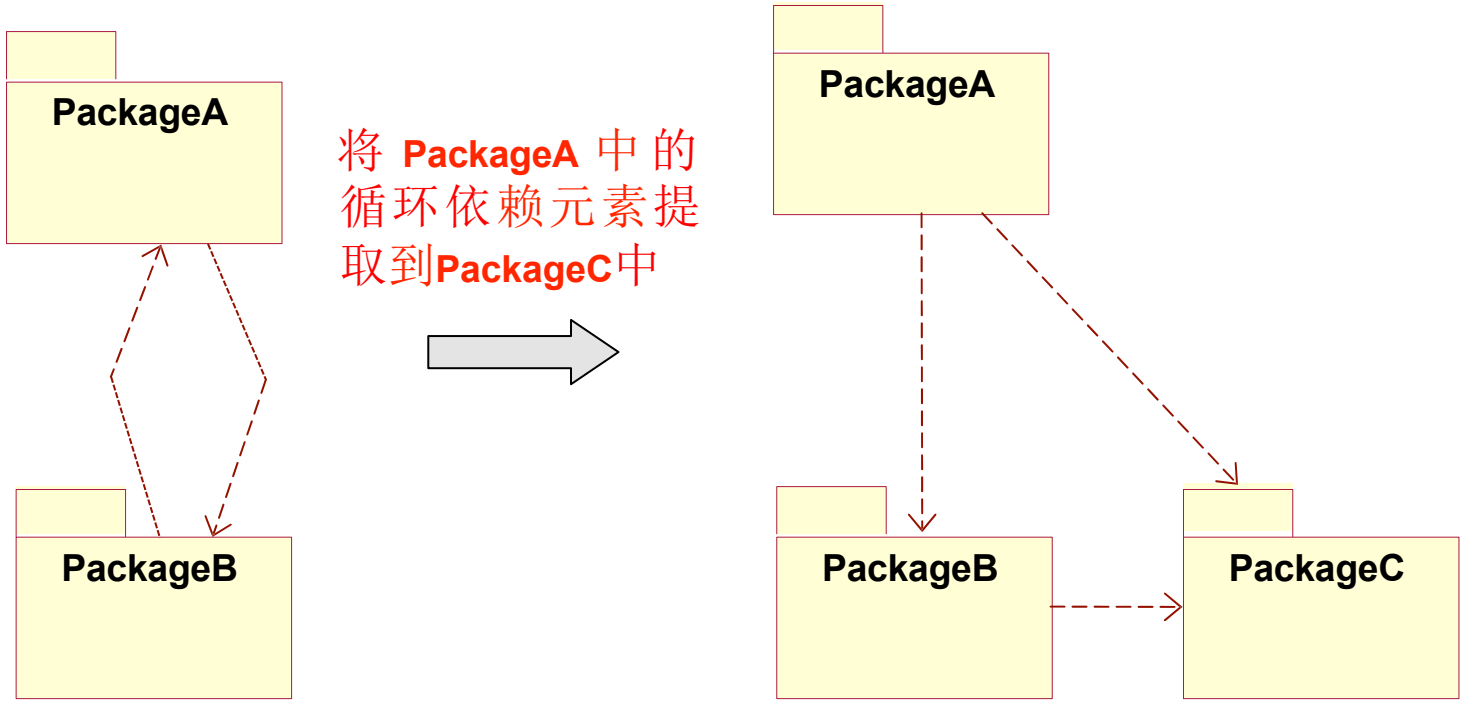

A:Subsystem
B:Subsystem
C:Subsystem
D:Subsystem
E:Subsystem
F:Subsystem
G:Subsystem

包依赖性

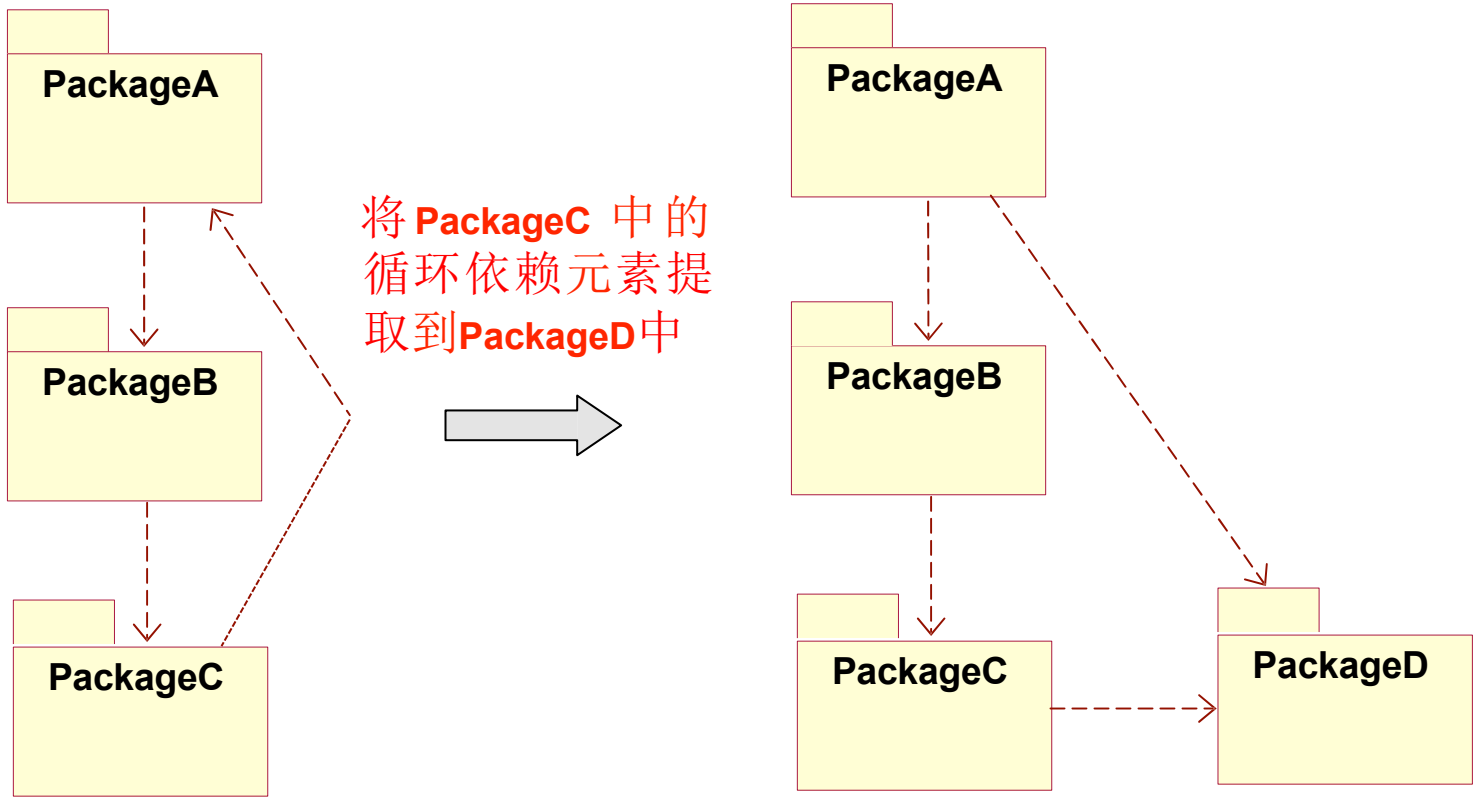
- 依赖性
  - **PackageA** 的一些成员引用 **PackageB** 的某些成员
  - **PackageB** 的变化可能会影响到 **PackageA**



循环依赖的消除


PackageA
PackageB
将 PackageA 中的循环依赖元素提取到PackageC中
PackageA
PackageB
PackageC

循环依赖的消除


PackageA
PackageB
PackageC
将 PackageC 中的循环依赖元素提取到PackageD中
PackageA
PackageB
PackageC
PackageD

SDD （IEEE 1016－1998）

1. 引言
   1. 目的
   2. 范围
   3. 定义和缩写词
2. 参考文献
3. 分解说明
   1. 模块分解
   2. 并发进程
   3. 数据分解
4. 依赖关系说明
   1. 模块间的依赖关系
   2. 进程间的依赖关系
   3. 数据间的依赖关系
5. 接口说明
   1. 模块接口
   2. 进程接口
6. 详细设计
   1. 模块详细设计
   2. 数据详细设计

附录

注： **3－5**部分是体系结构设计
6部分是详细设计

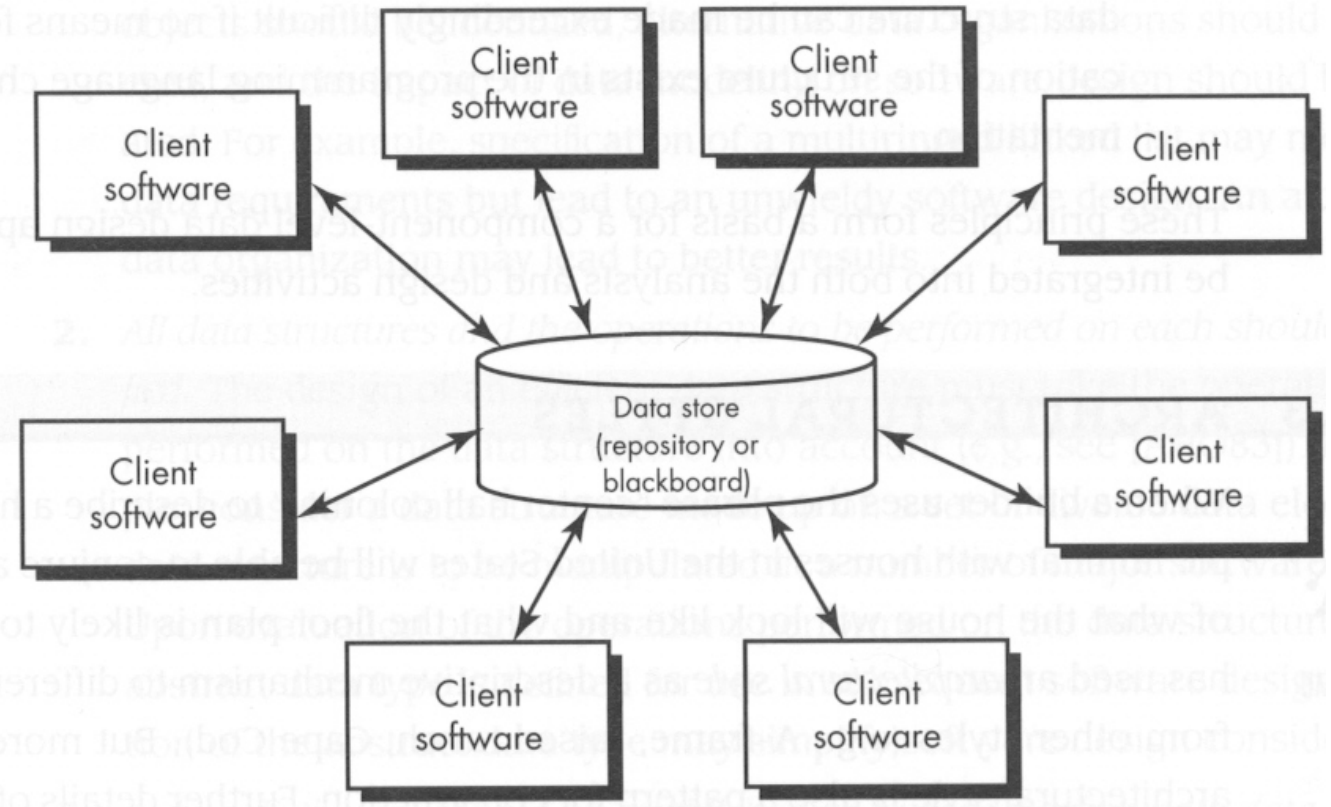
目录 Contents

体系结构风格

- 软件体系结构风格是描述某一特定应用领域中系统组织方式的惯用模式，它反映了领域中众多系统所共有的结构和语义特性，并指导如何将各个模块和子系统有效地组织成一个完整系统。
- 典型的软件体系结构风格
  - 仓库或知识库结构
  - 模型 / 视图 / 控制器体系结构
  - 控制结构
  - 客户机 / 服务器结构
  - 分层体系结构

仓库结构

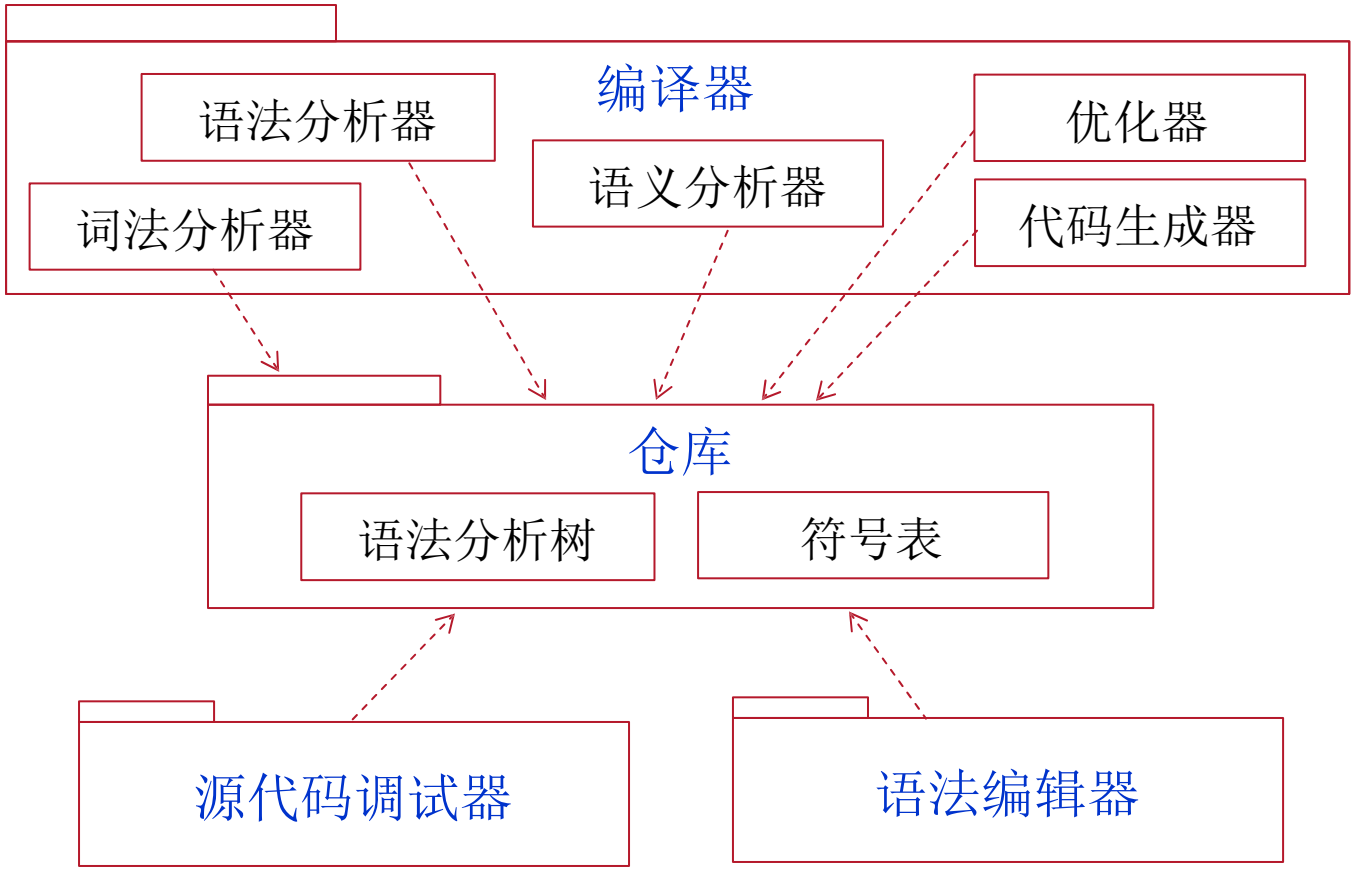
- 仓库或知识库结构（ ***Repository architecture*** ）



仓库结构

- 仓库结构是一种以数据为中心的体系结构，它包含一个中心数据库和一组相互独立的处理中心数据的子系统，主要适合于数据由一个子系统产生而由其他子系统使用的情形。
- 优点：在共享数据模型稳定的情况下，扩展新的子系统十分容易
- 缺点：子系统与共享数据之间的耦合度很高，共享数据将对系统的性能和子系统的修改产生瓶颈。
- 应用：现代编译器、管理信息系统、**CAD** 系统和 **CASE** 工具集等。

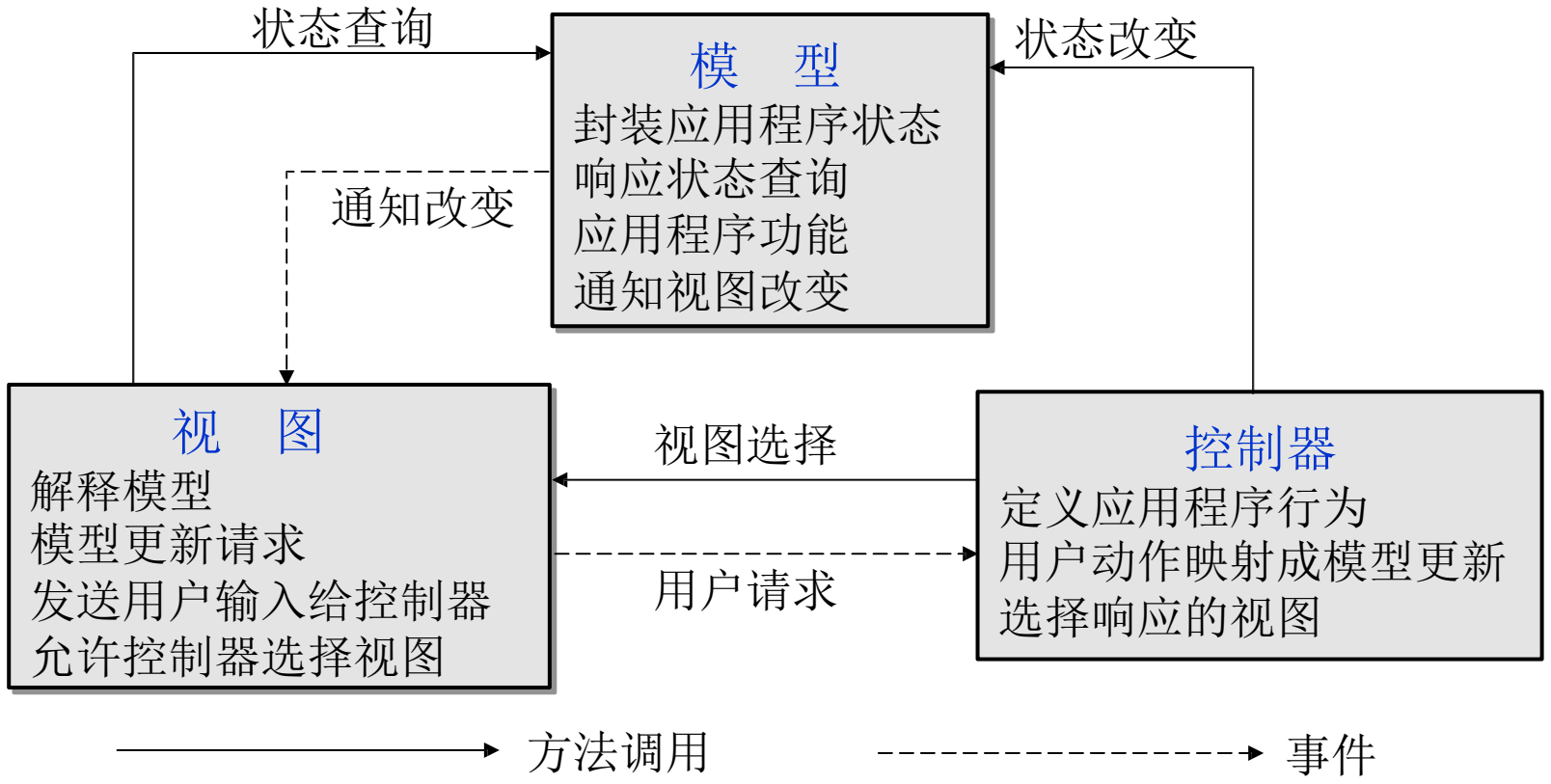
仓库结构



模型／视图／控制器结构

- 模型/视图/控制器结构（*Model/View/Controller Architecture*）
  - 该结构是为同样的数据提供多个视图的应用程序而设计的，它将交互系统的组成分解成模型、视图、控制器三种部件。
    - 视图是应用程序中用户界面相关的部分，即用户看到并与之交互的界面。
    - 控制器工作就是根据用户的输入，控制用户界面数据显示和更新模型对象的状态。
    - 模型是应用程序的主体部分，表示业务数据或者业务逻辑。
  - 该结构适合于交互式系统，特别是同一个模型需要多个视图的情况。

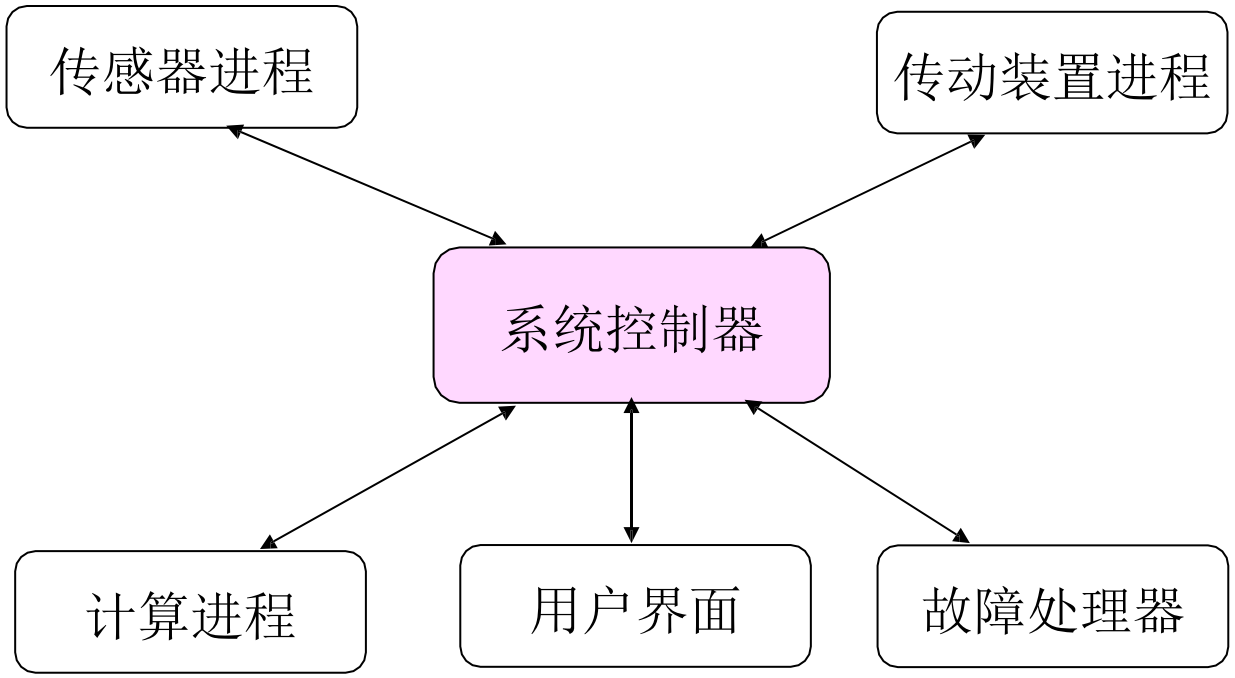
模型／视图／控制器结构


状态查询
模　型
封装应用程序状态
响应状态查询
应用程序功能
通知视图改变
状态改变
通知改变
视　图
解释模型
模型更新请求
发送用户输入给控制器
允许控制器选择视图
视图选择
用户请求
控制器
定义应用程序行为
用户动作映射成模型更新
选择响应的视图
方法调用
事件

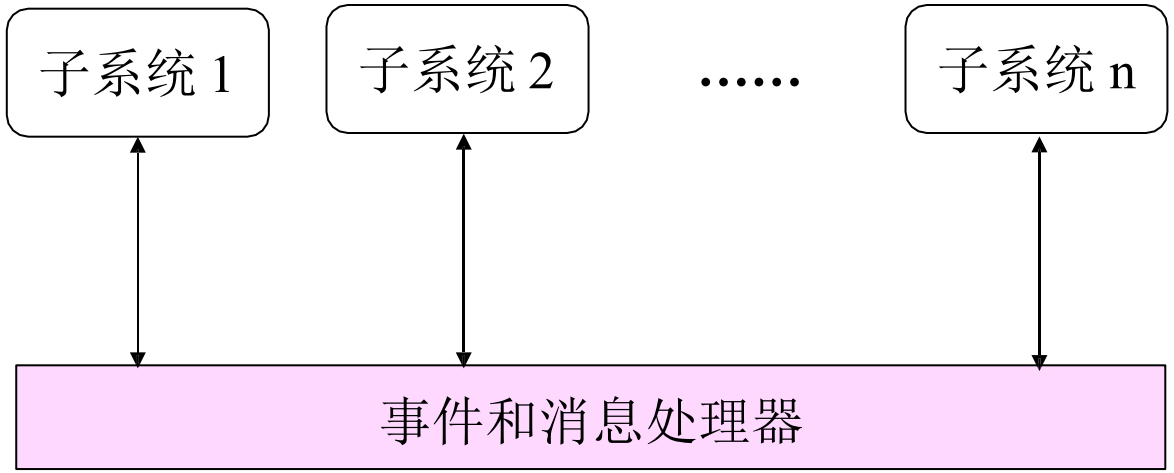
控制结构

- 该结构关心的是子系统之间的控制流
  - 集中式控制
    - 一个子系统专门负责控制，控制其他子系统的启动和停止。它也可能将控制交给一个子系统，但在控制完成后控制权仍然要归还给它。
  - 基于事件的控制
    - 控制信息不是集中于一个子系统中，而是每个子系统都能接收来自外部的事件并对此作出响应。这些事件可能来自其他子系统或来自系统的环境。

控制结构



控制结构

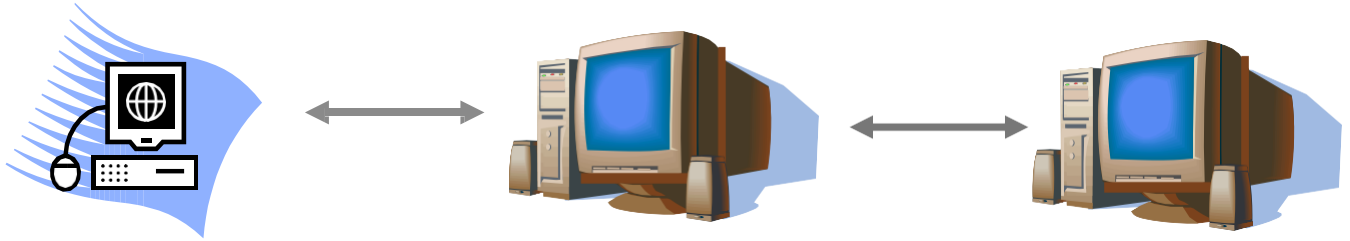


客户机 / 服务器结构

- 客户机 / 服务器结构（ ***Client/Server Architecture*** ）
  - 在客户机 / 服务器体系结构中，作为服务器的子系统为其他客户机的子系统提供服务，作为客户机的子系统负责与用户的交互。
- 瘦客户机模型
  - 所有的应用处理和数据管理都是在服务器上执行，客户机只是负责数据表示部分。
  - 由于繁重的处理负荷全部集中在服务器和网络上，有可能造成性能上的问题。

客户机 / 服务器结构

- 胖客户机模型
  - 服务器只负责对数据的管理，客户机上的软件实现应用逻辑与用户的交互。
  - 系统管理更加复杂，因为应用程序的改变必须在客户机上重新安装。
- 三层的客户机 / 服务器体系结构



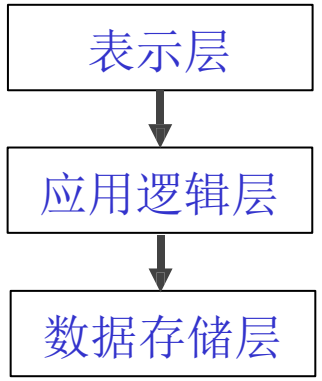
客户机
- 数据表示

应用服务器
- 应用处理

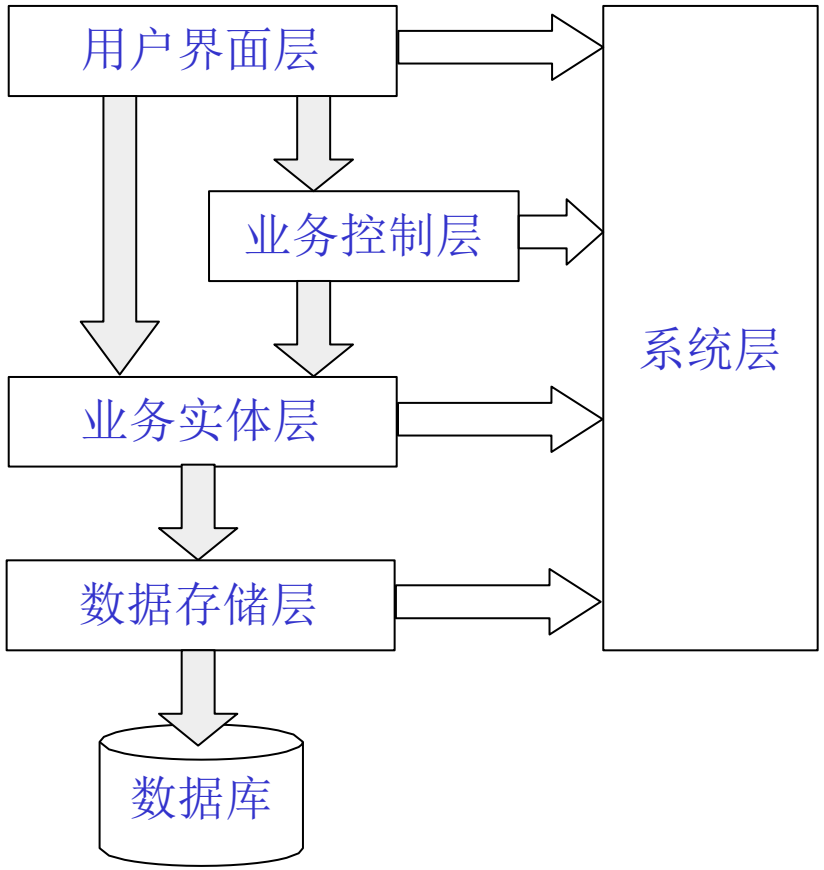
数据服务器
- 数据管理

分层体系结构

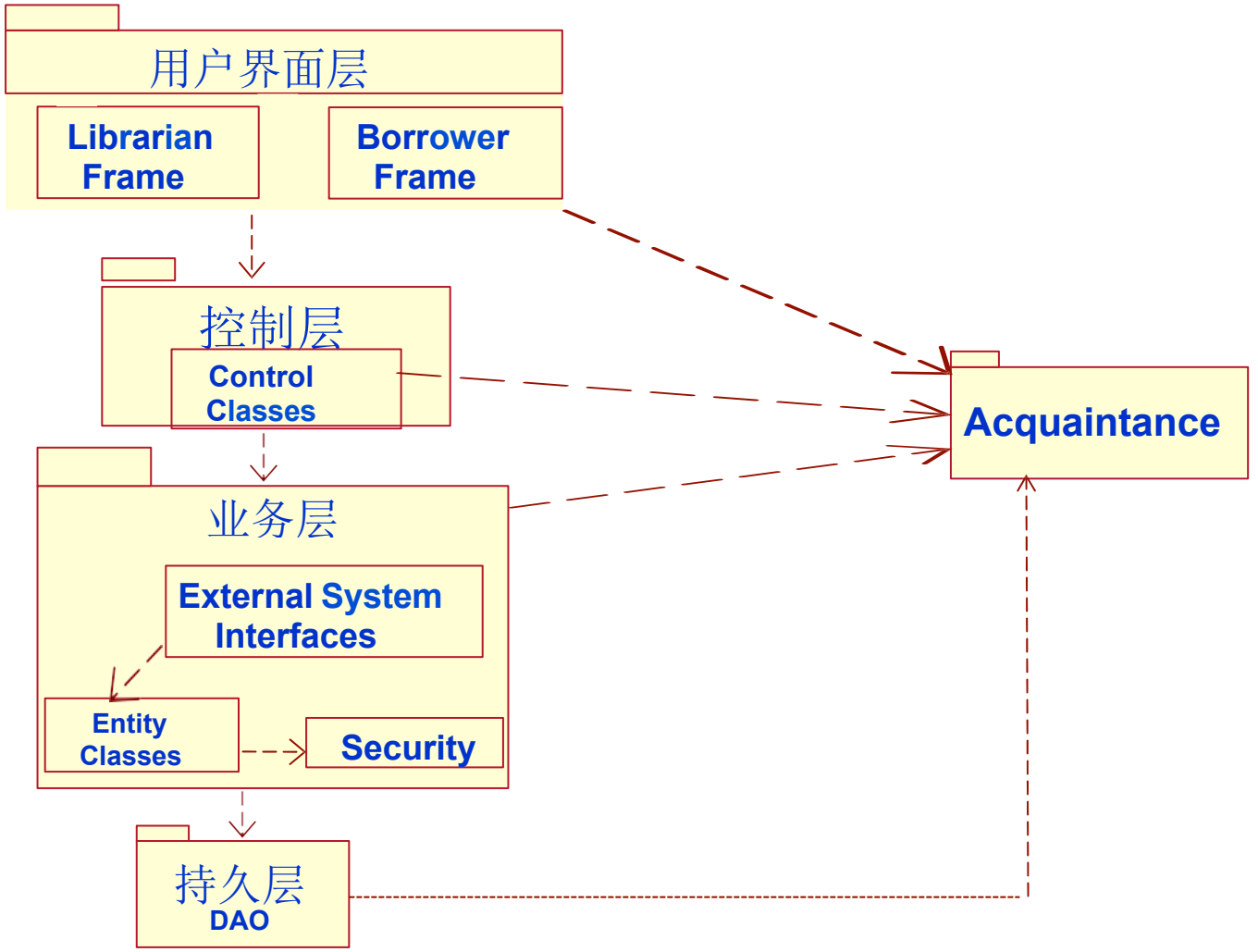
- 层次化是一种概念，它将软件设计组织成为类或组件的层次或集合，在同一个层次上的类或组件完成一个特定的目的。
- 良好的层次结构可以易于系统的扩展与维护，不同的层次之间通过接口进行通信。
- 三层体系结构（ ***Three-tier Architecture*** ）
  - 表示层：窗口、报表等用户界面元素
  - 应用逻辑层：管理业务过程的任务和规则
  - 存储层：持久化存储机构



分层体系结构



MiniLibrary：软件体系结构



目录 Contents

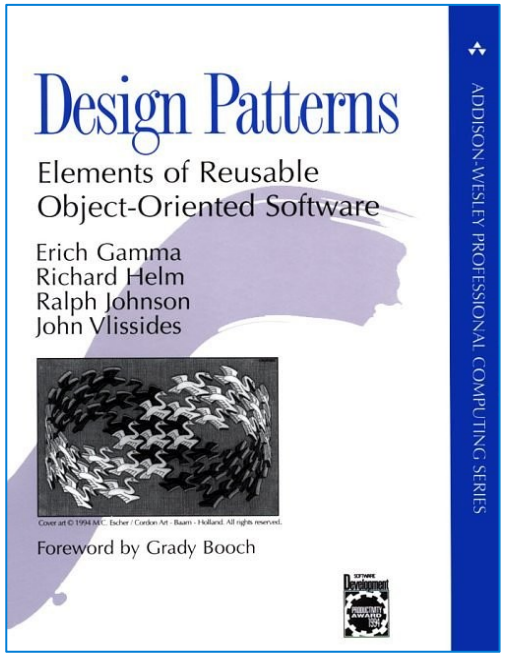
设计模式

- 回顾学过的数据结构
  - ***Trees, Stacks, Queues***
  - 它们给软件开发带来了什么？
- 问题
  - 在软件体系结构设计中是否存在一些可重用的解决方案？
- 答案是肯定的
  - 设计模式使我们可以重用已经成功的经验
  - ***Pattern = Documented experience***

设计模式

- ***Design Patterns: Elements of Reusable Object-Oriented Software***

- Gang of Four
  - Gamma, Helm, Johnson, Vlissides



设计模式

- 设计模式描述了软件系统设计过程中常见问题的解决方案，它是从大量的成功实践中总结出来的且被广泛公认的实践和知识。
- 设计模式的好处
  - 使人们可以简便地重用已有的良好设计
  - 提供了一套可供开发人员交流的语言
  - 提升了人们看待问题的抽象程度
  - 帮助设计人员更快更好地完成系统设计
  - 模式是经过考验的思想，具有更好的可靠性和扩展性

设计模式的基本要素

- 模式名称
  - 一个助记名，便于交流和思考
- 问题
  - 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
- 解决方案
  - 描述设计的组成部分，它们之间的相互关系以及各自的职责和协作方式
- 效果
  - 描述模式应用的效果以及应权衡的问题

设计模式的类型

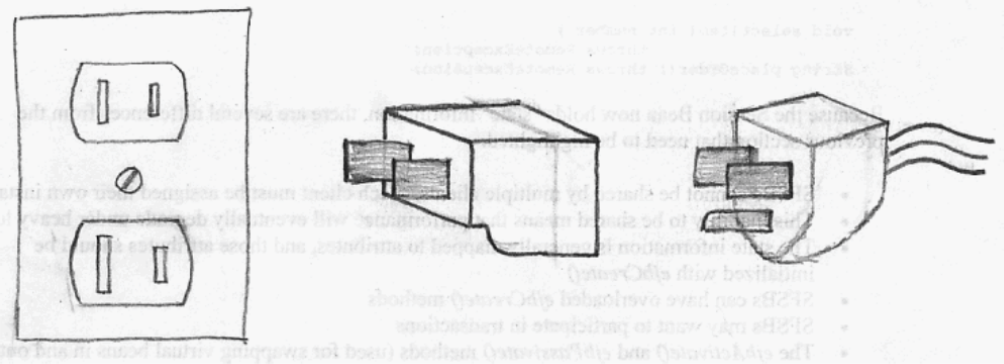
- 结构型模式
- 行为模式
- 创建型模式

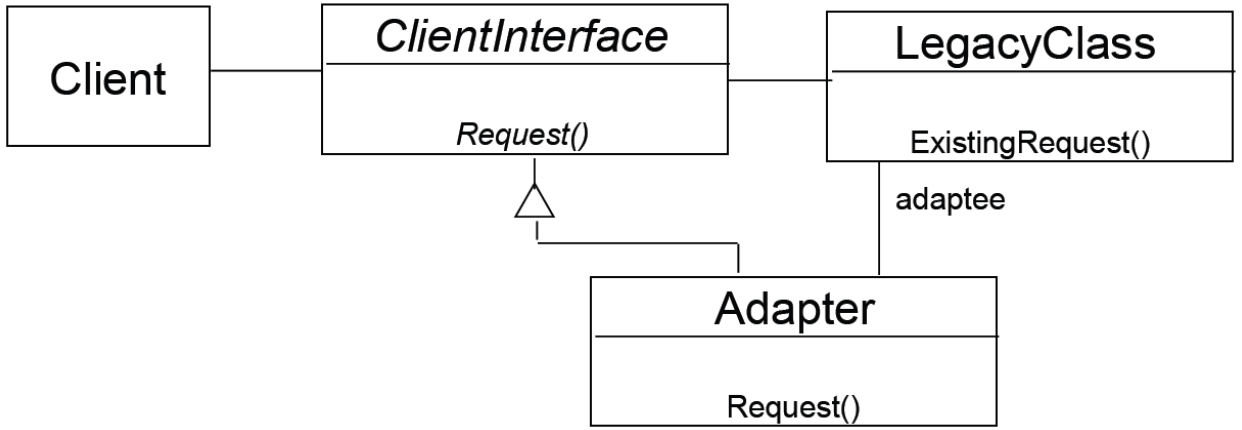
设计模式的类型

- 结构型模式
  - 结构型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，和类之间功能重叠的问题。
  - 结构型模式采用继承机制来组合接口或实现。

- 典型的模式
  - 适配器 *Adapter*、桥接 *Bridge*、组成 *Composite*
  - 装饰 ***Decorator***、外观 *Facade*、享元 ***Flyweight***、代理 *Proxy*

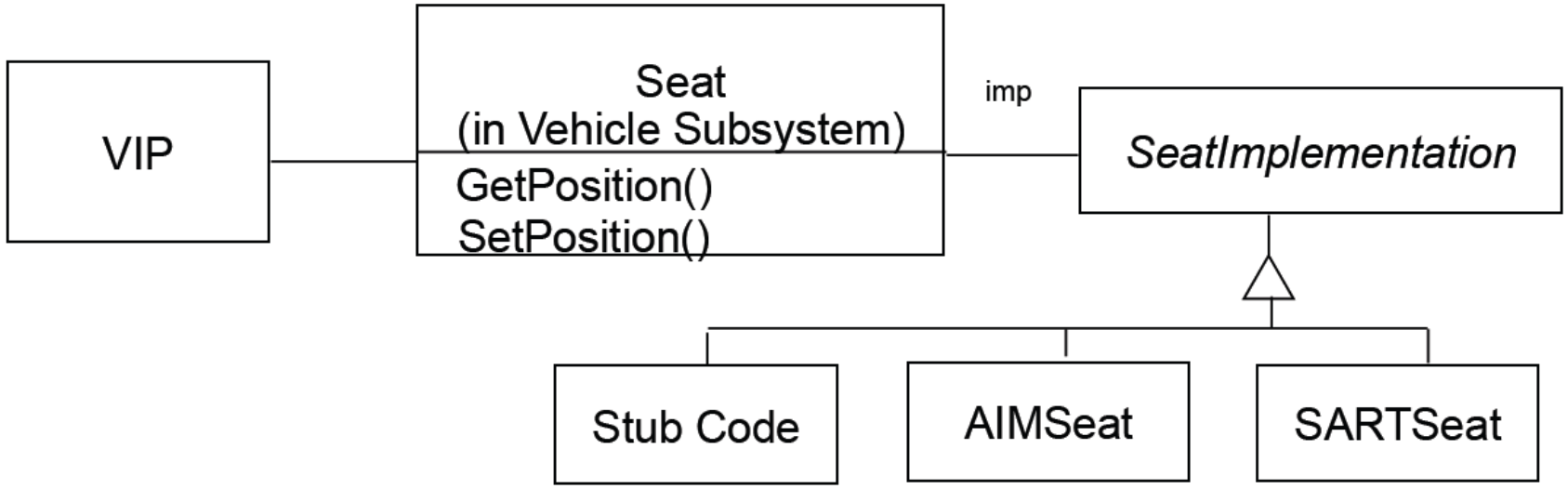
Adapter



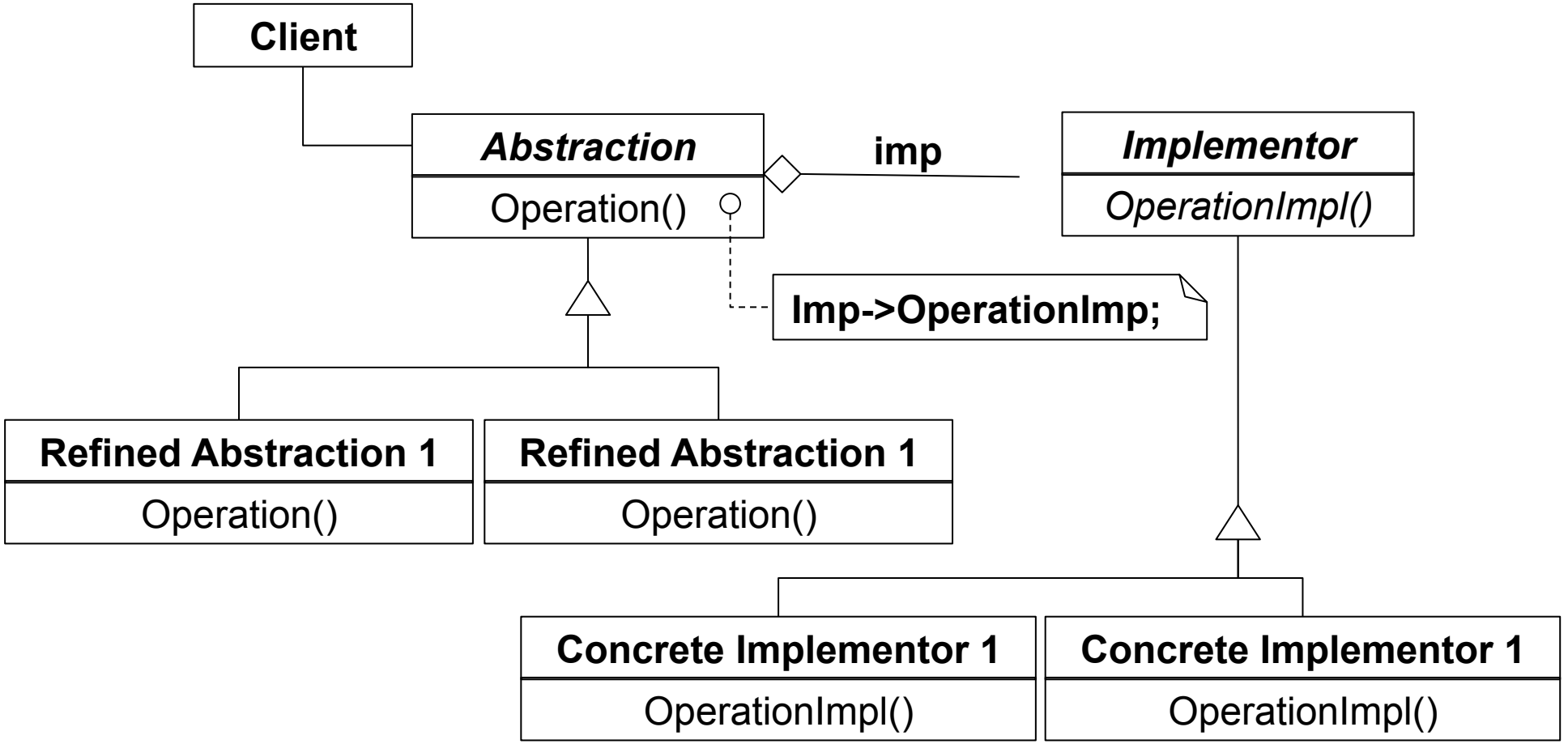


- Adepter将不兼容接口的类连接起来，一起工作
- 用接口继承将Adapter与Adaptee连接起来

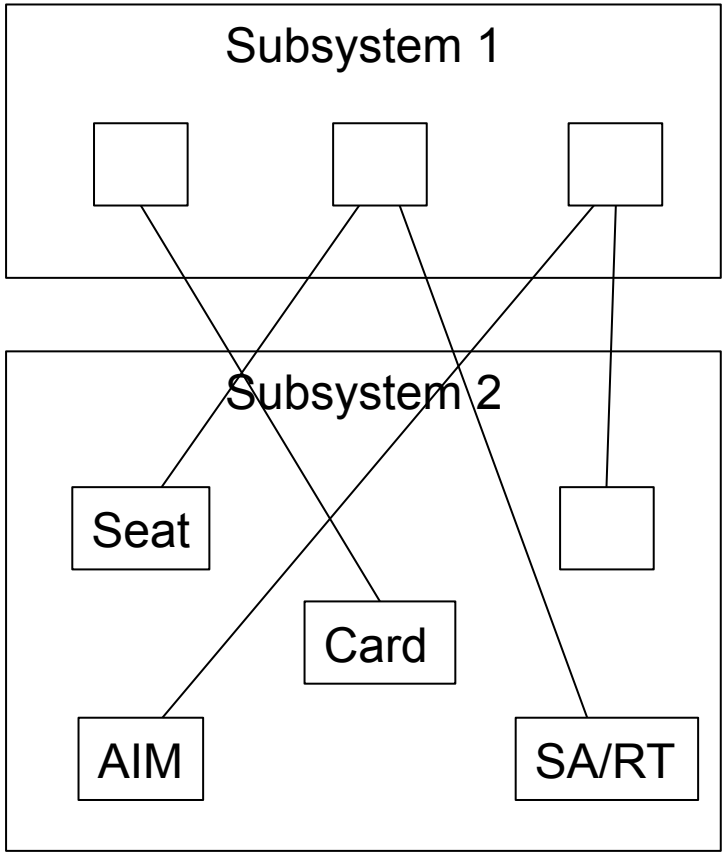
Bridge



- Use a bridge to “decouple an abstraction from its implementation so that the two can vary independently”. (From [Gamma et al 1995])
- Bridge 用于在相同的接口下提供多个实现


Client
Abstraction
Operation()
imp
Implementor
OperationImpl()
Imp->OperationImp;
Refined Abstraction 1
Operation()
Refined Abstraction 1
Operation()
Concrete Implementor 1
OperationImpl()
Concrete Implementor 1
OperationImpl()

Façade



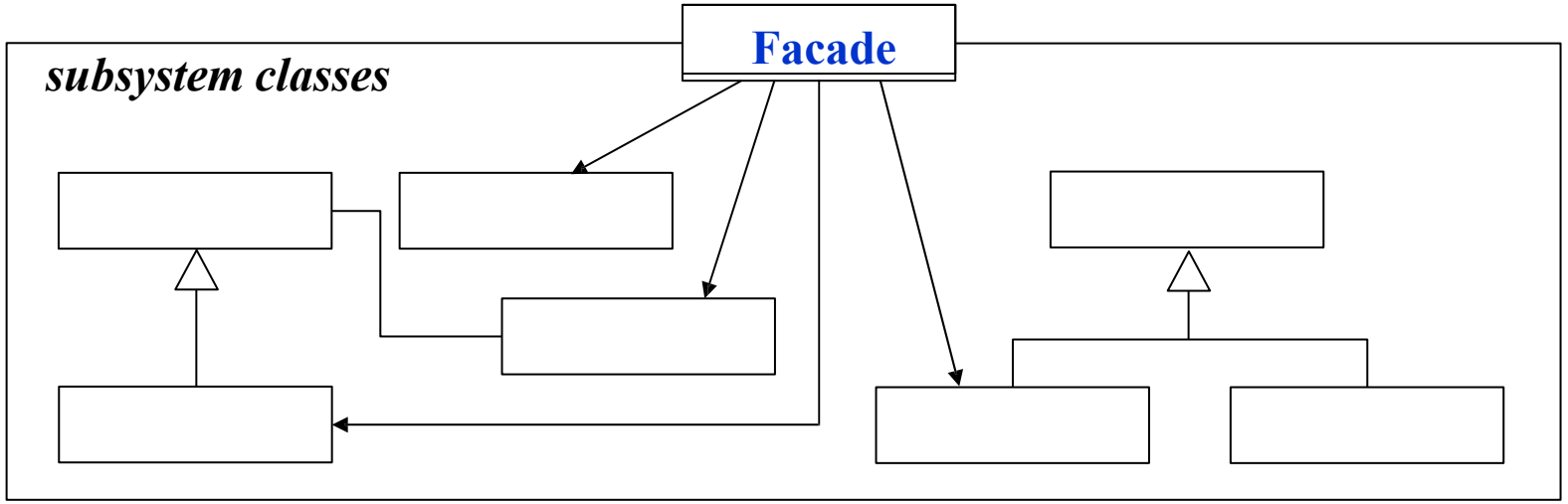
Why is this good?

效率高 Efficiency

Why is this bed?

需要了解子系统和系统中复杂的关系
子系统会本错误的利用
编程复杂

Façade



– 外观模式用简单的统一接口封装子系统，从而降低类之间的相关性。

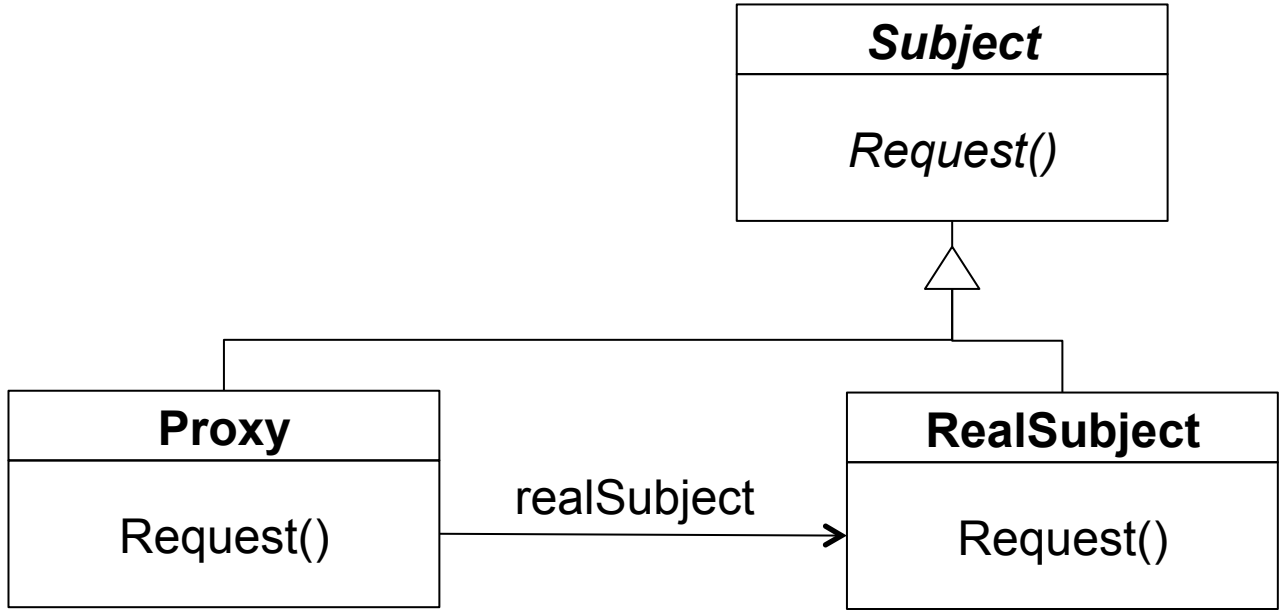
Proxy

What is expensive?

- 对象创建
- 对象初始化

Proxy pattern:

- 降低对象访问的代价
- 用另个对象(“the proxy”) 代替真实的对象
- 仅当用户需要时才创建真实的对象



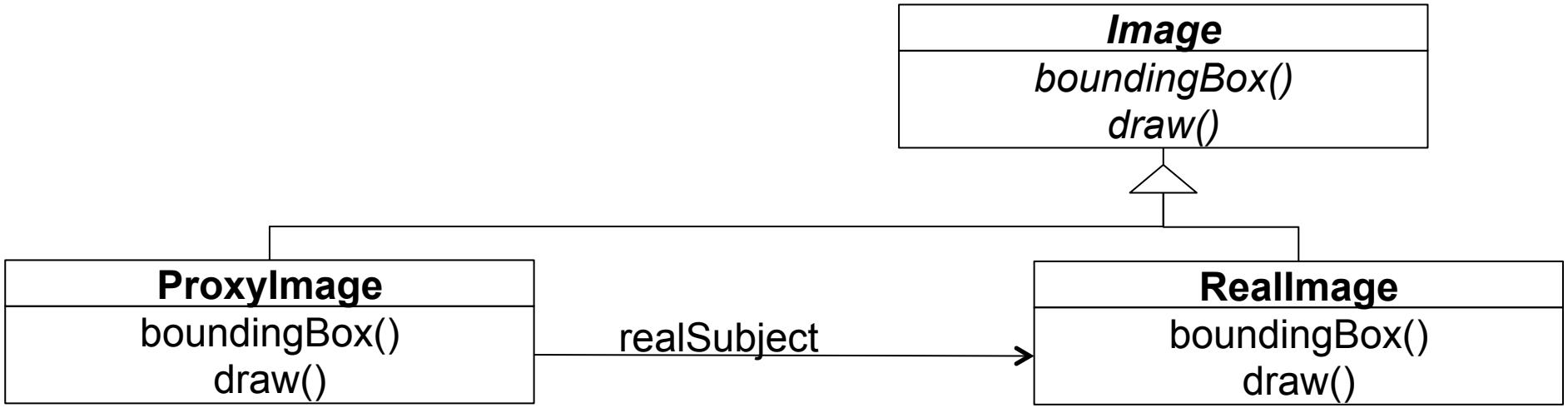
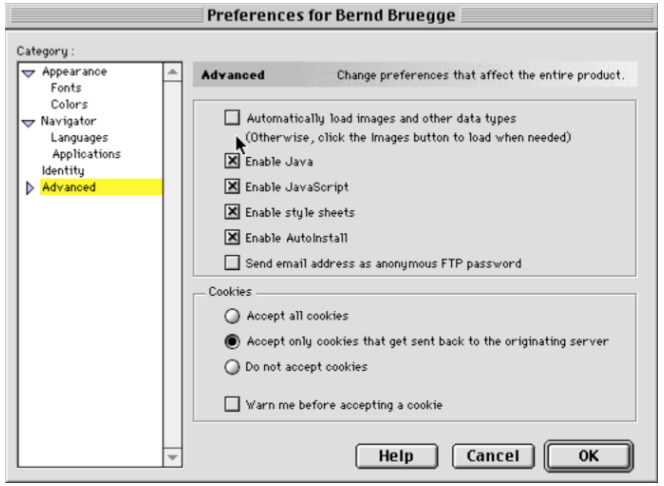
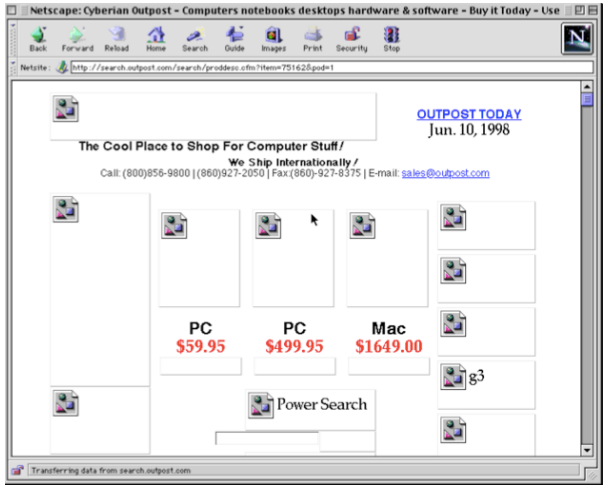


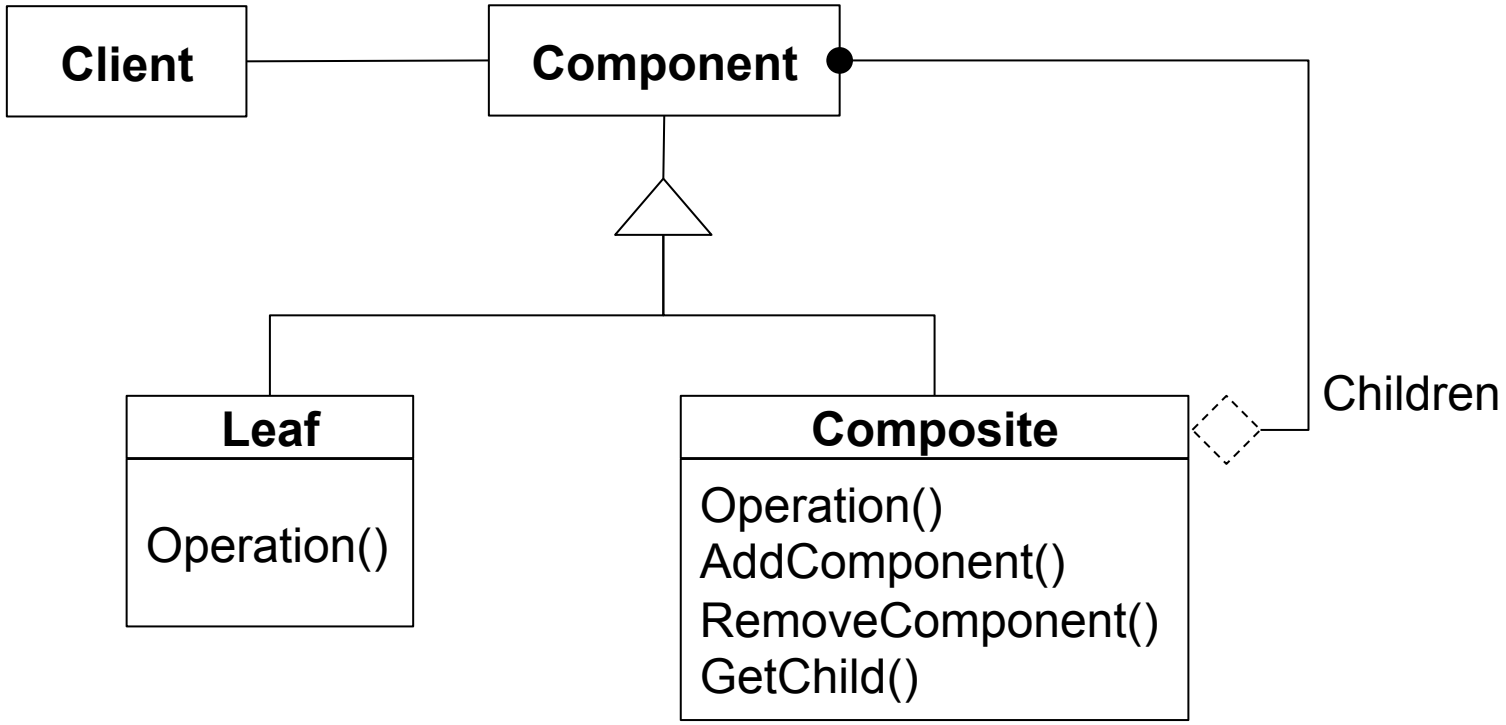
Image
boundingBox()
draw()
ProxyImage
boundingBox()
draw()
realSubject
RealImage
boundingBox()
draw()


Netscape: Cyberian Outpost - Computers notebooks desktops hardware & software - Buy it Today - Use
CYBERiAN Outpost
The Cool Place to Shop For Computer Stuff!
OUTPOST TODAY
Jun. 10, 1998
PC $59.95
Mac $1549.00
PC $69.95
PRODUCT SEARCH

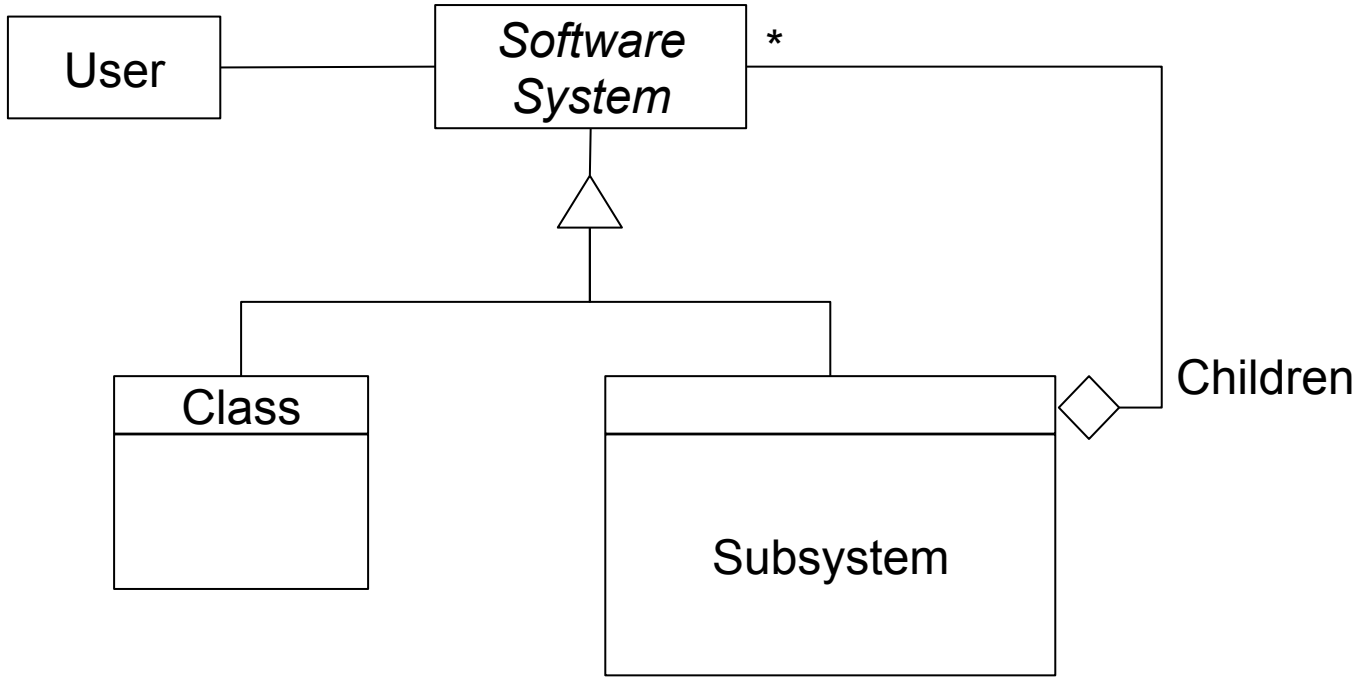

Preferences for Bernd Bruegge
Category :
Appearance
Fonts
Colors
Navigator
Languages
Applications
Identity
Advanced
Advanced
Change preferences that affect the entire product.
Automatically load images and other data types
(Otherwise, click the Images button to load when needed)
Enable Java
Enable JavaScript
Enable style sheets
Enable AutoInstall
Send email address as anonymous FTP password
Cookies
Accept all cookies
Accept only cookies that get sent back to the originating server
Do not accept cookies
Warn me before accepting a cookie
Help
Cancel
OK


Netscape: Cyberian Outpost - Computers notebooks desktops hardware & software - Buy it Today - Use
OUTPOST TODAY
Jun. 10, 1998
The Cool Place to Shop For Computer Stuff!
PC $59.95
PC $499.95
Mac $1649.00
Power Search

Composite



代表了部分-整体（part-whole），Composite方案代表了整体是由个体对象组成的

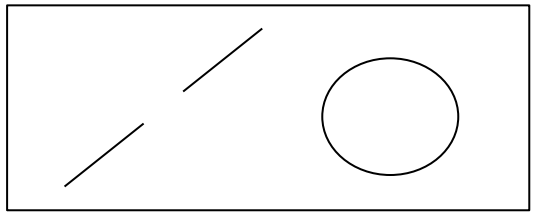


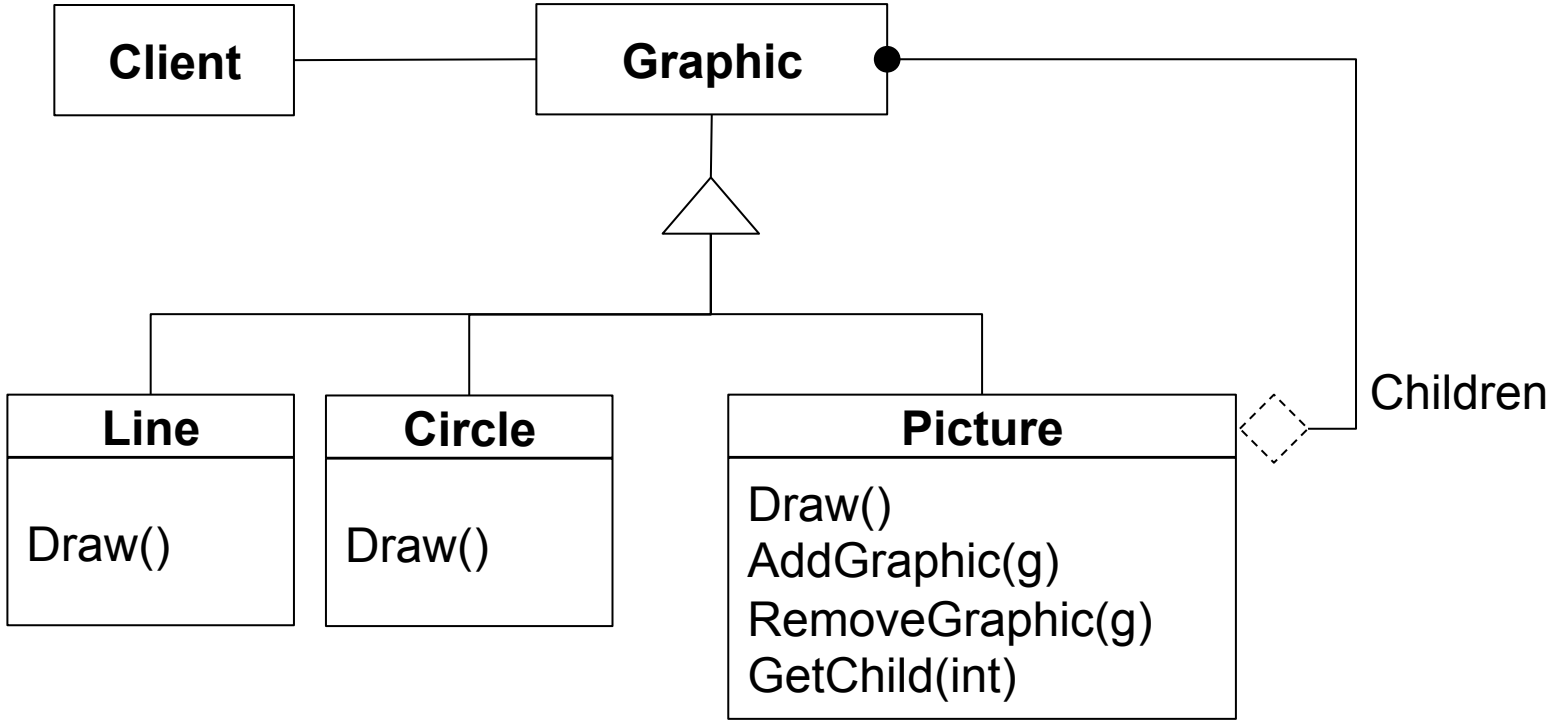
Software System:

Definition: A software system consists of subsystems which are either other subsystems or collection of classes

Composite: Subsystem (A software system consists of subsystems which consists of subsystems , which consists of subsystems, which...)
Leaf node: Class

- The *Graphic* Class represents both primitives (Line, Circle) and their containers (Picture)



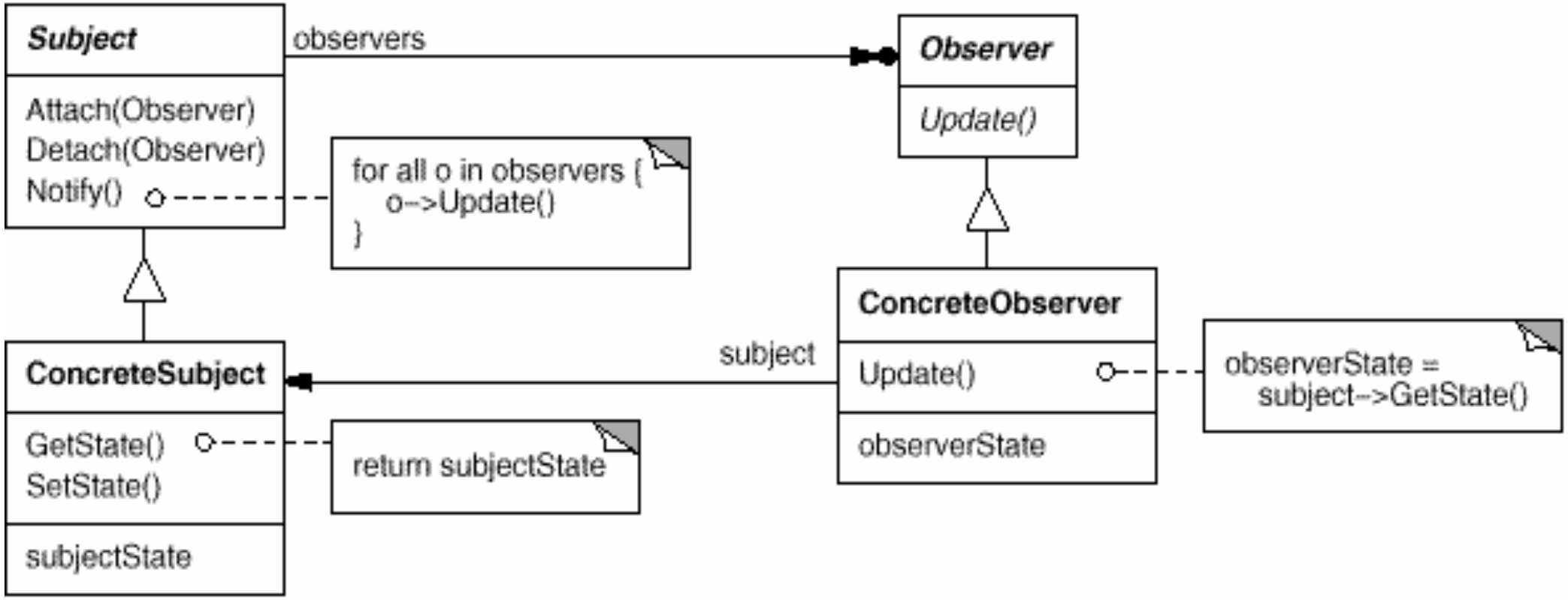


设计模式的类型

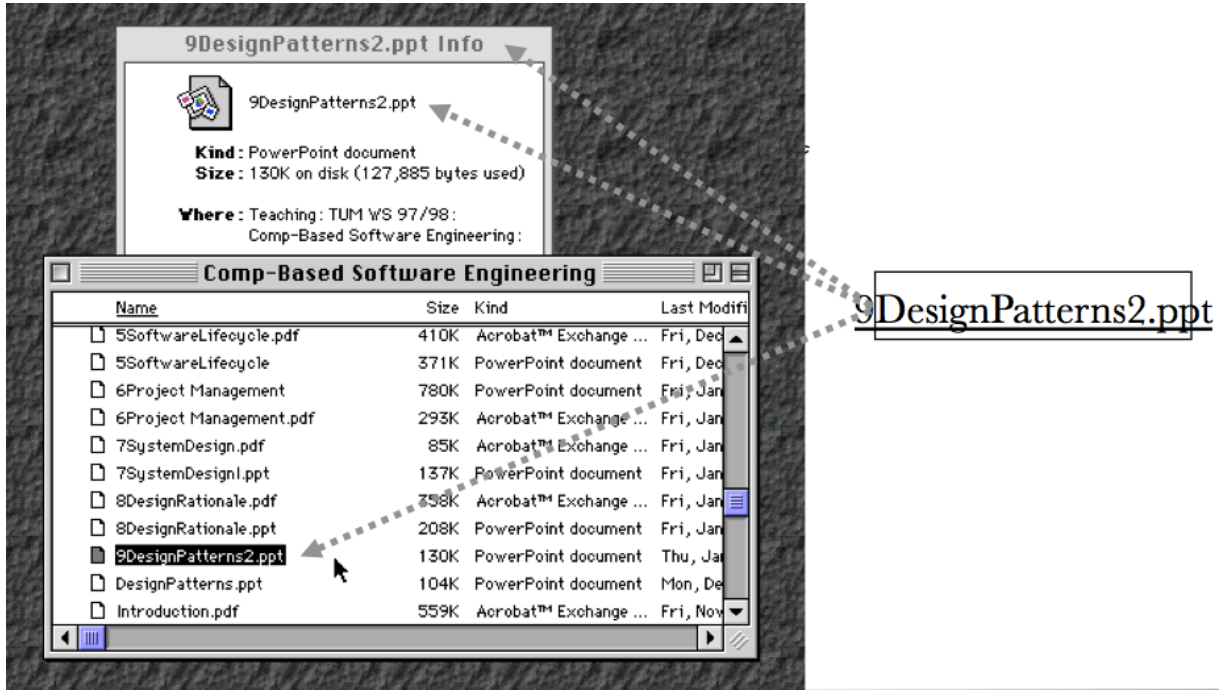
- 行为模式
  - 行为模式负责分配对象的职责，为对象间协作建模提供了有效的策略。
  - 行为模式使用继承机制在类件分配行为。

- 典型的模式
  - 职责链*Chain of Responsibility*、命令*Command*、
  - 解释器*Interpreter*、迭代器*Iterator*、中介者*Mediator*
  - 备忘录*Memento*、**观察者***Observer*、状态*State*
  - **策略***Strategy*、模板方法*Template Method*、访问者*Visitor*

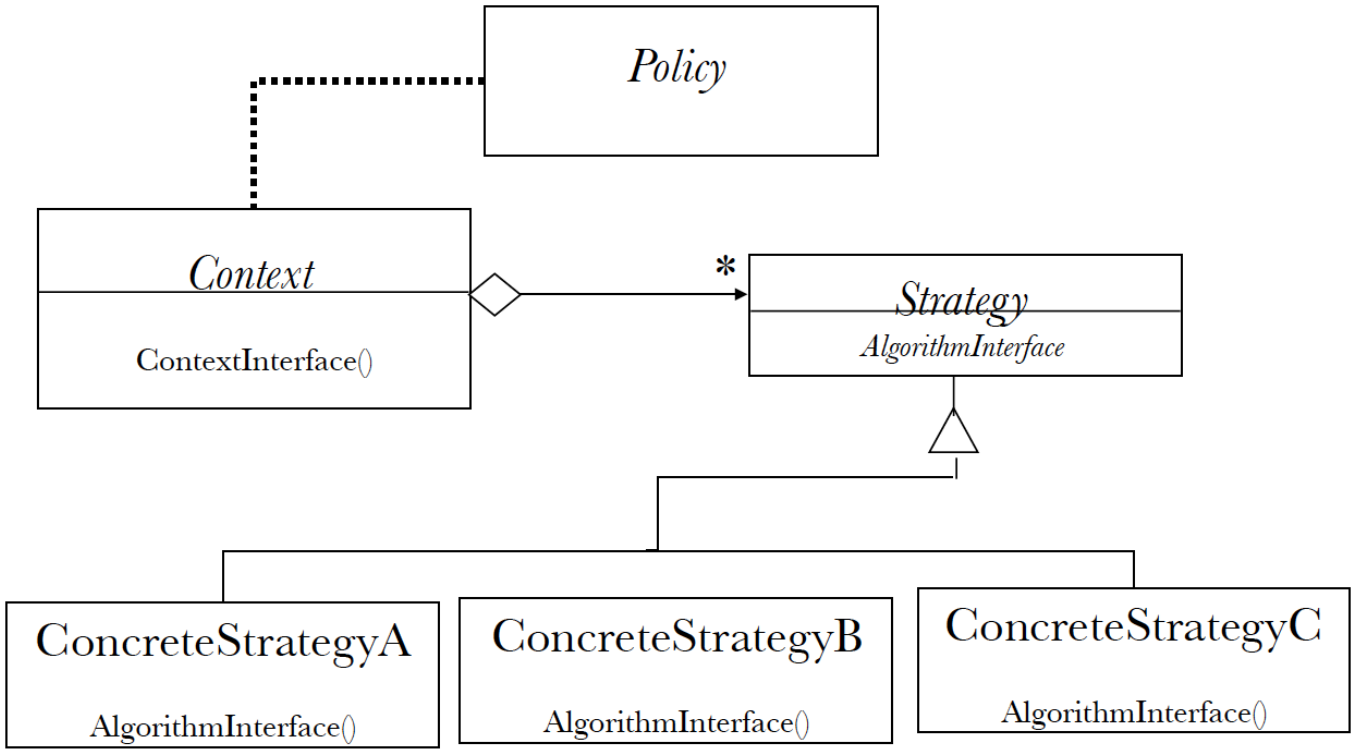
Observer



– 观察者模式用于定义对象之间一对多的依赖关系，当一个对象的状态发生变化时，所有依赖于它的对象都将得到通知而被自动更新。

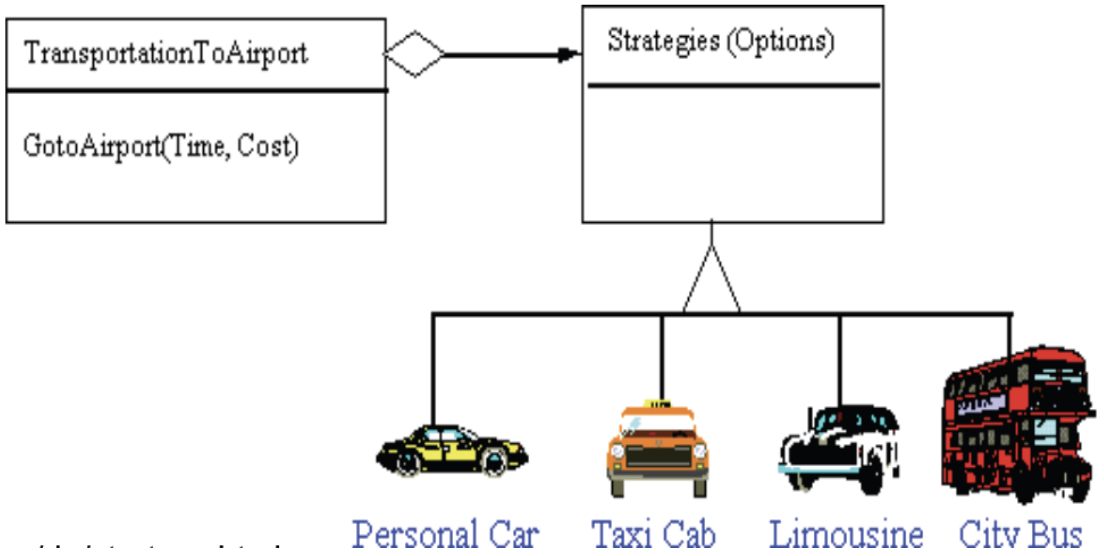

9DesignPatterns2.ppt Info
9DesignPatterns2.ppt
Kind : PowerPoint document
Size : 130K on disk (127,885 bytes used)
Where : Teaching : TUM WS 97/98 :
Comp-Based Software Engineering :
Comp-Based Software Engineering
Name
Size
Kind
Last Modifi
5SoftwareLifecycle.pdf 410K Acrobat™ Exchange ... Fri, Dec
5SoftwareLifecycle 371K PowerPoint document Fri, Dec
6Project Management 780K PowerPoint document Fri, Jan
6Project Management.pdf 293K Acrobat™ Exchange ... Fri, Jan
7SystemDesign.pdf 85K Acrobat™ Exchange ... Fri, Jan
7SystemDesignI.ppt 137K PowerPoint document Fri, Jan
8DesignRationale.pdf 358K Acrobat™ Exchange ... Fri, Jan
8DesignRationale.ppt 208K PowerPoint document Fri, Jan
9DesignPatterns2.ppt 130K PowerPoint document Thu, Ja
DesignPatterns.ppt 104K PowerPoint document Mon, De
Introduction.pdf 559K Acrobat™ Exchange ... Fri, Nov
9DesignPatterns2.ppt

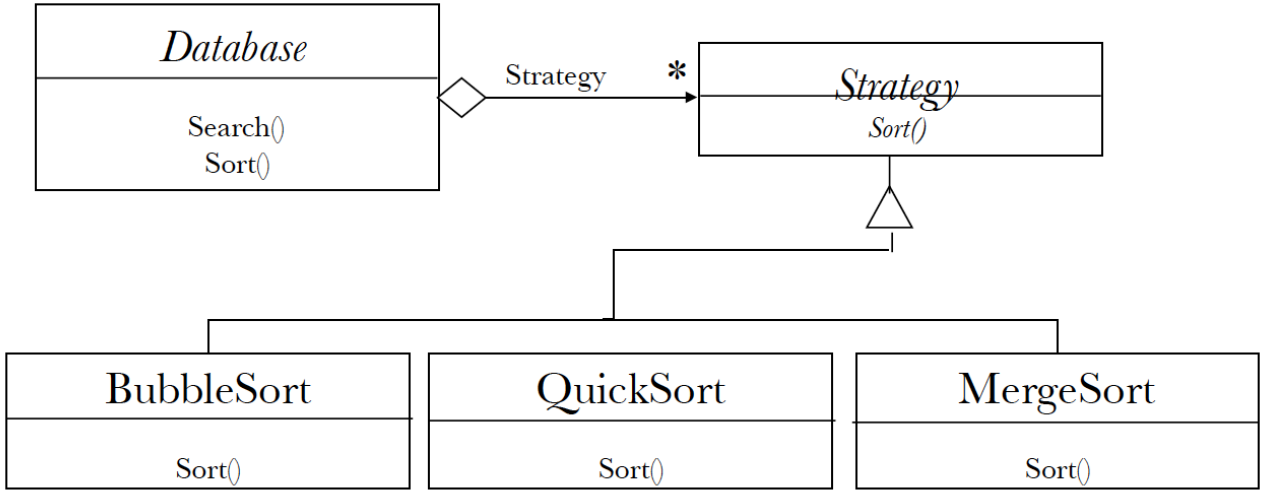
Strategy



决定哪个策略在当前情况下是最好的

- 完成一个相同的任务有不同的算法
- 不同的算法适应到不同的情况
- 在不需要的时候没有必要支持所有算法
- 如果需要新的算法，可以容易的增加新算法，但并不会干扰到软件本身的执行


TransportationToAirport
GotoAirport(Time, Cost)
Strategies (Options)
Personal Car
Taxi Cab
Limousine
City Bus

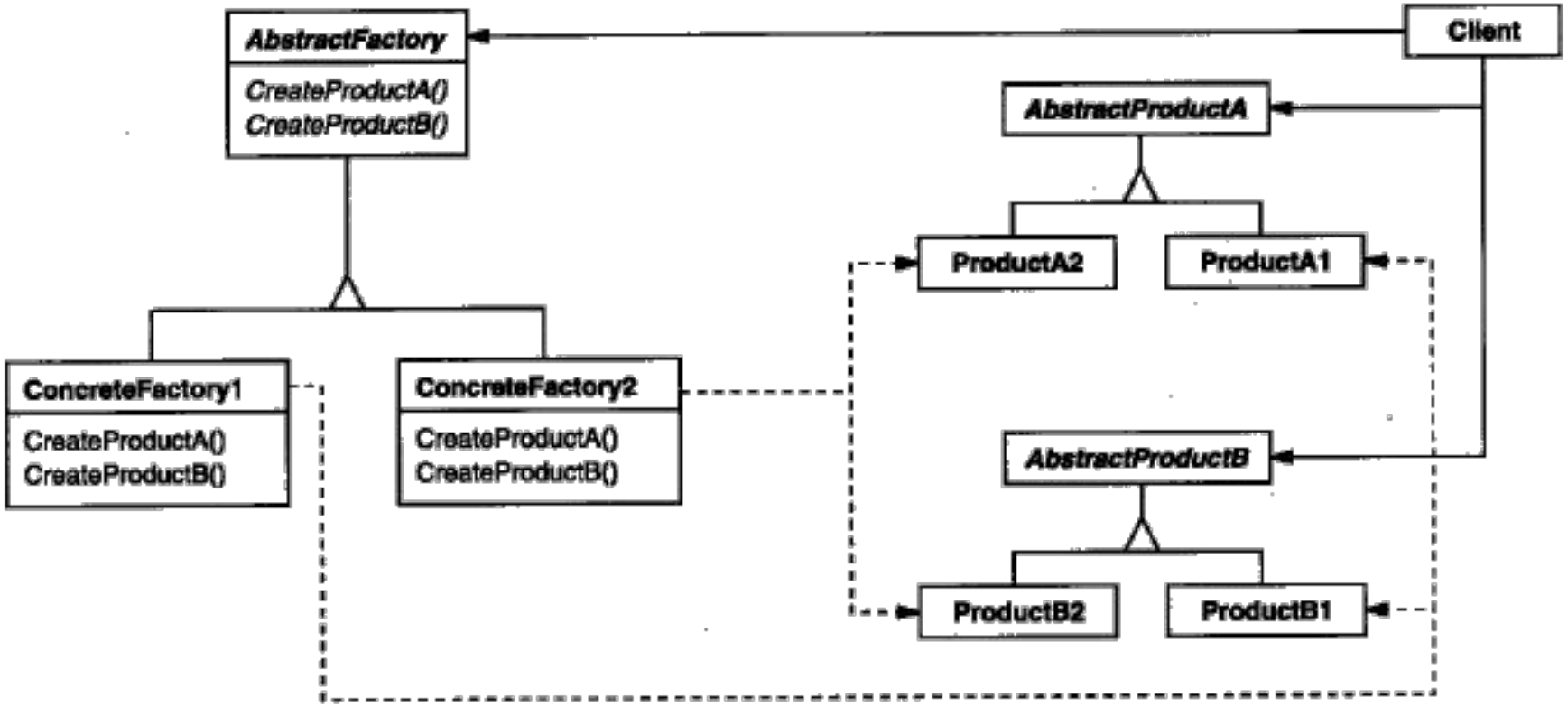

Database
Search()
Sort()
Strategy
*
Strategy
Sort()
BubbleSort
Sort()
QuickSort
Sort()
MergeSort
Sort()

设计模式的类型

- 创建型模式
  - 创建型模式描述了实例化对象的相关技术，解决了与创建对象有关的问题。
  - 创建型模式使用继承来改变被实例化的类，而一个对象创建型模式将实例化委托给另一个对象。

- 典型的模式
  - 工厂方法（*Factory Method*）、**抽象工厂（*Abstract Factory*）**
  - 生成器（*Builder*）、原型（*Prototype*）、单件（*Singleton*）

Abstract Factory



– 抽象工厂模式是用于封装具体的平台，从而使应用程序可以在不同的平台上运行。

设计模式的风险

- 设计模式不是万能的
  - 模式可以解决大多数问题，但不可能解决遇到的所有问题
  - 应用一种模式一般会“有得有失”，切记不可盲目应用
  - 滥用设计模式可能会造成过度设计，反而得不偿失

- 设计模式是有难度和风险的
  - 一个好的设计模式是众多优秀软件设计师集体智慧的结晶
  - 在设计过程中引入模式的成本是很高的
  - 设计模式只适合于经验丰富的开发人员使用

谢谢