软件工程

第6章 设计工程

@第6章.教材

设计工程

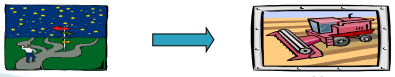
- 软件设计工程概述
- 软件设计的原则
- 软件设计的质量
- 软件设计的复用

设计模型是问题的解决方案，与平台有关（PSM）

表 6－1 分析模型与设计模型的不同侧重点

分析模型	设计模型
面向问题空间	面向解空间
平台无关模型(PIM)	平台有关模型(PSM)
偏重于软件整体的外向型行为的刻画	包含更多的内向型结构细节
模型要素是功能性需求的反映	模型要素顾及非功能性的要求，即质量要求
比较简单	比较复杂

从分析模型逐步演化成设计模型



分析模型 VS. 设计模型

Analysis Model	Design Model
▪ Focus on understanding the problem ▪ Idealized design ▪ Behavior ▪ System structure ▪ Functional requirements ▪ A small model	▪ Focus on understanding the solution ▪ Operations and attributes ▪ Performance ▪ Close to real code ▪ Object lifecycles ▪ Nonfunctional requirements ▪ A large model

软件设计被分为两个阶段：

- **架构设计**

  又称概要设计，它定义了软件的全貌，记录了最重要的设计决策，并成为随后的设计与实现工作的战略指导原则。

  架构设计(*architecture design*)的内容：

  选择软件质量属性的设计策略
  确定合适的架构风格和设计模式
  定义软件的主要结构元素——模块
  接口设计

  目标：使得软件系统在架构层面的设计上满足拟建系统功能性和非功能性需求。

- **详细设计**

  又称构件级设计，它在软件架构的基础上定义各模块的内部细节，例如内部的数据结构、算法和控制流等，其所做的设计决策常常只影响单个模块的实现。

1）软件质量属性的设计策略：

架构因素分析与设计就是分析拟建系统需求，识别影响系统的质量属性，并设计应对措施。

1、可用性设计

系统可用的最小时间间隔、可用时间比例以及故障修复时间对系统可用性的度量。

（1）错误检测，如心跳、异常、命令/响应
（2）错误恢复，如主动/被动/闲置备份、检查点/回滚
（3）错误预防，如事务、进程监视、从服务中删除

2、可维护（修改）性设计

系统变更的难易程度，最小化系统变更的成本。

（1）局部化修改，语义内聚、预判变更、限制变更选项、泛化模块、抽取公共服务等
（2）防止连漪效应，信息隐藏、维持现有接口、限制通信路径、使用中介等
（3）推迟绑定，运行时注册、使用配置文件、多态、构件替代、遵守通信协议

3、性能设计

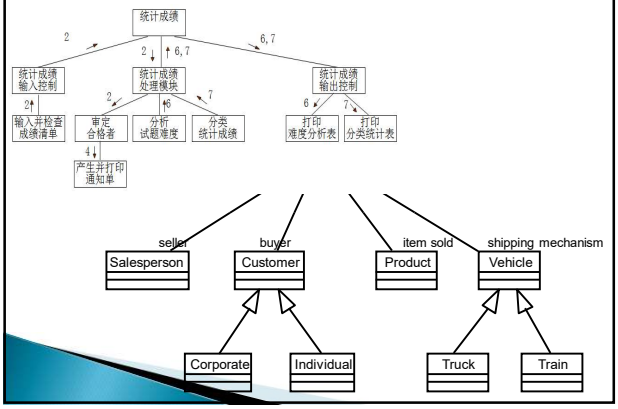
（1）资源需求，如提高计算效率、减少计算开销、控制采样频率、限制队列大小
（2）资源管理，如引入并发、增加可用资源、维持数据或计算的多个副本
（3）资源仲裁，如先进先出、优先级调度

4、安全性设计
系统向合法用户提供服务的同时，抵御非授权使用的能力。
（1）抵御攻击，如用户认证与授权、数据加密、保持数据完整性、限制暴露和访问途径；
（2）检测攻击，如入侵检测等；
（3）从攻击中恢复，恢复系统状态、攻击者身份识别。
5、可测性设计
系统通过测试发现软件缺陷的难易程度。
（1）提供输入/捕获输出，如录制与回放、将接口与实现分离、特化访问路由/接口等；
（2）内部监视，如开机自动检测、周期性检测等。
6、易用性设计
用户期待系统完成期待任务的难易程度和对用户的支持度。
（1）运行时策略，如为用户提供适当的反馈和协助、提供“取消”、“撤消”等命令，建立用户模型、任务模型和系统模型充分了解用户、任务和系统特性；
（2）设计时策略，将用户接口/界面与系统其余部分分离，降低耦合性等。

2）确定合适的软件架构风格

- 数据流风格 *(Dataflow)*：批处理序列、管道－过滤器风格 *(Pipe-and-Filter)*
- 调用/返回风格：主程序/子程序、面向对象风格 *(ADT)*、层次系统 *(Layered Systems)*
- 独立构件风格：进程通信、事件系统
- 虚拟机风格：解释器、基于规则的系统
- 仓库风格：数据库系统、超文本系统、黑板系统

3）定义软件的主要结构元素——模块



4）接口设计

- 接口设计*(interface design)*描述了软件和协作系统之间、软件和使用人员之间是如何通信的。
- 接口就意味着信息流(如数据流、控制流)和特定的行为类型。
- 包括三个方面：
  - 软件模块间的内部接口
  - 模块和协作系统(如外部软件系统、外部设备、网络等)之间的外部接口
  - 使用人员和软件的接口(用户界面)

5）详细设计

- 将软件架构的结构性元素变换为对软件模块的描述。
  - 为所有数据对象定义详细的数据结构
  - 为所有在模块内发生的处理定义算法细节、控制流和数据流

设计工程

- 软件设计工程概述
- 软件设计的原则
- 软件设计的质量
- 软件设计的复用

1、抽象 abstraction

- **抽象：问题的共性，忽略差异。**

  软件开发过程就是对软件抽象层次的一次次细化的过程。

  （上层是下层的**抽象**，下层是上层的**分解**）

- **主要抽象手段：**
  - **数据抽象**

    把一个数据对象的定义抽象为一个数据类型名，用此类型名可定义多个具有相同性质的数据对象
  - **过程抽象**

    把完成一个特定功能的动作序列抽象为一个过程名和参数表，以后通过指定过程名和实际参数调用此过程
  - **对象抽象**

    通过操作和属性，组合了这两种抽象，即在抽象数据类型的定义中加入一组操作的定义，以确定在此类数据对象上可以进行的操作。

2、分解和模块化

- **分解**（decomposition）：控制复杂性的另一种有效方法，软件设计用分解来实现模块化设计。
- **模块化**（modularity）：将一个复杂的软件系统自顶向下地分解成若干模块（module），每个模块完成一个软件的特性，所有的模块组装起来，成为一个整体，完成整个系统所要求的特性。

  模块是能够单独命名并独立地完成一定功能的程序语句的集合，例如，过程、函数、子程序、宏、类等
- **模块具有两个基本的特征：**
  - 外部特征是指模块跟外部环境联系的接口和模块的功能；
  - 内部特征是指模块的内部环境具有的特点，即该模块的局部数据和处理逻辑。

分解 decomposition

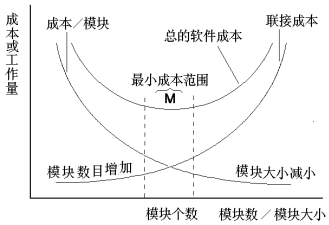
$$C(P1+P2) > C(P1) + C(P2)$$

$$E(P1+P2) > E(P1) + E(P2)$$

- C为问题的复杂度，E为解题需要的工作量
- **思考：**
  - 如果我们无限制地划分软件，开发它所需的工作量会变得小到可以忽略？！
  - 事实上，影响软件开发的工作量的因素还有很多，例如模块接口费用等等
  - 上述不等式只能说明，当模块的总数增加时，单独开发各个子模块的工作量之和会有所减少

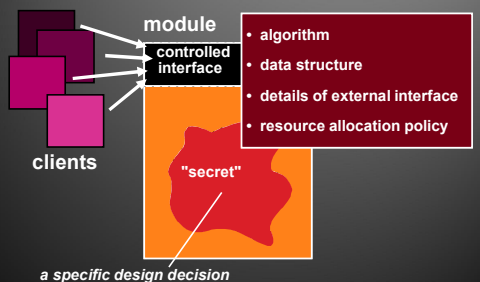
模块化的成本

- **如果模块是相互独立的，当模块变得越小，每个模块花费的工作量越低；**
- **但当模块数增加时，模块间的联系也随之增加，把这些模块联接起来的工作量也随之增加。**



3、信息隐藏 Information Hiding

- 每个模块的实现细节对于其它模块来说应该是隐蔽的



4、模块独立性

- **模块独立的定义：**
  - 每个模块完成一个相对独立的子功能，并且和其他模块之间的关系很简单。
  - 模块的独立性是模块化追求的目标。
- **模块独立的重要性：**
  - 软件质量的关键：

    (1) 模块化程度较高的软件容易开发；

    (2) 模块化程度较高的软件也比较容易测试和维护。

模块独立的衡量指标

- 模块独立可以由两项指标来衡量：
  - **内聚**（cohesion），是一个模块内部各个元素彼此结合的紧密程度的度量
  - **耦合**（coupling），是模块之间的相对独立性（互相连接的紧密程度）的度量
- 模块独立追求高内聚和低耦合。

内聚

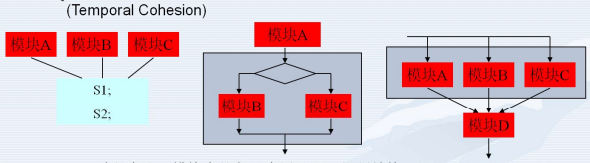
- 一般模块的内聚性分为七种类型：
  1. 偶然内聚 coincidental cohesion
  2. 逻辑内聚 logical cohesion
  3. 时间内聚 temporal cohesion
  4. 过程内聚 procedural cohesion
  5. 通信内聚 communicational cohesion
  6. 顺序内聚 sequential cohesion
  7. 功能内聚 functional cohesion

低 ↓ 高

- 确定内聚的精确级别是不必要的，重要的该是尽量争取高内聚和识别低内聚

内聚(Cohesion)：一个模块内各个元素彼此结合的紧密程度。

低内聚
- 偶然内聚：一个模块内的元素之间的无关系很松散。（公共语句）。(Coincidental Cohesion)
- 逻辑内聚：若干个逻辑功能类似的任务组成一个模块。(Logical Cohesion)
- 时间内聚：若干个任务必须在同一段时间内执行。如初始化工作。(Temporal Cohesion)



中内聚
- 过程内聚：模块内的各元素属于同一控制结构。(Procedural Cohesion)
- 通信内聚：模块中所有元素都使用同一个输入数据，和/或产生同一个输出数据。(Communicational Cohesion)

高内聚
- 顺序内聚：模块中所有处理元素和同一个功能密切相关，且这些处理必须顺序执行。(Sequential Cohesion)
- 功能内聚：所有处理元素属于一个整体，完成一个单一的功能。(Functional Cohesion)

耦合

- 一般模块之间可能的耦合方式有七种类型
  1. 非直接耦合 no direct coupling
  2. 数据耦合 data coupling
  3. 标记（特征）耦合 stamp coupling
  4. 控制耦合 control coupling
  5. 外部耦合 external coupling
  6. 公共耦合 common coupling
  7. 内容耦合 content coupling

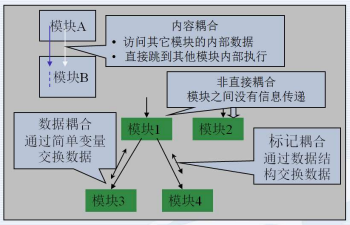
低 ↓ 高

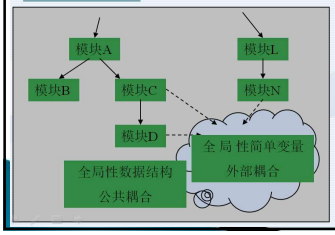
- 确定耦合的精确级别是不必要的，重要的该是尽量争取低耦合和识别高耦合

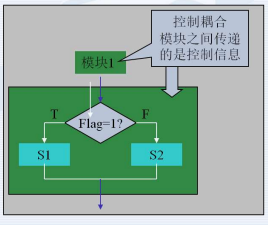
2 软件设计的概念和原理 ---- 耦合

1. 非直接耦合
2. 数据耦合
3. 标记耦合
4. 控制耦合
5. 外部耦合
6. 公共耦合
7. 内容耦合

⇨ 弱耦合 / 中耦合 / 较强耦合 / 强耦合







设计工程

- 软件设计工程概述
- 软件设计的原则
- 软件设计的质量
- 软件设计的复用

1、软件设计的质量要求

（1）设计应当模块化，高内聚、低耦合。
- 支持多人合作开发
- 易于测试和修改
- 能够以演化过程实现

（2）设计应当包含数据、体系结构、接口和构件的清楚的表示。

（3）设计应根据软件需求采用可重复使用的方法进行。

（4）应使用能够有效传达其意义的表示法来表达设计模型。

2、7种软件设计的坏味道

1）僵化性（*Rigidity*）
- 很难对软件进行改动，因为每个改动都会迫使对系统其他部分的许多改动

2）脆弱性（*Fragility*）
- 对系统的改动会导致系统中和改动的地方在概念上无关的许多地方出现问题

3）牢固性（*Immobility*）
- 很难解开系统中某部分与其它部分之间的纠结，从而难以使其中的任何部分可以被分离出来被其它系统复用

4）粘滞性（*Viscosity*）
- 做正确的事情要比做错误的事情困难。表现为两种形式：
  - 软件粘滞性
    - 需要对软件进行修改时，可能存在多种方法。有的方法可以保持原有的设计质量，另一些方法则会破坏原有的设计质量。如果，破坏软件质量的修改比保持原有设计质量的修改更容易实施时，我们就称该软件具有“软件粘滞性”。
  - 环境粘滞性
    - 当开发环境迟钝、低效时，就会产生环境粘滞性。
    - 例如：如果编译时间很长，那么开发人员可能会放弃那些能保持设计质量，但是却需要导致大规模重新编译的改动。

5）不必要的复杂性（*Needless Complexity*）
- 设计中包含不具有任何好处的基础结构。

6）不必要的重复（*Needless Repetition*）
- 设计中包含一些重复的结构，这些结构本来可以通过单一的抽象进行统一
  - 使用Cut/Copy/Paste实施源代码级的软件复用容易导致这一问题
  - 这种代码级别的冗余，将带来修改上的问题

7）晦涩性（*Opacity*）
- 很难阅读和理解，不要相信你永远都会如此清楚的了解你的每一行代码，“时间会冲淡一切”。要站在阅读者的角度进行设计

设计工程

- 软件设计工程概述
- 软件设计的原则
- 软件设计的质量
- 软件设计的复用

什么是模式(Pattern)

- *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”，*1977*年。
- 定义：“在一个上下文中对一种问题的解决方案”。

模式可重用！

设计重用 Design Reuse

- Pattern
  - Architectural style
  - Design pattern
  - Idiom (成例)，成例有时称为代码模式(Coding Pattern)。
- Framework

软件模式分类

- An architectural pattern (architectural style) expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines （指南） for organizing the relationships between them.
- A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring（常见） structure of communicating components （组件） that solves a general design problem within a particular context（上下文）.
- An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

常见的架构风格

- **通用结构**: *Organize components*
  - Layers: Organize components into layers where layer's services are only used by layer i+1.
  - Pipes and Filters: Divide the task into several sequential processing steps -- the output of task i is the input of task i+1.
  - Blackboard: Several independent programs work cooperatively on a common data structure.
- **分布式系统**: *Handle distributed computation.*
  - Broker（代理）: Introduce a *broker* component to achieve better decoupling（降低耦合） of clients and servers -- brokers accept requests from clients and forward the requests to servers, then return the results back to the clients.
  - Client/Server
  - 3 Tiers（三层架构）

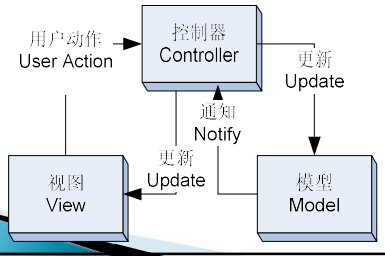
常见的架构风格（续）

- **交互式系统**: Keep a program's functional core independent of the user interface
  - Model – View – Controller: Divides the application into processing, output, and input. View and controller parts are usually observers of the model via the observer pattern
  - Presentation – Abstract – Control: Divides the application up to heirarchies or MVC-like components. Each component is dependent upon and provides functionality for the a higher-level component. There is only one top-level component
- **自适应系统**: Design for change
  - Microkernel （微内核） Encapsulate（封装） the fundamental services of the application
  - Reflection（反射） Divide the application into a meta-level and a base level to make the application "self-aware"（自意识）. The meta level encapsulates knowledge of the system; the base level encapsulates knowledge about the problem domain

Others: 批处理、解释器、进程控制、基于规则

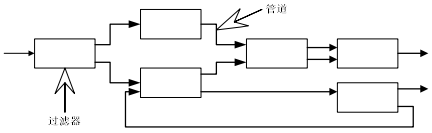
1、MVC

模型Model：管理系统中存储的数据和业务规则，并执行相应的计算功能。
视图View：根据模型生成提供给用户的交互界面，不同的视图可以对相同的数据产生不同的界面。
控制器Control：接收用户输入，通过调用模型获得响应，并通知视图进行用户界面的更新。



2、管道和过滤器（Pipes and Filters）风格

- In this style, each component has a set of inputs and a set of outputs.
- A component reads streams of data on its inputs and produces streams of data on its output.

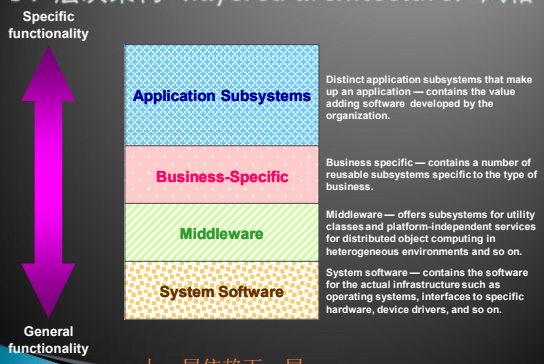


管道和过滤器架构举例

- Linux的Shell程序可以看做是典型的管道与过滤器架构的例子
- 例如下面的Shell脚本：
  - $cat *TestResults* | sort | grep *Good*

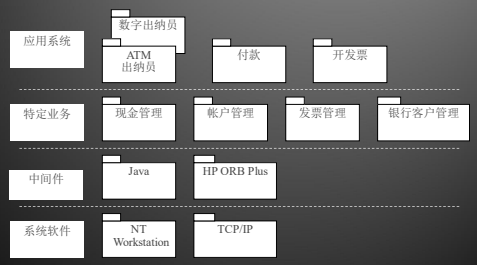
会将TestResults文件的文本进行排序，然后找出其中包含单词Good的行，并显出在屏幕上。Shell命令cat、sort和grep依次执行，就构成了一个管道-过滤器架构。

3、层次架构（layered architecture）风格

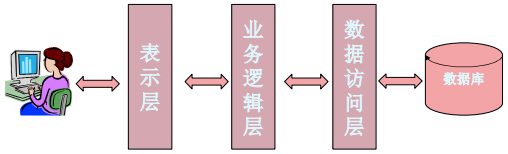


上一层依赖下一层

层次架构实例



4、Three-Tiers三层架构



- **表示层：** 负责向用户呈现界面，并接收用户请求发送给业务逻辑层；
- **业务逻辑层：** 负责执行业务逻辑以处理用户请求，并调用数据访问层提供的持久性操作；
- **数据访问层：** 负责执行数据库持久性操作。

5、黑板（Blackboard）风格

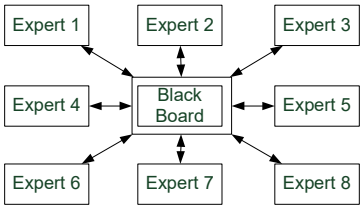
- **黑板架构为参与问题解决的知识源提供了共享的数据表示，这些数据表示是与应用相关的。**

在黑板架构中，控制流是由黑板数据的状态决定的，而并非按照某个固定的顺序执行。黑板架构的结构包括如下部分：

- **知识源：** 是指彼此分离独立的与应用相关的知识包，知识源之间的交互都是通过黑板来完成的，它们彼此并不直接交互。
- **黑板数据结构：** 按照应用相关的层次结构组织而成的问题解决过程中的状态数据。知识源会更新黑板数据，从而增量式地解决问题。
- **控制流：** 完全有黑板状态驱动，知识源会在黑板数据更新时根据其状态做出相应的响应。

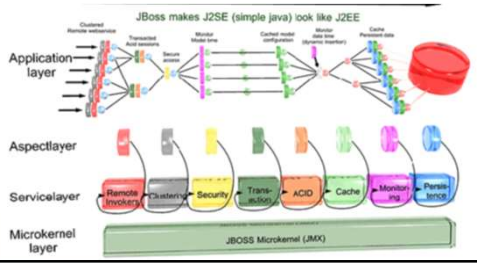
通用结构 – 黑板架构

- 黑板架构专门针对没有确定的解决方法的问题，例如信号处理和模式识别，它通过多个知识源的协作来解决问题，而这种协作完全是状态驱动的，因此各个知识源具有公平的机会获取并更新黑板中的状态数据。



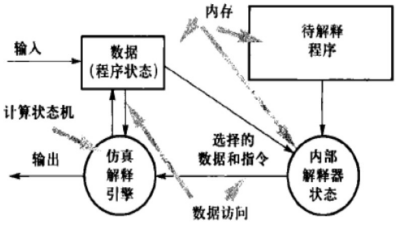
6、微内核风格

- 微内核概念来源与操作系统领域。微内核是提供了操作系统核心功能的内核，它只需占用很小的内存空间即可启动，并向用户提供了标准接口，以使用户能够按照模块化的方式扩展其功能。现在大多数操作系统都采用了微内核架构。



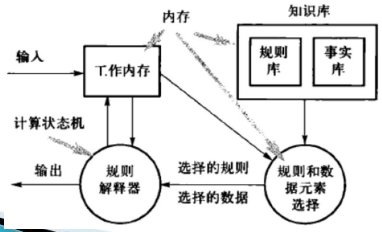
7、解释器架构

- **解释器架构用于仿真当前不具备的计算环境，通常包含四个组成部分**：用来解释伪码程序的解释引擎、包含待解释程序的内存、解释引擎的控制状态，以及被仿真程序的当前状态：



8、基于规则的架构

- **基于规则的架构是一种解释器架构风格，它将人类专家的问题解决知识编码成规则，这些规则在系统执行计算满足指定的条件时被执行或激活，通过规则不断地被执行和激活，最终使得问题被解决**。由于这些规则不能被计算机系统直接执行，因此需要通过解释器来解释它们。

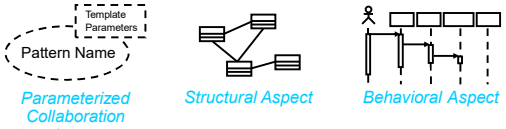


设计模式 Design Patterns

- 架构风格是宏观的设计模式，描述高层组织结构
- 设计模式描述的低成层局部细节，是实现设计复用的有效手段

按作用分类：

创造型（对象创造过程）、结构型、行为型



按作用域分类：

类模式（静态关系）、对象模式（动态关系）

表 6-3 GOF 设计模式分类

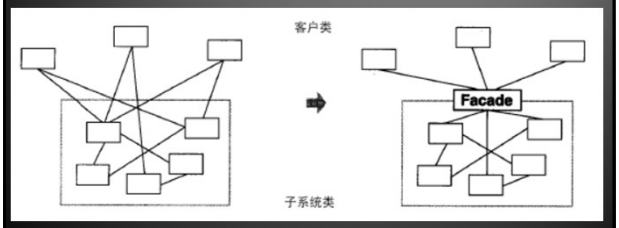
目的 / 作用域	创建型	结构型	行为型
类	工厂方法	适配器	解释器 模板方法
对象	抽象工厂 构建器 原型 单例	适配器 桥接 组合 装饰器 外观 享元 代理	职责链 命令 迭代器 中介器 备忘录 观察者 状态 策略 访问者

GOF 23个设计模式

- 创建型模式
  - ABSTRACT FACTORY (抽象工厂)；BUILDER (生成器)；FACTORY METHOD (工厂方法)；PROTOTYPE (原型)；SINGLETON (单件)
- 结构型模式
  - ADAPTER（适配器）；BRIDGE（桥接）；COMPOSITE（组合）；DECORATOR（装饰）；FACADE（外观）；FLYWEIGHT（享元）；PROXY（代理）
- 行为模式
  - CHAIN OF RESPONSIBILITY (职责链)； COMMAND (命令)；INTERPRETER (解释器)；ITERATOR (迭代器)；MEDIATOR (中介者)；*MEMENTO (备忘录)；OBSERVER (观察者)；STATE (状态)；STRATERY (策略)；TEMPLATE METHOD (模板方法)；VISITOR (访问者)

举例：Facade模式

外观模式为子系统中的一组接口提供了更高一层的统一接口，达到了用简单接口访问复杂子系统，且对子系统分层的目的。

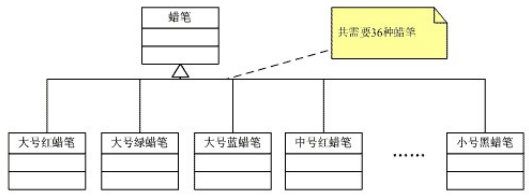


例如，在4S系统中，假设对于用户购车的处理需要调用3个方法：第一个方法需要用户名和密码作为参数，以进行用户验证；第二个方法需要用车辆型号作为参数，以提交购车信息；第三个方法以车辆型号为参数，以获取该车辆最新的售价和促销信息，用于刷新用户显示。对这3个方法的调用会使得代码编写量较多，而且当用户购车的处理逻辑发生变化时，就需要修改这些代码。为了提高系统的可用性和可维护性，此时就可以选择使用外观模式。

在4S系统中，可以新添加一个外观类Façade，它包含一个purchase方法，一次性接收用户名、密码、竞拍物品与竞拍价等信息作为参数，然后依次调用前面提到的3个方法。现在只需要调用Façade类的purchase方法就可以完成以前需要调用3个方法才能完成的对购车请求的处理，从而使得购车处理子系统的接口变得简单了。而且，当竞价处理逻辑发生变化时，使用Façade的代码并不受影响，只需要修改Façade的具体实现即可。

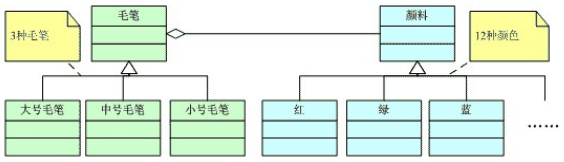
举例： Bridge模式

- 蜡笔和毛笔的故事
  - 我们需要用蜡笔或者毛笔绘制图画，画里有山水、花鸟、人物等；
  - 我们需要
    - 不同粗细的笔（填充颜色用细笔不方便，勾勒细节用粗笔无法完成）
    - 不同的色彩（彩色图画）
  - 36只蜡笔
    - 购置了粗、中、细3套蜡笔，一套12色，一共36只蜡笔



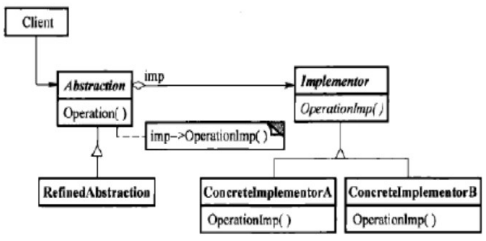
- 蜡笔和毛笔的故事（续）
  - 我们也需要36只毛笔吗？

3只毛笔，12种颜料，不同的毛笔使用不同的颜料



Bridge模式

- **意图：**
  - **将抽象部分与其实现部分分离，在运行时连接起来（不是编译时绑定），使它们可以独立变化。**

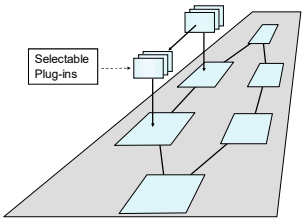


例如，在4S系统中，假设将用户分成了Gold和Silver两种，随着时间的推移，将来可能还会有Bronze和Diamond等更多的种类。另一方面，用户联系方式有E-mail信箱，将来可能还会有即时聊天账户名、手机、邮寄地址和社区网站账户等。这样，User类就有两个可以独立演化的属性，即用户种类Type和联系方式Contact，此时就可以应用桥接模式来设计它们。

对照图6-20，在4S系统中，User对应于Abstraction，RefinedAbstraction对应于GoldUser、SilverUser和其他的用户类型。Contact对应于Implementor，表示用户的联系方式，ConcreteImplementorA和ConcreteImplementorB等就对应于email、IM、mobilephone和SNaccount等具体的联系方式。User拥有一个对Contact对象的引用，该引用可以通过User对象的setContact方法赋值，对应于imp应用，对User的联系方式的访问都是通过调用这个引用的方法实现的，即imp->OperationImpl()。因此，具体的User对象和具体的Contact对象是在运行时连接起来的。而且，User类的扩展和Contact类的扩展彼此间不会造成任何影响。

框架(Framework)

- 框架是一个代码骨架，可以使用为解决问题而设计的特定类或功能来填充这个代码骨架，使之丰满。



Frameworks with Plug-ins

框架举例

1. MVC框架
   - J2EE
   - Struts
2. ORM 框架
   - EJB
   - Hibernate
3. 界面框架
   - Java界面框架：AWT, Swing, SWT, SwingWT, Java 2D, Java 3D，JSF
   - AJAX界面框架：DWR， DOJO
4. 复合框架
   - Spring包含Web MVC, IOC, ...
   - Java EE 包含JSF、EJB 、JAX-WS 、IOC, ...