软件体系结构 100

软件体系结构概述 10

系统需求来自于企业目标，架构来自于系统需求，系统来自于架构。

Mary Shaw 和 David Garlan 认为软件体系结构包括构成系统的设计元素的描述，设计元素的交互，设计元素组合的模式，以及在这些模式中的约束

软件体系结构包括构件(Component)、连接件(Connector)和约束(Constraint)或配置(Configuration)三大要素。

国内普遍接受的定义：软件体系结构包括构件、连接件和约束，它是可预制和可重构的软件框架结构。

构件是可预制和可重用的软件部件，是组成体系结构的基本计算单元或数据存储单元,可以是一个处理过程或数据元素。

连接件也是可预制和可重用的软件部件，是构件之间的连接单元

构件和连接件之间的关系用约束来描述

软件体系结构 = 构件 + 连接件 + 约束 Component+Connector+Constraint

软件体系结构优势:
容易理解、重用、控制成本、可分析性

软件体系结构风格定义和常见的体系结构风格 14

An architectural style defines a family of systems in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of components and connector types, and a set of constraints on how they can be combined.

体系结构风格定义了一个系统家族，即一个体系结构定义一个词汇表和一组约束。词汇表中包含一些构件和连接件类型，而这组约束指出系统是如何将这些构件和连接件组合起来的。

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。
数据流风格：批处理序列；管道/过滤器。

调用/返回风格：主程序/子程序；面向对象风格；层次结构。
独立构件风格：进程通讯；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。
过程控制环路
C/S 风格
B/S 风格

管道和过滤器:

每个构件都有一组输入和输出，构件读输入的数据流，经过内部处理，然后产生输出数据流。过滤器风格的连接件就象是数据流传输的管道，将一个过滤器的输出传到另一个过滤器的输入。
不变量：
过滤器虽然可以增量式地处理数据，但是它们是独立的
管道和过滤器的正确输出不依赖其顺序
实例：
编译器，功能程序，并行程序

数据抽象和面向对象组织

数据的表示方法和它们的相应操作被封装在一个抽象数据类型或对象中
这种风格的构件是对象或者说是抽象数据类型的实例
对象通过函数和过程的调用来进行交互

基于事件的隐式调用

构件不直接调用一个过程，而是触发或广播一个或多个事件
系统中的其他构件中的过程在一个或多个事件中注册，当一个事件被触发，系统自动调用在这个事件中注册的所有过程。
这种风格的构件是一个模块，这些模块可以是一些过程，又可以是一些事件的集合。
不变量：事件的触发者并不知道哪些构件会被这些事件影响（观察者模式-Observer）

实例：数据库管理系统，用户界面

分层系统

组织成一个层次结构
每一层都为上一层提供了相应的服务，并且接受下一层提供的服务
在分层系统的一些层次中构件实现了虚拟机的功能
实例：分层的通信协议

仓库系统

构件：中心数据结构（仓库）和一些独立构件的集合
仓库和在系统中很重要的外部构件之间的相互作用
实例：需要使用一些复杂表征的信号处理系统

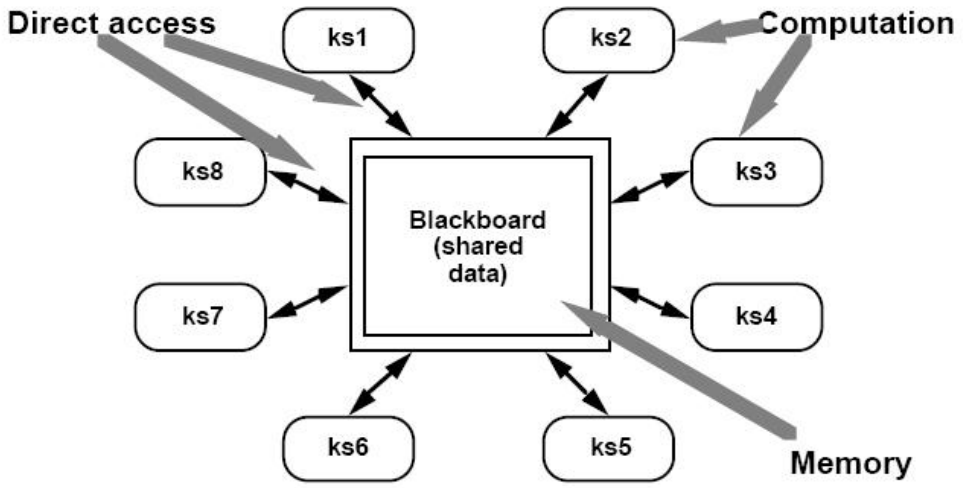


Figure 4: The Blackboard

过程控制环路

源自于控制理论中的模型框架，将事务处理看成输入、加工、输出、反馈、再输入的一个持续的过程模型。
通过持续性的加工处理过程将输入数据转换成既定属性的“产品”，在工控系统、供电、水利甚至可以推广到商务软件体现的管理模型中。

C/S 风格

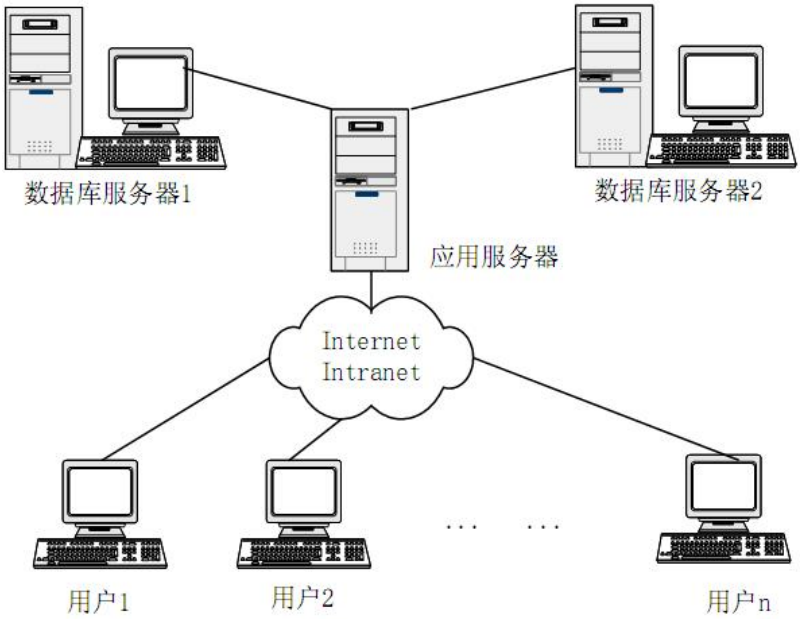
C/S 软件体系结构是基于资源不对等，且为实现共享而提出来的，是 20 世纪 90 年代成熟起来的技术，C/S 体系结构定义了工作站如何与服务器相连，以实现数据和应用分布到多个处理机上。
C/S 体系结构有三个主要组成部分：数据库服务器、客户应用程序和网络。
任务分配:
服务器
数据库安全性的要求；
数据库访问并发性的控制；
数据库前端的客户应用程序的全局数据完整性规则；
数据库的备份和恢复。

客户应用程序
提供用户与数据库交互的界面；
向数据库服务器提交用户请求并接收来自数据库服务器的信息；
利用客户应用程序对存在于客户端的数据执行应用逻辑要求

优点:
C/S 体系结构具有强大的数据操作和事务处理能力， 模型思想简单，易于人们理解和接受。
系统的客户应用程序和服务器构件分别运行在不同的计算机上，系统中每台服务器都可以适合各构件的要求，这对于硬件和软件的变化显示出极大的适应性和灵活性，而且易于对系统进行扩充和缩小。
在 C/S 体系结构中，系统中的功能构件充分隔离，客户应用程序的开发集中于数据的显示和分析，而数据库服务器的开发则集中于数据的管理，不必在每一个新的应用程序中都要对一个 DBMS 进行编码。将大的应用处理任务分布到许多通过网络连接的低成本计算机上，以节约大量费用。

缺点:
开发成本较高
客户端程序设计复杂
信息内容和形式单一
用户界面风格不一，使用繁杂，不利于推广使用
软件移植困难
软件维护和升级困难
新技术不能轻易应用

三层 CS 风格:



优点:

允许合理地划分三层结构的功能，使之在逻辑上保持相对独立性，能提高系统和软件的可维护性和可扩展性。

允许更灵活有效地选用相应的平台和硬件系统，使之在处理负荷能力上与处理特性上分别适应于结构清晰的三层；并且这些平台和各个组成部分可以具有良好的可升级性和开放性。

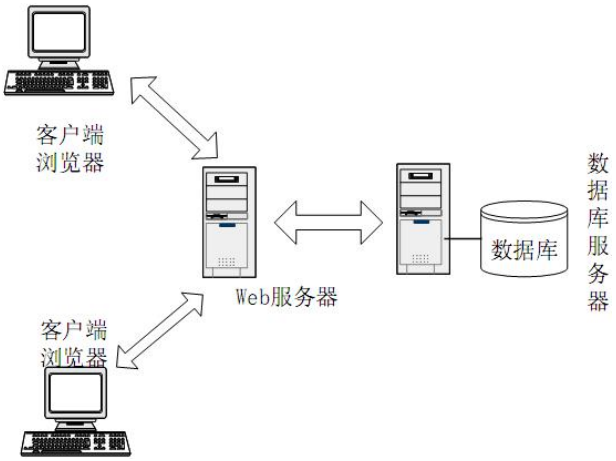
应用的各层可以并行开发，可以选择各自最适合的开发语言。

利用功能层有效地隔离开表示层与数据层，未授权的用户难以绕过功能层而利用数据库工具或黑客手段去非法地访问数据层，为严格的安全管理奠定了坚实的基础。

缺点:
三层 C/S 结构各层间的通信效率不高，即使分配给各层的硬件能力很强，其作为整体来说也达不到所要求的性能。
设计时必须慎重考虑三层间的通信方法、通信频率及数据量，这和提高各层的独立性一样是三层 C/S 结构的关键问题。

B/S 风格:

浏览器/服务器(B/S)风格就是上述三层应用结构的一种实现方式，其具体结构为：浏览器/Web 服务器/数据库服务器。
B/S 体系结构主要是利用不断成熟的 WWW 浏览器技术，结合浏览器的多种脚本语言，用通用浏览器就实现了原来需要复杂的专用软件才能实现的强大功能，并节约了开发成本。从某种程度上来说，B/S 结构是一种全新的软件体系结构。



优点:
基于 B/S 体系结构的软件，系统安装、修改和维护全在服务器端解决。用户在使用系统时，仅仅需要一个浏览器就可运行全部的模块，真正达到了“零客户端”的功能，很容易在运行时自动升级。
B/S 体系结构还提供了异种机、异种网、异种应用服务器的联机、联网、统一服务的最现实的开放性基础。

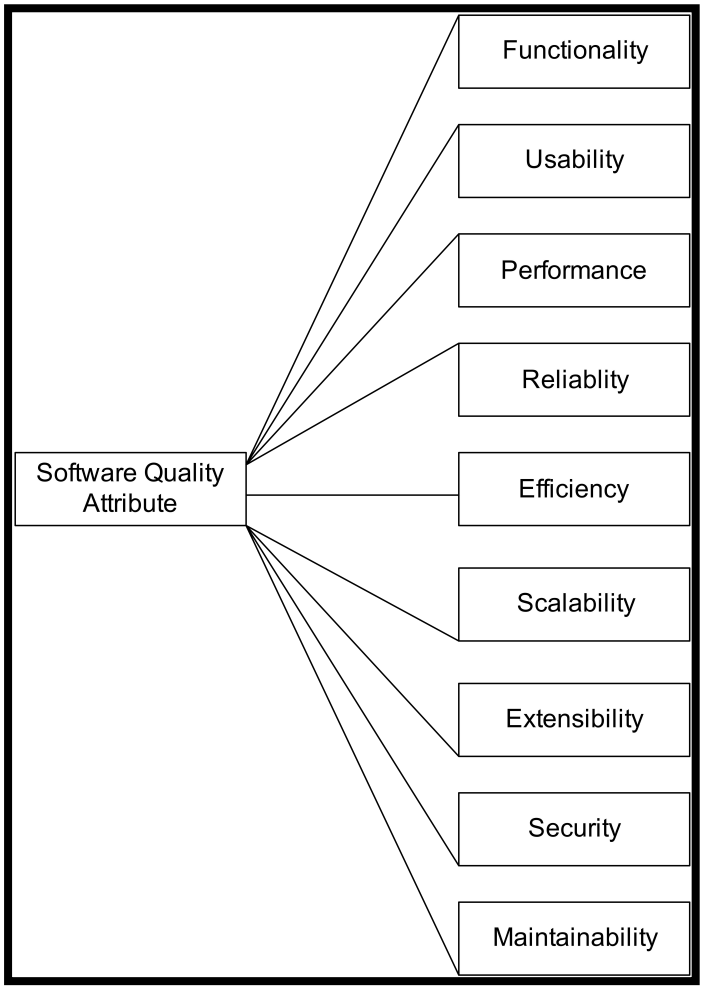
缺点:
B/S 体系结构缺乏对动态页面的支持能力，没有集成有效的数据库处理功能。
B/S 体系结构的系统扩展能力差，安全性难以控制。
采用 B/S 体系结构的应用系统，在数据查询等响应速度上，要远远低于 C/S 体系结构。
B/S 体系结构的数据提交一般以页面为单位，数据的动态交互性不强，不利于在线事务处理(OLTP)应用。

软件质量属性 6

高质量的软件符合商业目标和用户需求

它具有正确的功能和优良的属性

质量属性需求来源于商业和产品目标。
关键的质量属性必须刻画系统的细节特征。
质量属性场景是用于描述质量属性和表达项目干系人观点的强有力的工具
几个重要的质量属性:



外部质量

外部质量对于用户而言是可见的

包括正确性、健壮性、可靠性、性能、安全性、易用性、兼容性等。

正确性
正确性是指软件按照需求正确执行任务的能力。
健壮性
健壮性是指在异常情况下，软件能够正常运行的能力。

可靠性是指在一定的环境下，在给定的时间内，系统不发生故障（可以正常运行）的概率。

性能通常是指软件的“时间-空间”效率，而不仅是指软件的运行速度。

安全性是指防止系统被非法入侵的能力，既属于技术问题又属于管理问题

易用性是指用户使用软件的容易程度。

兼容性是指不同产品（或者新老产品）相互交换信息的能力。又称互操作性.

究竟什么样的安全性是令人满意的呢？一般地，如果黑客为非法入侵花费的代价（考虑时间、费用、风险等因素）高于得到的好处，那么这样的系统可以认为是安全的。对于普通软件，并不一点要追求很高的安全性，也不能完全忽视安全性，要先分析黑客行为。

内部质量

内部质量只有开发人员关心
它们可以帮助开发人员实现外部质量
包括易理解性、可测试性、可维护性、可扩展性、可移植性、可复用性等 6

易理解性是易被开发人员理解软件产品的能力，意味着所有的工作成果要易读、易理解，可以提高团队开发效率，降低维护代价。

可测试性指的是测试软件组件或集成产品时查找缺陷的简易程度，又称为可验证性

可维护性表明了在软件中纠正一个缺陷或做一次更改的简易程度。

可扩展性反映软件适应“变化”的能力。

可移植性指的是软件不经修改或稍加修改就可以运行于不同软硬件环境（CPU、OS 和编译器）的能力，主要体现为代码的可移植性。

可复用性是指一个软件的组成部分可以在同一个项目的不同地方甚至在不同的项目中重复

使用的能力。

过程质量

过程质量与开发活动相关
产品通过过程来进行开发
如开发效率，时间控制等

如果想保持一如既往的开发高质量的产品，过程必须是可靠的

如果想适应无法预计的工具或环境改变，过程必须是稳健的

过程的执行必须是高效的

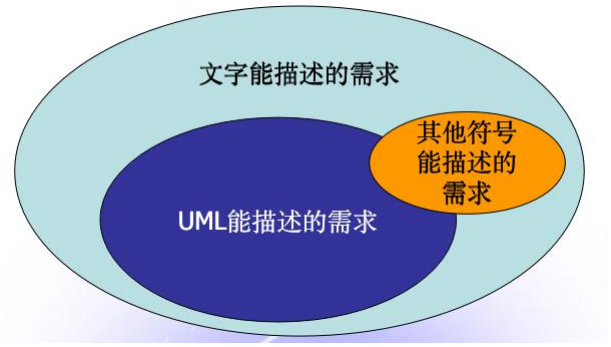
如果想适应新的管理方式或组织形式，过程必须是可扩展的

如果想跨项目和组织来使用，过程必须是可重用的

外部	正确性、健壮性、可靠性、性能、安全性、易用性、兼容性
内部	易理解性、可测试性、可维护性、可扩展性、可移植性、可复用性
过程	可靠的、稳健的、高效的、可扩展的、可重用的

UML 20

UML 是一种语言，它有属于自己的标准表达规则。它不是一种类似 Java、C++的编程语言，而是一种分析设计语言，也就是一种建模语言。
UML 是由图形符号表达的建模语言。
UML 是一种用于描绘软件蓝图的标准语言。



UML 的结构
模型元素

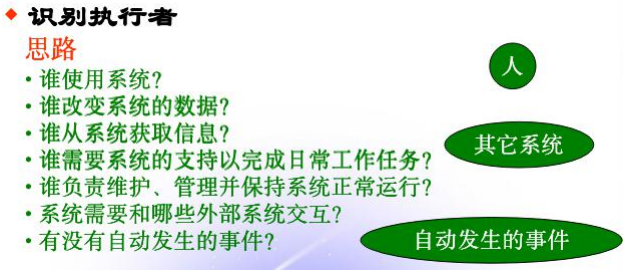
通用机制(General mechanism)
额外的注释、修饰和语义等
包括规格说明、修饰、公共分类和扩展机制四种
允许用户对 UML 进行扩展

特点:
工程化
规范化
可视化
系统化
文档化
智能化

用例图

用例建模步骤
识别执行者
识别用例
绘制用例图
书写用例文档
检查用例模型

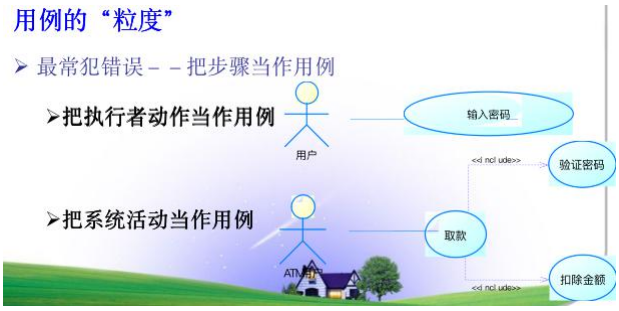
执行者——Actor(小人
定义：在系统之外，透过系统边界与系统进行有意义交互的任何事物。
引入执行者的目的：帮助确定系统边界。



用例(椭圆
用例是在系统中执行的一系列动作，这些动作将生成特定执行者可见的价值结果。一个用例定义一组用例实例。

用例要点:
有意义的目标
价值结果由系统生成
业务语言，用户观点
注意用例的命名(动宾结构

用例的“粒度”



【执行者】使用系统来【用例】
关系:

在用例图中，执行者和用例之间进行交互，相互之间的关系用一根直线来表示，称为关联关系(Association)或通信关系(Communication)。

执行者之间可以有泛化(Generalization)关系（或称为“继承”关系）

包含关系
描述在多个用例中都有的公共行为，由用例 A 指向用例 B，表示用例 A 中使用了用例 B 中的行为或功能，包含关系是通过在依赖关系上应用<<include>>构造型（衍型）来表示的。

扩展关系
扩展用例可以在基用例之上添加新的行为，但是基用例必须声明某些特定的“扩展点”，并且扩展用例只能在这些扩展点上扩展新的行为。
在扩展（extend）关系中，基础用例(Base)中定义有一至多个已命名的扩展点，扩展关系是指将扩展用例(Extension)的事件流在一定的条件下按照相应的扩展点插入到基础用例(Base)中。
扩展关系是通过在依赖关系上应用<<extend>>构造型（衍型）来表示的。

a 包含 b
a 拓展到 b
都是使用箭头从 a 指到 b

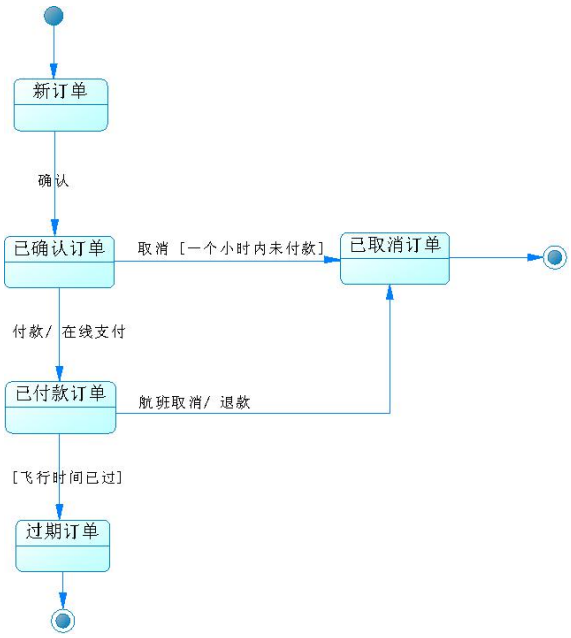
泛化关系(用例之间,略

用例是文本文档，而非图形
用例建模主要是编写文本的活动，而非制图

检查用例模型
用例模型完成之后，需要对用例模型进行检查，看看是否有遗漏或错误之处。主要可以从以下几个方面来进行检查：
功能需求的完备性
模型是否易于理解
是否存在不一致性
避免语义二义性

状态图:

状态图(State Diagram)用来描述一个特定对象的所有可能状态及其引起状态转移的事件。
通常用状态图来描述单个对象的行为，它确定了由事件序列引出的状态序列，但并不是所有的类都需要使用状态图来描述它的行为，只有那些具有重要交互行为的类，才会使用状态图来描述。
一个状态图包括一系列对象的状态及状态之间的转换。



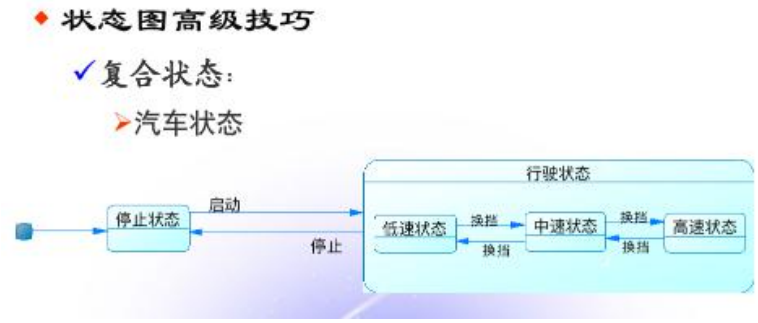
状态图用初始状态（Initial State）表示对象创建时的状态，每一个状态图一般只有一个初始状态，用实心的圆点表示。
每一个状态图可能有多个终止状态（Final State），用一个实心圆外加一个圆圈表示。

状态图中可有多个状态框，每个状态框中有两格：上格放置状态名称，下格说明处于该状态时，系统或对象要进行的活动（Action）

从一个状态到另一个状态之间的连线称为转移（Transition）。状态之间的过渡事件（Event）对应对象的动作或活动（Action）。事件有可能在特定的条件下发生，在 UML 中这样的条件称为守护条件（Guard Condition），发生的事件可通过对象的动作（Action）进行处理。状态之间的转移可带有标注，由三部分组成（每一部分都可省略），其语法为：事件名［条

件] / 动作名(在什么条件下,发生什么事件引发了一个什么 action

状态图组成元素:
初始状态、终止状态、状态、转移、守护条件、事件、动作



使用同步条:

活动图:

活动图(Activity Diagram)用来表示系统中各种活动的次序，它的应用非常广泛，既可用来描述用例的工作流程，也可以用来描述类中某个方法的操作行为。

活动图是 UML 中的流程图，它是事件流的另一种建模方式。

活动图是一种描述工作流的方式，它用来描述采取何种动作、做什么（对象状态改变）、何时发生（动作序列）以及在何处发生（泳道）。

活动图作用
描述业务流程
描述用例路径
描述方法执行流程（程序流程图）

活动图组成元素

活动图由起始活动(Start Activity)、终止活动(End Activity)、活动(Activity)、转移(Transition)或流(Flow)、决策(Decision)、守护条件(Condition)、同步条(Synchronization)和泳道(Swimlane)等组成。

起始活动显式地表示活动图工作流程的开始，用实心圆饼来表示，在一个活动图中，只有一个起始活动。

终止活动表示一个活动图的最后和终结活动，一个活动图中可以有 0 个或多个终止活动，终止活动用实心圆点外加一个小圆圈来表示。

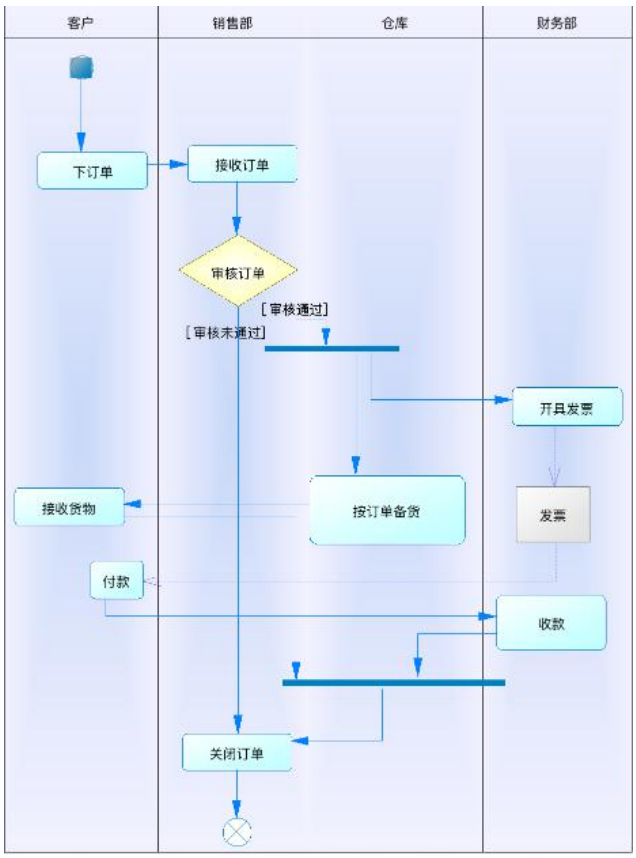
活动图中的活动用一个圆角矩形表示.

守护条件用来约束转移，守护条件为真时转移才可以开始。

用菱形符号来表示判定，判定符号可以有一个或多个进入转移，两个或更多的带有守护条件的发出转移

一条粗黑线表示将转移分解成多个分支(fork)，同样用粗黑线来表示分支的合并(join)，这种粗黑线称为同步条。

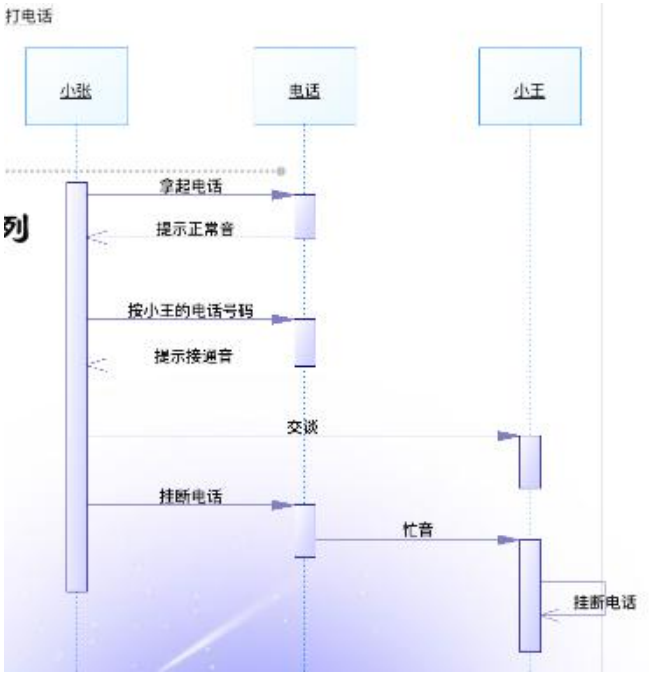
泳道用于划分活动图，有助于更好地理解执行活动的场所。泳道划分负责活动的对象，明确地表示哪些活动是由哪些对象进行的，每个活动只能明确地属于一个泳道。



顺序图:

UML 顺序图一般用于确认和丰富一个使用情境的逻辑。

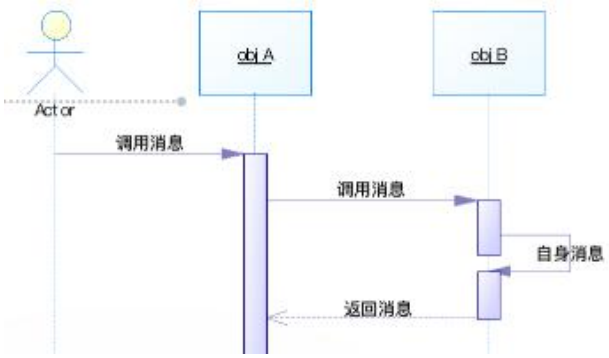
顺序图将交互关系表现为一个二维图，纵向是时间轴，时间沿竖线向下延伸。横向轴代表了在协作中各独立对象的类元角色，类元角色的活动用生命线表示。



对象
生命线
启动交互的对象放在最左边，随后放入消息的对象放在启动交互对象的右边。

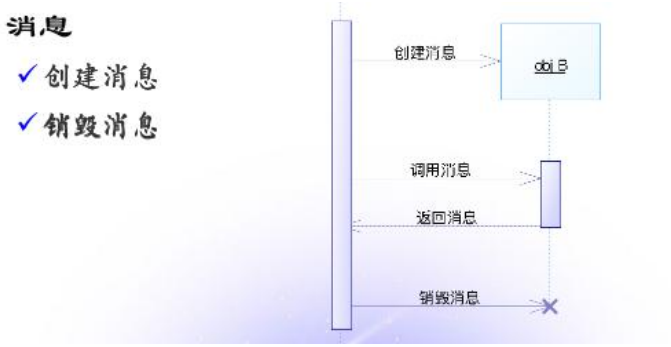
在顺序图中，有的消息对应于激活，表示它将会激活一个对象，这种消息称为调用消息(Call Message)；如果消息没有对应激活框，表示它不是一个调用消息，不会引发其他对象的活动，这种消息称为发送消息(Send Message)。

三种消息



创建消息

销毁消息



交互片段

一个复杂的顺序图可以划分为几个小块，每一个小块称为一个交互片段。每个交互片段由一个大方框包围，其名称显示在方框左上角的间隔区内，表示该顺序图的信息。常用操作符如下：

alt:多条路径，条件为真时执行。

opt:任选，仅当条件为真时执行。

par:并行，每一片段都并发执行。

loop:循环，片段可多次执行。

critical:临界区，只能有一个线程对它立即执行

类图:

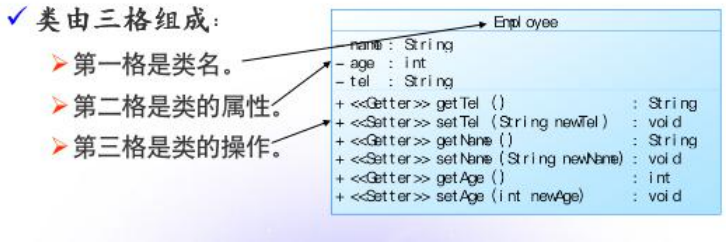
类(Class)封装了数据和行为，是面向对象的重要组成部分，它是具有相同属性、操作、关系的对象集合的总称。

在系统中，每个类具有一定的职责，职责指的是类所担任的任务，即类要完成什么样的功能，要承担什么样的义务。一个类可以有多种职责，设计得好的类一般只有一种职责（单一职责原则），在定义类的时候，将类的职责分解成为类的属性和操作（即方法）。

类的属性即类的数据职责，类的操作即类的行为职责。

类图使用需要出现在系统内的不同的类来描述系统的静态结构，类图包含类和它们之间的关系，它描述系统内所声明的类，但它没有描述系统运行时类的行为。

在 UML 中，类使用具有类名称、属性、操作分隔的长方形来表示：



属性表示方式:

可见性 名称:类型 [= 默认值]

Public +
Private -
Protected #

类的操作表示方式:

可见性 名称(参数列表) [: 返回类型]

类与类之间的关系

关联关系

关联关系(Association)是类与类之间最常用的一种关系，它是一种结构化关系，用于表示一类对象与另一类对象之间有联系。

在 UML 类图中，用实线连接有关联的对象所对应的类，在使用 Java、C#和 C++等编程语言实现关联关系时，通常将一个类的对象作为另一个类的属性。
默认双向关联,单向关联有箭头的实线
自关联

多重性关联
多重性关联关系又称为重数性关联关系(Multiplicity)，表示一个类的对象与另一个类的对象连接的个数。

表示方式	多重性说明
1..1	表示另一个类的一个对象只与一个该类对象有关系
0..*	表示另一个类的一个对象与零个或多个该类对象有关系
1..*	表示另一个类的一个对象与一个或多个该类对象有关系
0..1	表示另一个类的一个对象没有或只与一个该类对象有关系
m..n	表示另一个类的一个对象与最少m、最多n个该类对象有关系(m<=n)

聚合关系:Aggregation

聚合关系(Aggregation)表示一个整体与部分的关系。通常在定义一个整体类后，再去分析这个整体类的组成结构，从而找出一些成员类，该整体类和成员类之间就形成了聚合关系。
在聚合关系中，成员类是整体类的一部分，即成员对象是整体对象的一部分，但是成员对象可以脱离整体对象独立存在。在 UML 中，聚合关系用带空心菱形的直线表示。
构造注入或者设值注入.

组合关系(Composition)也表示类之间整体和部分的关系，但是组合关系中部分和整体具有统一的生存期。一旦整体对象不存在，部分对象也将不存在，部分对象与整体对象之间具有同生共死的关系。

在组合关系中，成员类是整体类的一部分，而且整体类可以控制成员类的生命周期，即成员类的存在依赖于整体类。在 UML 中，组合关系用带实心菱形的直线表示。

在构造函数里面构造.

依赖关系
依赖关系(Dependency)是一种使用关系，特定事物的改变有可能会影响到使用该事物的其他事物，在需要表示一个事物使用另一个事物时使用依赖关系。大多数情况下，依赖关系体现在某个类的方法使用另一个类的对象作为参数。
在 UML 中，依赖关系用带箭头的虚线表示，由依赖的一方指向被依赖的一方。

参数,局部变量,静态方法

泛化关系
泛化关系(Generalization)也就是继承关系，也称为“is-a-kind-of”关系，泛化关系用于描述父类与子类之间的关系，父类又称作基类或超类，子类又称作派生类。在 UML 中，泛化关系用带空心三角形的直线来表示。

接口之间也可以有与类之间关系类似的继承关系和依赖关系,但是接口和类之间还存在一种实现关系(Realization)，在这种关系中，类实现了接口，类中的操作实现了接口中所声明的操作。在 UML 中，类与接口之间的实现关系用带空心三角形的虚线来表示。

包图

包是一种把元素组织到一起的通用机制，包可以嵌套于其他包中。
包图用于描述包与包之间的关系，包的图标是一个带标签的文件夹。

组件图

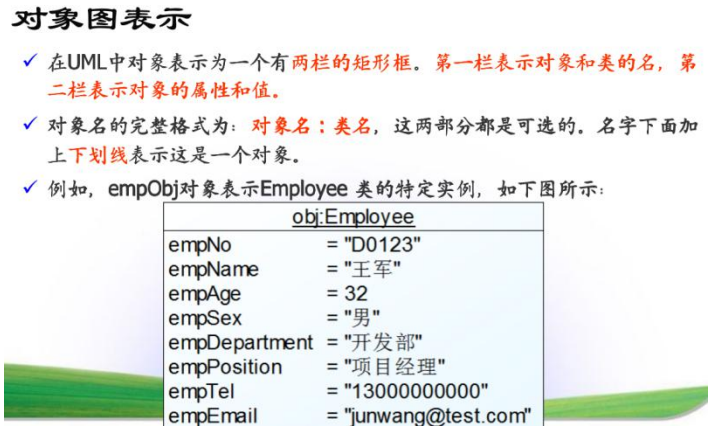
组件图又称为构件图(Component Diagram) 。组件图中通常包括组件、接口，以及各种关系。组件图显示组件以及它们之间的依赖关系，它可以用来显示程序代码如何分解成模块或组件。

部署图:

部署图(Deployment Diagram)，也称为实施图，描述系统硬件的物理拓扑结构及在此结构上执行的软件。部署图可以显示计算节点的拓扑结构和通信路径、节点上运行的软件组件。

对象图:

对象图是类图在某一时刻的一个实例



组合结构图

组合结构图将每一个类放在一个整体中，从类的内部结构来审视一个类。
用于表示一个类的内部结构

通信图 协作图

通信图强调参与一个交互对象的组织。
它与顺序图是同构图，也就是它们包含了相同的信息，只是表达方式不同而已，通信图与顺序图可以相互转换

顺序图清晰地显示了时间次序，但没有显式指明对象间关系。通信图清晰地显示了对象间关系，但时间次序必须从顺序号来获得。

定时图:

定时图采用一种带数字刻度的时间轴来精确地描述消息的顺序，而不是像顺序图那样只是指定消息的相对顺序

交互概览图

交互概览图是交互图(顺序图)与活动图的混合物，可以把交互概览图理解为细化的活动图，在其中的活动都通过一些小型的顺序图来表示

图名	概述	使用频率
用例图	描述用户与系统如何交互	★★★★★
类图	描述类、类的特性以及类之间的关系	★★★★★
包图	描述类的层次结构	★★★★☆
顺序图	描述对象之间的交互，重点在于强调顺序	★★★★☆
状态图	描述事件如何改变对象状态	★★★☆☆
活动图	描述过程行为及其并发行为	★★★☆☆
组件图	描述组件的结构与连接	★★★☆☆
部署图	描述各个节点上组件的部署情况及节点间的连接	★★★☆☆
通信图	描述对象之间的交互，重点在于连接	★★☆☆☆
对象图	描述一个时间点上系统中各个对象的一个快照	★★☆☆☆
组合结构图	描述类结构的分解	★☆☆☆☆
交互概览图	顺序图与活动图的混合	★☆☆☆☆
定时图	描述对象之间的交互，重点在于定时	★☆☆☆☆

面向对象设计原则与设计模式 50

单一职责原则:

单一职责原则：一个对象应该只包含单一的职责，并且该职责被完整地封装在一个类中。
Single Responsibility Principle (SRP): Every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class.

开闭原则:

开闭原则：软件实体应当对扩展开放，对修改关闭。
Open-Closed Principle (OCP): Software entities should be open for extension, but closed for modification.

里氏代换原则

里氏代换原则：所有引用基类的地方必须能透明地使用其子类的对象。

Liskov Substitution Principle (LSP): Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

依赖倒转原则

依赖倒转原则：高层模块不应该依赖低层模块，它们都应该依赖抽象。抽象不应该依赖于细节，细节应该依赖于抽象。

Dependency Inversion Principle (DIP): 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.

接口隔离原则

接口隔离原则：客户端不应该依赖那些它不需要的接口。

Interface Segregation Principle (ISP): Clients should not be forced to depend upon interfaces that they do not use.

合成复用原则

合成复用原则：优先使用对象组合，而不是继承来达到复用的目的。

Composite Reuse Principle (CRP)： Favor composition of objects over inheritance as a reuse mechanism.

迪米特法则

迪米特法则：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。

Law of Demeter (LoD): Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.

---创建型

简单工厂模式

简单工厂模式 (Simple Factory Pattern)：定义一个工厂类，它可以根据参数的不同返回不同类的实例，被创建的实例通常都具有共同的父类。

模式优点
实现了对象创建和使用的分离
客户端无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可
通过引入配置文件，可以在不修改任何客户端代码的情况下更换和增加新的具体产品类，在一定程度上提高了系统的灵活性

模式缺点
工厂类集中了所有产品的创建逻辑，职责过重，一旦不能正常工作，整个系统都要受到影响
增加系统中类的个数（引入了新的工厂类），增加了系统的复杂度和理解难度
系统扩展困难，一旦添加新产品不得不修改工厂逻辑
由于使用了静态工厂方法，造成工厂角色无法形成基于继承的等级结构，工厂类不能得到很好地扩展

模式适用环境
工厂类负责创建的对象比较少，由于创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂
客户端只知道传入工厂类的参数，对于如何创建对象并不关心

工厂方法模式

工厂方法模式：定义一个用于创建对象的接口，但是让子类决定将哪一个类实例化。工厂方法模式让一个类的实例化延迟到其子类。

Factory Method Pattern: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

模式优点
工厂方法用来创建客户所需要的产品，同时还向客户隐藏了哪种具体产品类将被实例化这一细节

能够让工厂自主确定创建何种产品对象，而如何创建这个对象的细节则完全封装在具体工厂内部
在系统中加入新产品时，完全符合开闭原则

模式缺点
系统中类的个数将成对增加，在一定程度上增加了系统的复杂度，会给系统带来一些额外的开销
增加了系统的抽象性和理解难度

模式适用环境
客户端不知道它所需要的对象的类（客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体产品对象由具体工厂类创建）
抽象工厂类通过其子类来指定创建哪个对象

抽象工厂

抽象工厂模式：提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。

Abstract Factory Pattern: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

模式优点
隔离了具体类的生成，使得客户端并不需要知道什么被创建
当一个产品族中的多个对象被设计成一起工作时，它能够保证客户端始终只使用同一个产品族中的对象
增加新的产品族很方便，无须修改已有系统，符合开闭原则

模式缺点
增加新的产品等级结构麻烦，需要对原有系统进行较大的修改，甚至需要修改抽象层代码，这显然会带来较大的不便，违背了开闭原则

模式适用环境
一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节
系统中有多于一个的产品族，但每次只使用其中某一产品族
属于同一个产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来
产品等级结构稳定，在设计完成之后不会向系统中增加新的产品等级结构或者删除已有的产品等级结构

原型模式

原型模式：使用原型实例指定待创建对象的类型，并且通过复制这个原型来创建新的对象。

Prototype Pattern: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

模式优点
简化对象的创建过程，通过复制一个已有实例可以提高新实例的创建效率
扩展性较好
提供了简化的创建结构，原型模式中产品的复制是通过封装在原型类中的克隆方法实现的，无须专门的工厂类来创建产品
可以使用深克隆的方式保存对象的状态，以便在需要的时候使用，可辅助实现撤销操作

模式缺点
需要为每一个类配备一个克隆方法，而且该克隆方法位于一个类的内部，当对已有的类进行改造时，需要修改源代码，违背了开闭原则
在实现深克隆时需要编写较为复杂的代码，而且当对象之间存在多重的嵌套引用时，为了实现深克隆，每一层对象对应的类都必须支持深克隆，实现起来可能会比较麻烦

模式适用环境
创建新对象成本较大，新对象可以通过复制已有对象来获得，如果是相似对象，则可以对其成员变量稍作修改
系统要保存对象的状态，而对象的状态变化很小
需要避免使用分层次的工厂类来创建分层次的对象
Ctrl + C -> Ctrl + V

单例模式:

单例模式：确保一个类只有一个实例，并提供一个全局访问点来访问这个唯一实例。

Singleton Pattern: Ensure a class has only one instance, and provide a global point of access to it.

模式优点
提供了对唯一实例的受控访问
可以节约系统资源，提高系统的性能
允许可变数目的实例（多例类）

模式缺点
扩展困难（缺少抽象层）
单例类的职责过重
由于自动垃圾回收机制，可能会导致共享的单例对象的状态丢失

模式适用环境
系统只需要一个实例对象，或者因为资源消耗太大而只允许创建一个对象
客户调用类的单个实例只允许使用一个公共访问点，除了该公共访问点，不能通过其他途径访问该实例

---结构型

适配器模式

适配器模式：将一个类的接口转换成客户希望的另一个接口。适配器模式让那些接口不兼容的类可以一起工作。

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.

模式优点
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，无须修改原有结构
增加了类的透明性和复用性，提高了适配者的复用性，同一个适配者类可以在多个不同的系统中复用
灵活性和扩展性非常好
类适配器模式：置换一些适配者的方法很方便
对象适配器模式：可以把多个不同的适配者适配到同一个目标，还可以适配一个适配者的子类

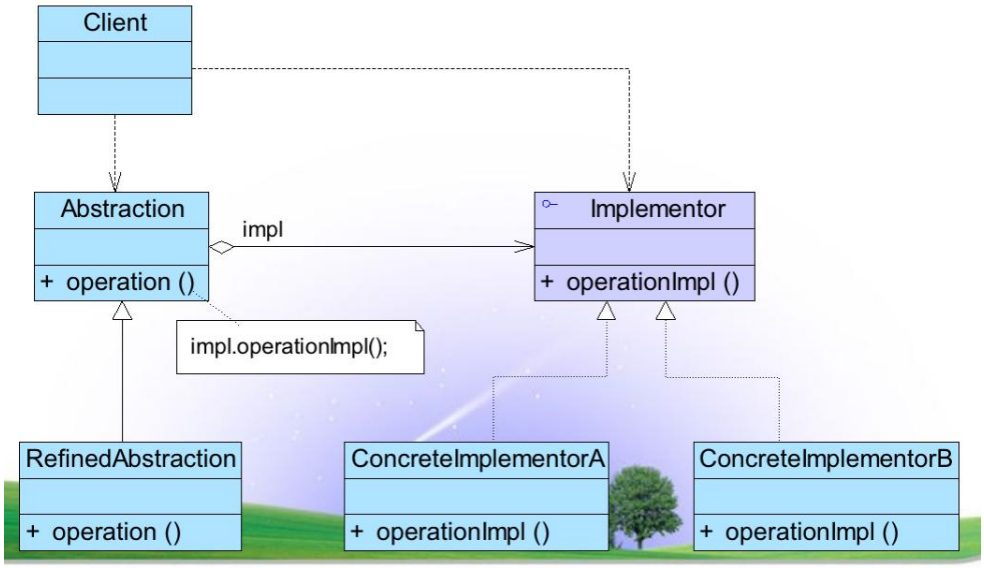
模式缺点
类适配器模式：(1) 一次最多只能适配一个适配者类，不能同时适配多个适配者；(2) 适配者类不能为最终类；(3) 目标抽象类只能为接口，不能为类
对象适配器模式：在适配器中置换适配者类的某些方法比较麻烦

模式适用环境
系统需要使用一些现有的类，而这些类的接口不符合系统的需要，甚至没有这些类的源代码
创建一个可以重复使用的类，用于和一些彼此之间没有太大关联的类，包括一些可能在将来引进的类一起工作.

桥接模式:

桥接模式：将抽象部分与它的实现部分解耦，使得两者都能够独立变化。

Bridge Pattern: Decouple an abstraction from its implementation so that the two can vary independently.



模式优点
分离抽象接口及其实现部分
可以取代多层继承方案，极大地减少了子类的个数
提高了系统的可扩展性，在两个变化维度中任意扩展一个维度，不需要修改原有系统，符合开闭原则

模式缺点
会增加系统的理解与设计难度，由于关联关系建立在抽象层，要求开发者一开始就针对抽象层进行设计与编程
正确识别出系统中两个独立变化的维度并不是一件容易的事情.

模式适用环境
需要在抽象化和具体化之间增加更多的灵活性，避免在两个层次之间建立静态的继承关系
抽象部分和实现部分可以以继承的方式独立扩展而互不影响
一个类存在两个（或多个）独立变化的维度，且这两个（或多个）维度都需要独立地进行扩展
不希望使用继承或因为多层继承导致系统类的个数急剧增加的系统

组合模式:

组合模式：组合多个对象形成树形结构以表示具有部分-整体关系的层次结构。组合模式让客户端可以统一对待单个对象和组合对象。

Composite Pattern: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

模式优点
可以清楚地定义分层次的复杂对象，表示对象的全部或部分层次，让客户端忽略了层次的差异，方便对整个层次结构进行控制
客户端可以一致地使用一个组合结构或其中单个对象，不必关心处理的是单个对象还是整个组合结构，简化了客户端代码
增加新的容器构件和叶子构件都很方便，符合开闭原则
为树形结构的面向对象实现提供了一种灵活的解决方案

模式缺点
在增加新构件时很难对容器中的构件类型进行限制

模式适用环境
在具有整体和部分的层次结构中，希望通过一种方式忽略整体与部分的差异，客户端可以一致地对待它们
在一个使用面向对象语言开发的系统中需要处理一个树形结构
在一个系统中能够分离出叶子对象和容器对象，而且它们的类型不固定，需要增加一些新的类型

外观模式

外观模式：为子系统中的一组接口提供一个统一的入口。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。

Facade Pattern: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

模式优点
它对客户端屏蔽了子系统组件，减少了客户端所需处理的对象数目，并使得子系统使用起来更加容易
它实现了子系统与客户端之间的松耦合关系，这使得子系统的变化不会影响到调用它的客户端，只需要调整外观类即可

一个子系统的修改对其他子系统没有任何影响，而且子系统的内部变化也不会影响到外观对象

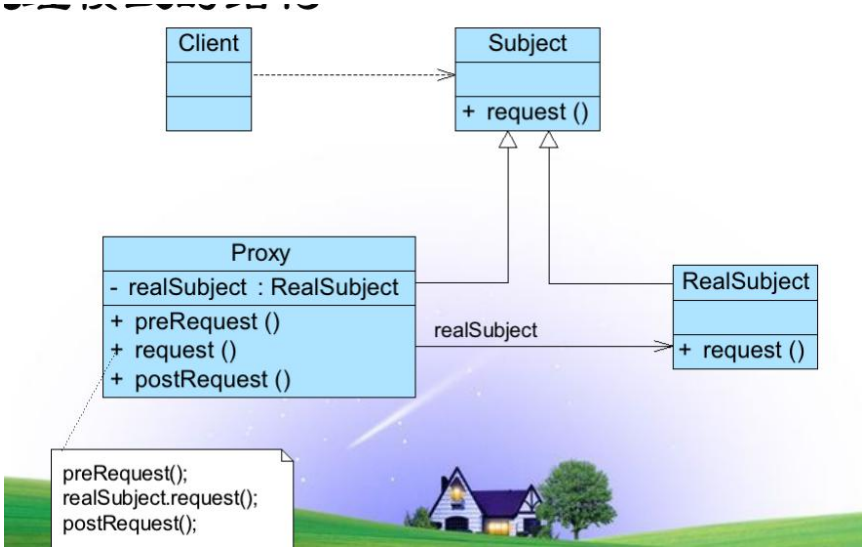
模式缺点
不能很好地限制客户端直接使用子系统类，如果对客户端访问子系统类做太多的限制则减少了可变性和灵活性
如果设计不当，增加新的子系统可能需要修改外观类的源代码，违背了开闭原则

模式适用环境
要为访问一系列复杂的子系统提供一个简单入口
客户端程序与多个子系统之间存在很大的依赖性
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，而是通过外观类建立联系，降低层之间的耦合度

代理模式

代理模式：给某一个对象提供一个代理或占位符，并由代理对象来控制对原对象的访问。

Proxy Pattern: Provide a surrogate or placeholder for another object to control access to it.



模式优点
能够协调调用者和被调用者，在一定程度上降低了系统的耦合度
客户端可以针对抽象主题角色进行编程，增加和更换代理类无须修改源代码，符合开闭原则，系统具有较好的灵活性和可扩展性

模式优点——逐个分析
远程代理：可以将一些消耗资源较多的对象和操作移至性能更好的计算机上，提高了系统的

整体运行效率
虚拟代理：通过一个消耗资源较少的对象来代表一个消耗资源较多的对象，可以在一定程度上节省系统的运行开销
缓冲代理：为某一个操作的结果提供临时的缓存存储空间，以便在后续使用中能够共享这些结果，优化系统性能，缩短执行时间
保护代理：可以控制对一个对象的访问权限，为不同用户提供不同级别的使用权限

模式缺点
由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢（例如保护代理）
实现代理模式需要额外的工作，而且有些代理模式的实现过程较为复杂（例如远程代理）

模式适用环境
当客户端对象需要访问远程主机中的对象时可以使用远程代理
当需要用一个消耗资源较少的对象来代表一个消耗资源较多的对象，从而降低系统开销、缩短运行时间时可以使用虚拟代理
当需要为某一个被频繁访问的操作结果提供一个临时存储空间，以供多个客户端共享访问这些结果时可以使用缓冲代理
当需要控制对一个对象的访问，为不同用户提供不同级别的访问权限时可以使用保护代理
当需要为一个对象的访问（引用）提供一些额外的操作时可以使用智能引用代理

---行为型模式

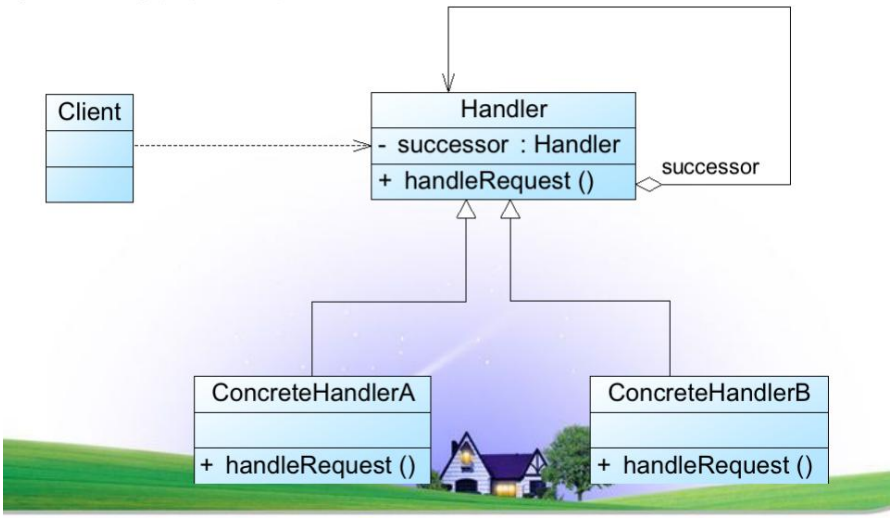
类行为型模式
使用继承关系在几个类之间分配行为，主要通过多态等方式来分配父类与子类的职责
对象行为型模式
使用对象的关联关系来分配行为，主要通过对象关联等方式来分配两个或多个类的职责

职责链模式

职责链模式：避免将一个请求的发送者与接收者耦合在一起，让多个对象都有机会处理请求。将接收请求的对象连接成一条链，并且沿着这条链传递请求，直到有一个对象能够处理它为止。

Chain of Responsibility Pattern: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

职责链模式的结构



模式优点
使得一个对象无须知道是其他哪一个对象处理其请求，降低了系统的耦合度
可简化对象之间的相互连接
给对象职责的分配带来更多的灵活性
增加一个新的具体请求处理者时无须修改原有系统的代码，只需要在客户端重新建链即可

模式缺点
不能保证请求一定会被处理
对于比较长的职责链，系统性能将受到一定影响，在进行代码调试时不太方便
如果建链不当，可能会造成循环调用，将导致系统陷入死循环

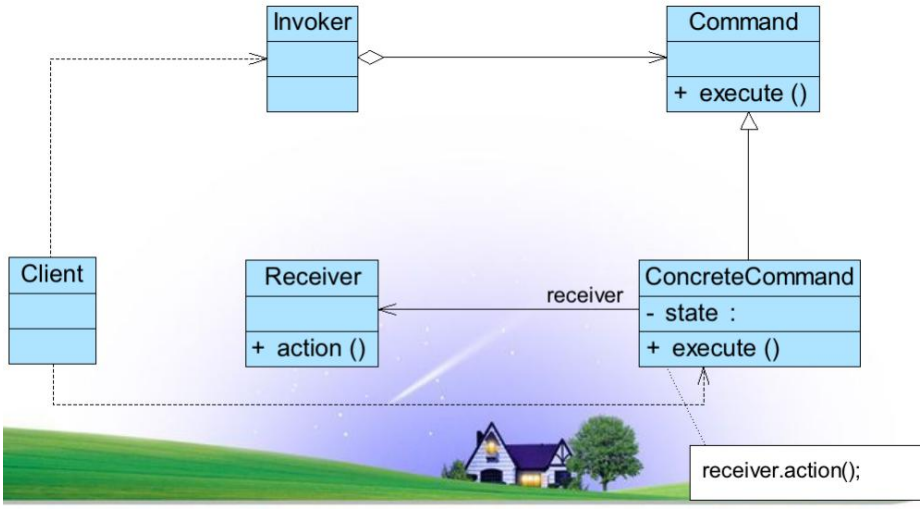
模式适用环境
有多个对象可以处理同一个请求，具体哪个对象处理该请求待运行时刻再确定
在不明确指定接收者的情况下，向多个对象中的一个提交一个请求
可动态指定一组对象处理请求

命令模式

命令模式：将一个请求封装为一个对象，从而让你可以用不同的请求对客户进行参数化，对请求排队或者记录请求日志，以及支持可撤销的操作。

Command Pattern: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

命令模式的结构



模式优点
降低系统的耦合度
新的命令可以很容易地加入到系统中，符合开闭原则
可以比较容易地设计一个命令队列或宏命令（组合命令）
为请求的撤销(Undo)和恢复(Redo)操作提供了一种设计和实现方案

模式缺点
使用命令模式可能会导致某些系统有过多的具体命令类(针对每一个对请求接收者的调用操作都需要设计一个具体命令类)

迭代器模式

迭代器模式：提供一种方法顺序访问一个聚合对象中各个元素，且不用暴露该对象的内部表示。

Iterator Pattern: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

模式优点
支持以不同的方式遍历一个聚合对象，在同一个聚合对象上可以定义多种遍历方式
简化了聚合类
由于引入了抽象层，增加新的聚合类和迭代器类都很方便，无须修改原有代码，符合开闭原则

模式缺点
在增加新的聚合类时需要对应地增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性

抽象迭代器的设计难度较大，需要充分考虑到系统将来的扩展。在自定义迭代器时，创建一个考虑全面的抽象迭代器并不是一件很容易的事情

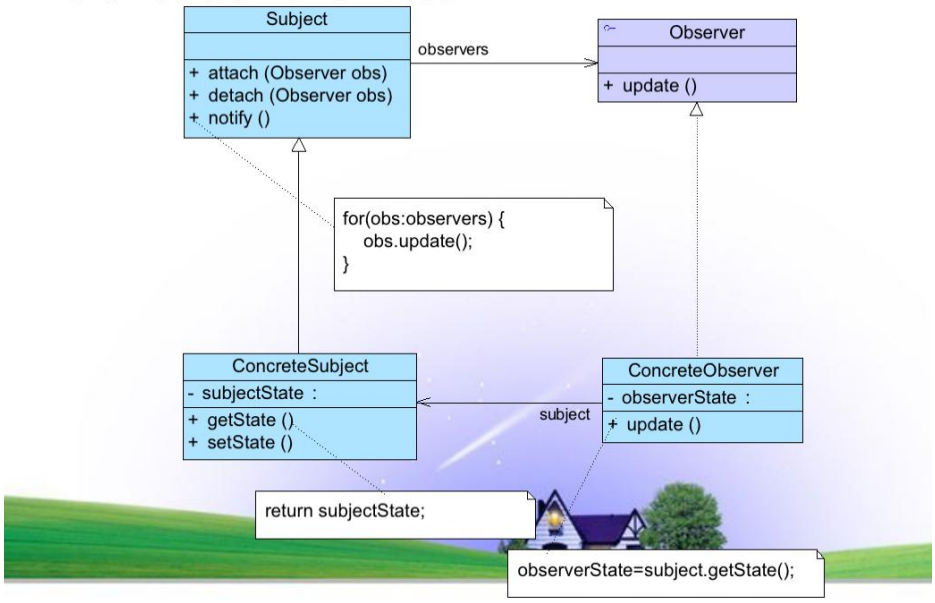
模式适用环境
访问一个聚合对象的内容而无须暴露它的内部表示
需要为一个聚合对象提供多种遍历方式
为遍历不同的聚合结构提供一个统一的接口，在该接口的实现类中为不同的聚合结构提供不同的遍历方式，而客户端可以一致性地操作该接口

观察者模式

观察者模式：定义对象之间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象都得到通知并被自动更新。

Observer Pattern: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

观察者模式的结构



事件源对象充当观察目标角色，事件监听器充当抽象观察者角色，事件处理对象充当具体观察者角色

模式优点
可以实现表示层和数据逻辑层的分离
在观察目标和观察者之间建立一个抽象的耦合
支持广播通信，简化了一对多系统设计的难度
符合开闭原则，增加新的具体观察者无须修改原有系统代码，在具体观察者与观察目标之间

不存在关联关系的情况下，增加新的观察目标也很方便

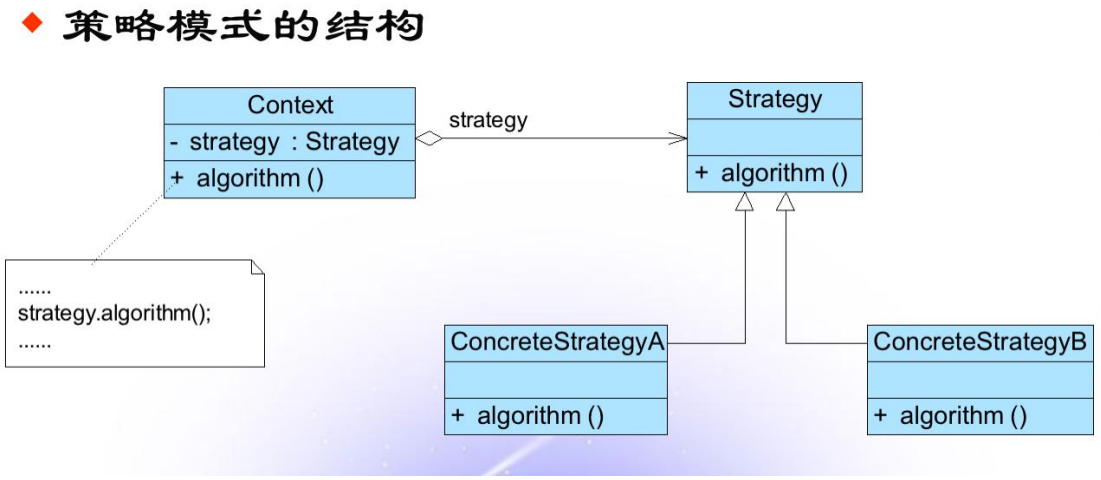
模式缺点
将所有的观察者都通知到会花费很多时间
如果存在循环依赖时可能导致系统崩溃
没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而只是知道观察目标发生了变化

模式适用环境
一个抽象模型有两个方面，其中一个方面依赖于另一个方面，将这两个方面封装在独立的对象中使它们可以各自独立地改变和复用
一个对象的改变将导致一个或多个其他对象发生改变，且并不知道具体有多少对象将发生改变，也不知道这些对象是谁
需要在系统中创建一个触发链

策略模式

策略模式：定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法可以独立于使用它的客户变化。

Strategy Pattern: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



模式优点
提供了对开闭原则的完美支持，用户可以在不修改原有系统的基础上选择算法或行为，也可以灵活地增加新的算法或行为
提供了管理相关的算法族的办法
提供了一种可以替换继承关系的办法
可以避免多重条件选择语句
提供了一种算法的复用机制，不同的环境类可以方便地复用策略类

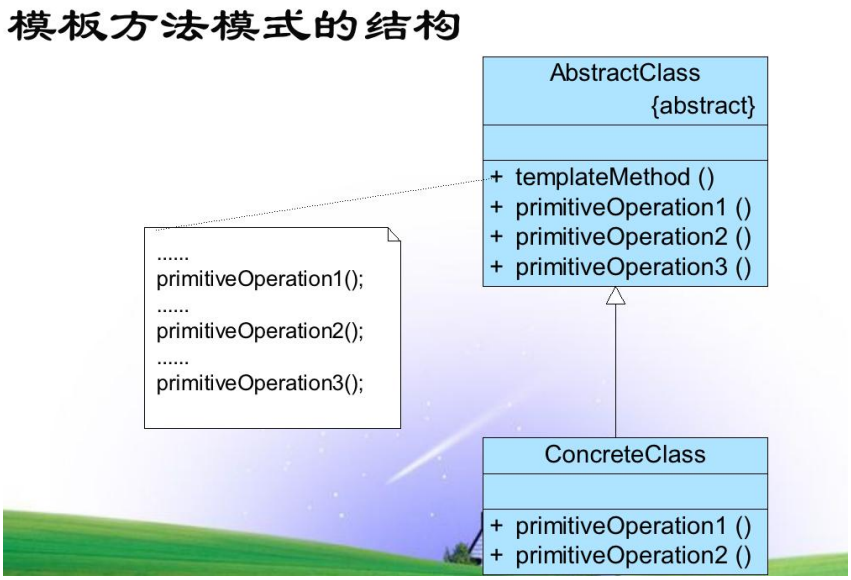
模式缺点
客户端必须知道所有的策略类，并自行决定使用哪一个策略类
将造成系统产生很多具体策略类
无法同时在客户端使用多个策略类

模式适用环境
一个系统需要动态地在几种算法中选择一种
避免使用难以维护的多重条件选择语句
不希望客户端知道复杂的、与算法相关的数据结构，提高算法的保密性与安全性

模版方法模式

模板方法模式：定义一个操作中算法的框架，而将一些步骤延迟到子类中。模板方法模式使得子类不改变一个算法的结构即可重定义该算法的某些特定步骤。

Template Method Pattern: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



模板方法 (Template Method)
基本方法 (Primitive Method)
抽象方法(Abstract Method)
具体方法(Concrete Method)
钩子方法(Hook Method)

模式优点

在父类中形式化地定义一个算法，而由它的子类来实现细节的处理，在子类实现详细的处理算法时并不会改变算法中步骤的执行次序
提取了类库中的公共行为，将公共行为放在父类中，而通过其子类来实现不同的行为
可实现一种反向控制结构，通过子类覆盖父类的钩子方法来决定某一特定步骤是否需要执行
更换和增加新的子类很方便，符合单一职责原则和开闭原则

模式缺点
需要为每一个基本方法的不同实现提供一个子类，如果父类中可变的基本方法太多，将会导致类的个数增加，系统会更加庞大，设计也更加抽象（可结合桥接模式）

模式适用环境
一次性实现一个算法的不变部分，并将可变的行为留给子类来实现
各子类中公共的行为应被提取出来，并集中到一个公共父类中，以避免代码重复
需要通过子类来决定父类算法中某个步骤是否执行，实现子类对父类的反向控制

押题:
结合工厂方法模式,谈谈对开闭原则的理解.开闭原则的定义,结合代码片段.
1. 开闭原则的定义：软件实体应该对扩展开放，对修改关闭。【4 分】
说明：【8 分】
工厂方法模式代码片段如下：

```
//抽象工厂类
public abstract class Factory {
public abstract Product createProduct();
}
//具体工厂类
public class ConcreteFactoryA extends Factory {
public Product createProduct() {
return new ConcreteProductA();
}
}
```

在客户端存在如下代码片段：

```
……
Factory factory;
factory = new ConcreteFactoryA(); //可通过反射和配置文件来实现，修改具体工厂类无须修
```

改源代码

```
Product product;
product = factory.createProduct();
```

在工厂方法模式中，引入了抽象工厂类，例如上面代码中的 Factory，所有具体工厂类 2 都是 Factory 的子类（例如 ConcreteFactoryA），具体工厂类负责创建具体的产品。如果要新
增一个具体产品 ConcreteProductB，只需要对应增加一个新的具体工厂类 ConcreteFactoryB

作为 Factory 类的子类，并实现在 Factory 中声明的 createProduct()方法即可，无须修改已有
工厂类和产品类的源代码；在客户端代码中需要将 ConcreteFactoryA 改为 ConcreteFactoryB，
如果采取反射和配置文件等方式来实现的话，只需修改配置文件中存储的类名即可，客户端代码也无须修改，完全符合开闭原则。

考试题型:

选择题 10 个 20 分 五个英文
填空题 14 个 14 分 部分英文
简答题 4 个 16 分 一个英文
软件体系结构风格、UML、面向对象设计原则、设计模式、软件质量属性
综合题 5 个 50 分 两个英文题(模式的英文名
1. 根据给定场景绘制用例图
2. 根据场景选择两种合适的设计模式,涉及到模式的名称和定义,结合实例绘制结构图(类图
3. 同二
4. 选择合适的设计模式,单个,给出对应模式的定义.并且绘制结构图
5. 结合某个设计模式,谈谈对某个设计模式的理解(结合 设计原则定义、模式类图并结合简单代码进行回答)