总复习

软件工程学科发展历程

软件工程：IEEE软件工程知识体系指南SWEBOK定义：
（1）软件开发、实施、维护的系统化、规范化、可量化的方法的应用、也就是软件的应用工程。
（2）对上述方法的研究

软件开发过程模型

软件开发过程

过程

针对一个给定目的的一系列操作步骤。
每个过程都有明确的以及具体的操作步骤，操作步骤说明了有哪些操作以及按照说明方式来执行操作。

软件开发过程

按照项目的进度、成本和质量限制、开发和维护满足用户需求的软件所必需的一组有序的软件开发活动集合。
eg：需求分析、软件设计、编码、测试

软件开发过程的组成

（1）软件开发
软件开发活动中存在许多相互关联的软件开发活动间的关系
eg：需求获取、需求分析、需求规格说明书撰写
（2）软件开发活动
为开发软件项目而执行的一项具有明确任务的具体工作。
eg：需求分析、规划项目范围
1．技术活动：对软件项目实施开发、产生软件产品。
eg：需求分析、概要设计、详细设计
2．管理活动：对软件项目中的人、产品和过程等实施管理的活动。
eg：制定软件项目计划、软件配置

软件开发过程模型

（1）定义软件开发活动
为什么需要过程：
1．明确软件开发过程和步骤，促进工程化软件开发、便于指定制定软件项目计划；
2．为软件开发提供可视性，便于软件开发过程管理和控制；

3．便于细化和安排任务，使得每个人明确各自的工作。

（2）什么是软件开发过程模型？

1．软件开发模型是软件开发全过程、软件开发活动以及他们之间的关系的结构框架

2．找到软件开发、以及软件开发过程的定义

3．常见模型：

瀑布模型、原型模型、增量模型、迭代模型、敏捷模型

软件开发生命周期模型

经典生命周期

（1）预测型生命周期（驱动型项目生命周期）

指事先详细定义项目可交付成果，尽量预测出以后需要开展的项目工作，编制出详细的项目计划，然后在执行阶段完成已定义好的项目工作和可交付成果，在收尾阶段验收并移交已完成的项目可交付成果。

1．预测型项目生命周期的特点

先设计好要做的产品，再实际去做，在做的过程中一般不进行实质性变更。如果想变更，必须进行严格控制。

2．预测型项目生命周期适用于有成熟做法、风险较低、待开发产品清晰明确的项目，如建筑工程项目，同时也适用于只能作为一个整体交付并发挥作用的项目产品。

3．预测型生命周期一般适合在以下场合使用：

1．充分了解拟交付的产品。

2．有厚实的行业实践基础。

3．整批一次性交付产品有利于干系人。

（2）适应型生命周期（敏捷项目生命周期）

指随用户需求的变化，通过短期迭代来逐步完善项目产品，直到生产出最终产品。

1．特点是，在每个迭代期都设计并生产出能满足用户当前需求的产品原形，并在下一个迭代期根据用户需求的变化，完善产品原型，相当于边设计边生产。

2．适应型生命周期（Adaptive Life Cycle）也称为变更驱动方法或敏捷方法，其目的在于应对大量变更，获取干系人的持续参与。适应型生命周期也包含迭代和增量的概念，但不同之处在于，迭代很快，通常2～4周迭代一次，而且所需时间和资源是固定的。例如，软件开发项目中的敏捷方法就是适应型生命周期的应用。

3．适应型生命周期一般适合在以下场合使用：

1．需要应对快速变化的环境。

2．需求和范围难以事先确定。

3．能够以有利于干系人的方式定义较小的增量改进。

（3）迭代与增量型生命周期

1．迭代与增量型生命周期是指同时采用迭代和增量的方式来开发产品，迭代是通过一系列重复的循环的活动来开发产品，增量的方法是通过渐进来增加产品功能。在迭代型生命周期（Interative Life Cycle）和增量型生命周期（Incremental Life Cycle）中，同时采用迭代和增量的方式来开发产品，随着项目团队对产品的理解程度逐渐提高，项目阶段有目的地重复一个或多个项目活动。例如，在软件开发项目中，迭代模型是典型的迭代型生命周期的应用，增量模型是典型的增量型生命周期的应用。

2．这种生命周期模型适用于组织需要管理不断变化的目标和范围，或是组织需要降低项目的复杂性，以及产品的部分交付有利于一个或多个相关方，且不影响最终交付。

3．迭代和增量型生命周期一般适合在以下场合使用：

1．组织需要管理不断变化的目标和范围。
2．组织需要降低项目的复杂性。
3．产品的部分交付有利于一个或多个干系人，且不会影响最终或整批可交付成果的交付。

软件项目生命周期（SDLC）的阶段划分

指软件从产生直到报废的生命周期，生命周期包含问题定义、可行性分析、总体描述、系统设计、编码、调试和测试、验收与运行、维护升级到废弃等阶段；

按照软件工程划分SDLC，分为6个阶段：

1. 计划阶段
2. 需求分析阶段
3. 软件设计阶段
4. 编码阶段
5. 测试阶段
6. 运行维护阶段

软件开发方法

软件开发流程

系统设计-设计原则

总体原则：高内聚、低耦合
1. 内聚：一个模块内各个元素彼此结合的紧密程度
2. 耦合：一个软件结构内不同模块之间互连程度的度量
3. 高内聚：在一个模块内，让每个元素之间都尽可能的紧密相连。也就是充分利用每一个元素的功能，各施所能，以最终实现某个功能。如果某个元素与该模块的关系比较疏松的话，可能该模块的结构还不够完善，或者是该元素是多余的。
4. 低耦合：一个完整的系统，模块与模块之间，尽可能的使其独立存在。也就是说，让每个模块，尽可能的独立完成某个特定的子功能。模块与模块之间的接口，尽量的少而简单。如果某两个模块间的关系比较复杂的话，最好首先考虑进一步的模块划分。这样有利于修改和组合。
内聚和耦合，包含了横向和纵向的关系。功能内聚和数据耦合，是我们需要达成的目标。横向的内聚和耦合，通常体现在系统的各个模块、类之间的关系，而纵向的耦合，体现在系统的各个层次之间的关系。

代码设计原则

设计模式

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。

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)

目的

编写软件过程中，程序员面临来自耦合性、内聚性及可维护性、可扩展性、重用性、灵活性等多方面的挑战。设计模式是为了让程序（软件），具有更好的：
（1）代码重用性：即相同功能的代码。不用多次编写；
（2）代码可读性：即编程规范性，便于其他程序员阅读理解；
（3）可扩展性：即当需要增加新的功能时，非常的方便方面，称为可维护性；
（4）可靠性：即当我们需要增加新功能后，对原来的功能没有影响；
（5）使程序呈现高内聚、低耦合的特性。

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。

在设计模式经典著作《GOF95》中，设计模式从应用的角度被分为三个大的类型
创建型模式/结构型模式/行为型模式

根据模式的范围分，模式用于类还是用于对象
类模式：处理类和子类之间的关系，这些关系通过继承建立，是静态的，在编译时刻便确定下来了
对象模式：处理对象间的关系，这些关系在运行时刻是可以变化的，更具动态性

从某种意义上来说，几乎所有模式都使用继承机制，所以“类模式”只指那些集中于处理类间关系的模式，而大部分模式都属于对象模式的范畴

GOF设计模式：

范围\目的	创建型模式	结构型模式	行为型模式
类模式	工厂方法模式	（类）适配器模式	解释器模式 模板方法模式
对象模式	抽象工厂模式 建造者模式 原型模式 单例模式	（对象）适配器模式 桥接模式 组合模式 装饰模式 外观模式 享元模式 代理模式	职责链模式 命令模式 迭代器模式 中介者模式 备忘录模式 观察者模式 状态模式 策略模式 访问者模式

23种设计模式

创建型模式
工厂方法模式　抽象工厂模式　单例模式　建造者模式　原型模式

结构型模式
适配器模式　装饰器模式　代理模式　门面模式　桥接模式　组合模式　享元模式

行为型模式
策略模式　模板方法模式　观察者模式　迭代子模式　责任链模式　命令模式
备忘录模式　状态模式　访问者模式　中介者模式　解释器模式

创建型模式

用来创建对象的模式，抽象了实例化过程

(1)工厂(Factory Method)模式：父类负责定义创建对象的公共接口，而子类则负责生成具体对象，将类的实例化操作延迟到子类中完成

(2)抽象工厂(Abstract Factory)模式：为一个产品族提供统一的创建接口。当需要这个产品族的某一系列的时候，可以从抽象工厂中选出相应的系列创建一个具体的工厂类

(3)建造者（Builder）模式：将复杂对象创建与表示分离，同样的创建过程可创建不同的表示。允许用户通过指定复杂对象类型和内容来创建对象，用户不需要知道对象内部的具体构建细节；
(4)单件（Singleton）模式：保证一个类有且仅有一个实例，提供一个全局访问点

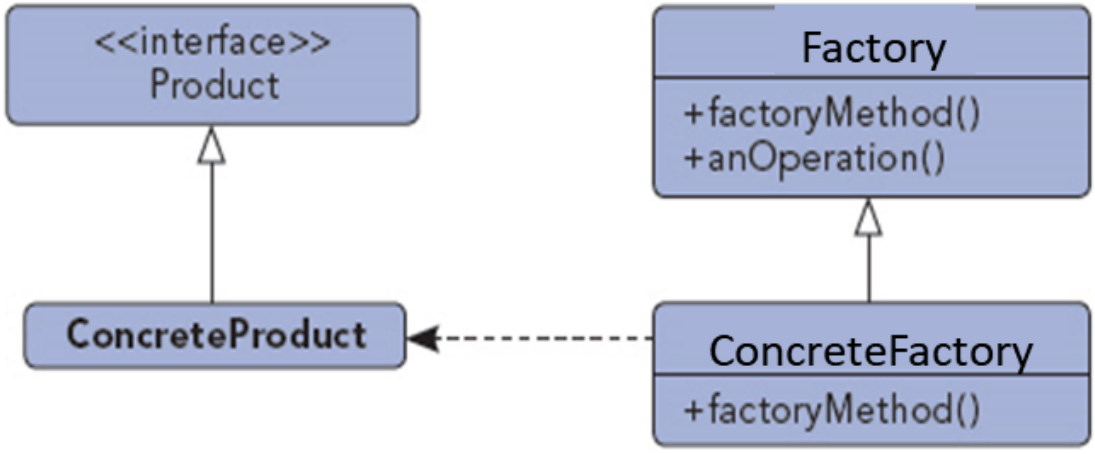
(5)原型（Prototype）模式：通过“复制”一个已经存在的实例来返回新的实例（不新建实例）。被复制的实例就是“原型”，这个原型是可定制的。原型模式多用于创建复杂的或者耗时的实例，因为这种情况下，复制一个已经存在的实例使程序运行更高效；或者创建值相等，只是命名不一样的同类数据

1. 工厂模式 Factory Method Pattern

『 定义一个用于创建对象的接口，让子类决定实例化哪一个类，使一个类的实例化延迟到其子类 』

特点

```
一个抽象产品类，可以派生出多个具体产品类
一个抽象工厂类，可以派生出多个具体工厂类
每个具体工厂类通常只能创建一个具体产品类的实例
```



（1）创建对象之前必须清楚所要创建对象的类信息，但个别情况下无法达到此要求，譬如打开一个视频文件需要一个播放器对象，但是用户可能不知道具体播放器叫什么名字，需要系统分派给这个视频文件一个合适的播放器，这种情况下用new运算符并不合适；

（2）许多类型对象的创造需要一系列步骤：需要计算或取得对象的初始设置；需要选择生成哪个子对象实例；在生成需要对象之前必须先生成一些辅助功能对象。在这些情况, 新对象的建立就是一个“过程”，而不仅仅是一个操作。为了能方便地完成这些复杂的对象创建工作，可引入工厂模式

(1.1)工厂模式的参与者

```
Product：表示抽象产品，定义产品的接口；
ConcreteProduct：表示具体产品，实现Product接口；
Factory：表示抽象工厂，声明工厂方法factoryMethod()，返回一个Product类型对象。 Factory也可以定义一个工厂方法的缺省实现，返回一个缺省ConcreteProduct对象；
ConcreteFactory：表示具体工厂，实现抽象工厂接口，重定义工厂方法，由客户端调用，返回一个ConcreteProduct实例
```

在工厂方法模式中，父类负责定义创建对象的公共接口，而子类负责生成具体的对象，这样做的目的是将类的实例化操作延迟到子类中完成，即由子类来决定究竟应该实例化（创建）哪一个类。

(1.2)工厂模式实例分析

✓一个日志管理器：设计日志记录类，支持记录的方法有

- ✓FileLog
- ✓EventLog

```
// LogFactory类
public abstract class LogFactory
 {
    public abstract Log Create();
}
```

```
//EventFactory类
public class EventFactory:LogFactory
 {
   public override EventLog Create()
   {
     return new EventLog();
   }
}
```

```
//FileFactory类
public class FileFactory:LogFactory
 {
   public override FileLog Create()
   {
     return new FileLog();
   }
}
```

(1.3)工厂模式客户端程序

```
public class App
{
    public static void Main(string[] args)
    {
        LogFactory factory = new EventFactory();
        //FileFactory factory = new FileFactory();

        Log log = factory.Create();

        log.Write();
    }
}
```

(1.4)工厂模式实例分析

（1）客户程序有效避免了具体产品对象和应用程序之间的耦合，增加了具体工厂对象和应用程序之间的耦合
（2）在类内部创建对象通常比直接创建对象更灵活
（3）工厂模式通过面向对象的手法，将具体对象的创建工作延迟到子类，提供了一种扩展策略，较好的解决了紧耦合问题

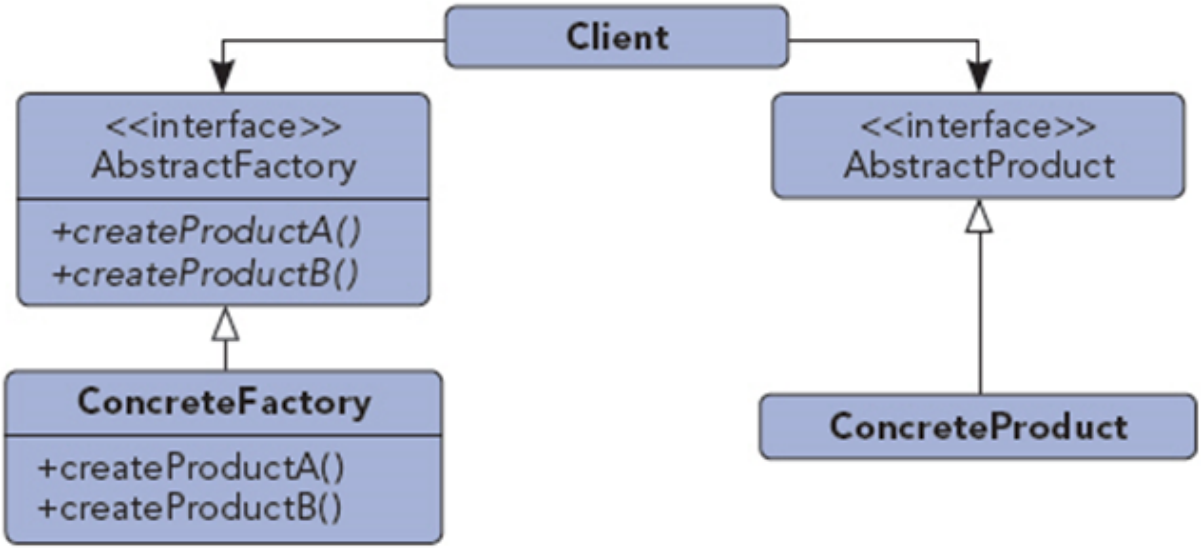
2. 抽象工厂模式 Abstract Factory Pattern

定义

『 为创建一组相关或相互依赖的对象提供一个接口，而且无需指定他们的具体类 』

特点

多个抽象产品类，每个抽象产品类可以派生出多个具体产品类
一个抽象工厂类，可以派生出多个具体工厂类
每个具体工厂类可以创建多个具体产品类的实例



(2.1)抽象工厂模式的参与者

```
AbstractFactory：声明创建抽象产品对象的操作接口
ConcreteFactory：实现创建具体对象的操作
AbstractProduct：为一类产品对象声明一个接口
ConcreteProduct：定义一个被具体工厂创建的产品对象
```

实例：

需要设计一个花园布局
花园有三种风格：典雅型、实用型和懒人型
花园中有3个位置需要种植植物：花台、墙角和花园中心

风格/位置	花台	中心	墙角
典雅型	郁金香	榕树	兰草
实用型	葡萄	石榴	丝瓜
懒人型	月季	茶花	竹子

在软件系统中，经常面临“一系列相互依赖对象”的创建工作，由于需求变化，这“一系列相互依赖的对象”也要改变，如何应对这种变化呢？如何像工厂模式一样绕过常规的”new”，提供一种“封装机制”来避免客户程序和这种“多系列具体对象创建工作”的紧耦合？

方法

可以将这些对象一个个通过工厂模式来创建。但是，既然是一系列相互依赖的对象，它们是有联系的，每个对象都这样解决，如何保证他们的联系呢？

实例
Windows桌面主题，当更换一个桌面主题的时候，系统的开始按钮、任务栏、菜单栏、工具栏等都

变了，而且是一起变的，他们的色调都很一致，类似这样的问题如何解决呢?

应用抽象工厂模式，是一种有效的解决途径

结构型模式

结构型模式讨论的是类和对象的结构，它采用继承机制来组合接口或实现（类结构型模式），或者通过组合一些对象来实现新的功能（对象结构型模式）

（1）适配器(Adapter)模式：将一个类的接口适配成用户所期待的接口。一个适配器允许因为接口不兼容而不能在一起工作的类工作在一起，做法是将类自己的接口包装在一个已存在的类中；

（2）组合（Composite）模式：定义一个接口，使之用于单一对象，也可以应用于多个单一对象组成的对象组；

（3）装饰（Decorator）模式：给对象动态添加额外的职责，就好像给一个物体加上装饰物，完善其功能；

（4）代理（Proxy）模式：在软件系统中，有些对象有时候由于跨越网络或者其他障碍，而不能够或者不想直接访问另一个对象，直接访问会给系统带来不必要的复杂性，这时候可以在客户程序和目标对象之间增加一层中间层，让代理对象来代替目标对象打点一切，这就是代理（Proxy）模式

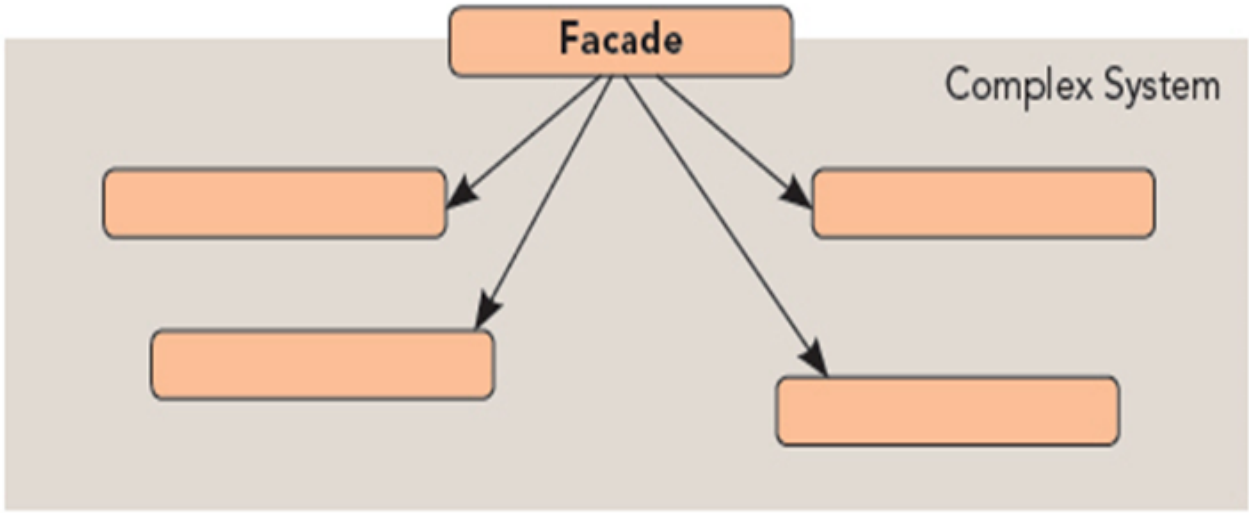
（5）享元（Flyweight）模式：Flyweight是一个共享对象，它可以同时在不同上下文（Context）使用；

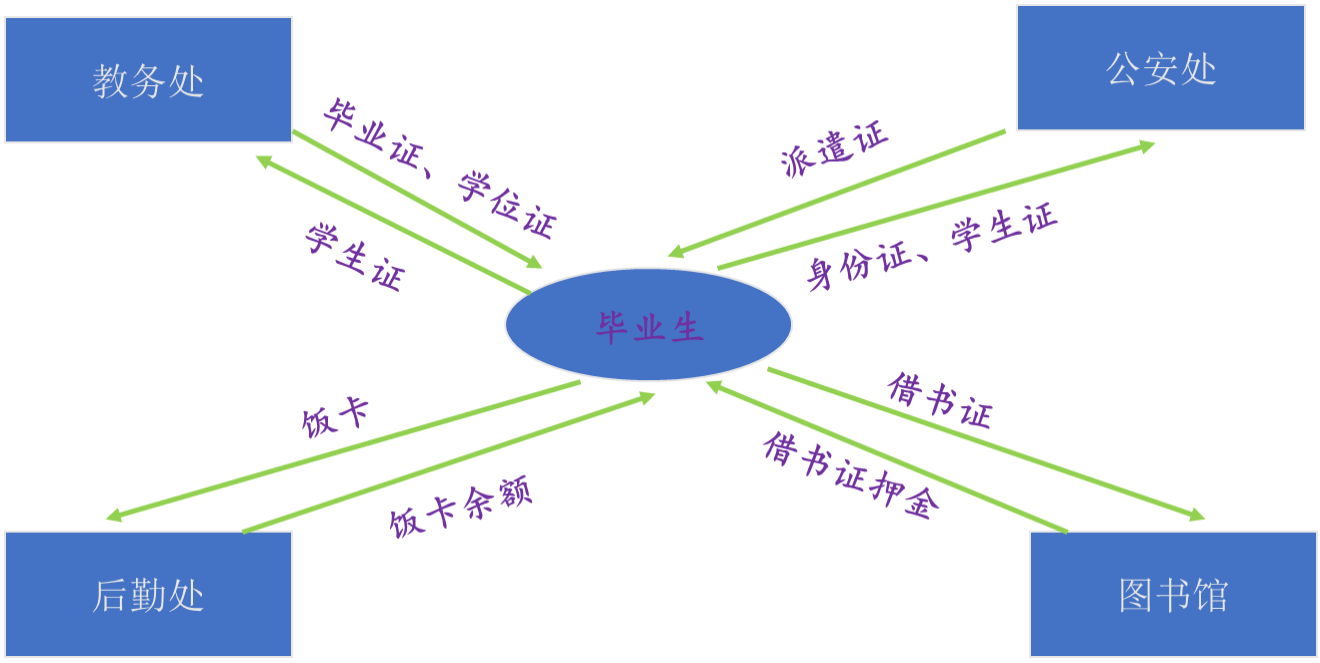
（6）外观（Facade）模式：外观模式为子系统提供了一个更高层次、更简单的接口，从而降低了子系统的复杂度，使子系统更易于使用和管理。外观承担了子系统中类交互的责任；

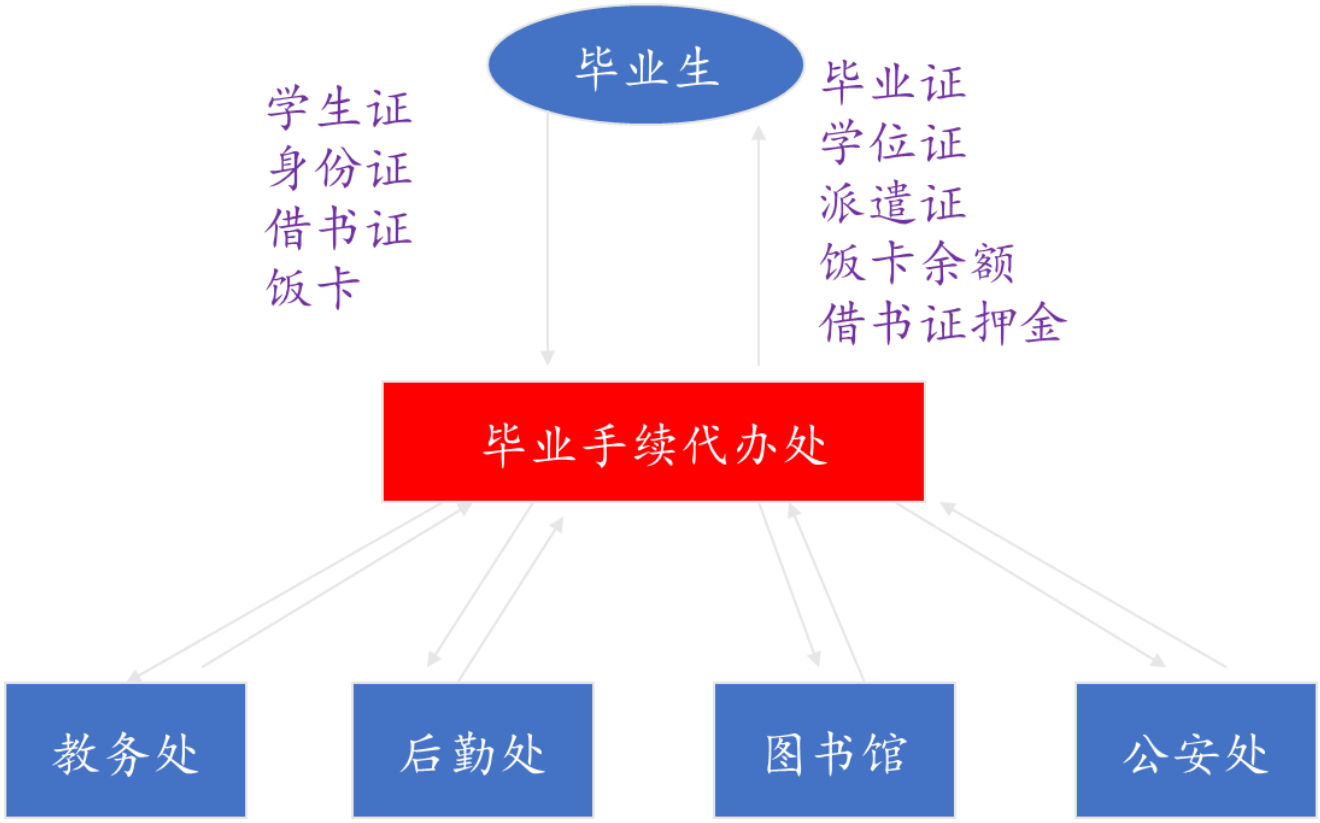
（7）桥梁（Bridge）模式：桥梁模式的用意是将问题的抽象和实现分离开来实现，通过用聚合代替继承来解决子类爆炸性增长的问题；

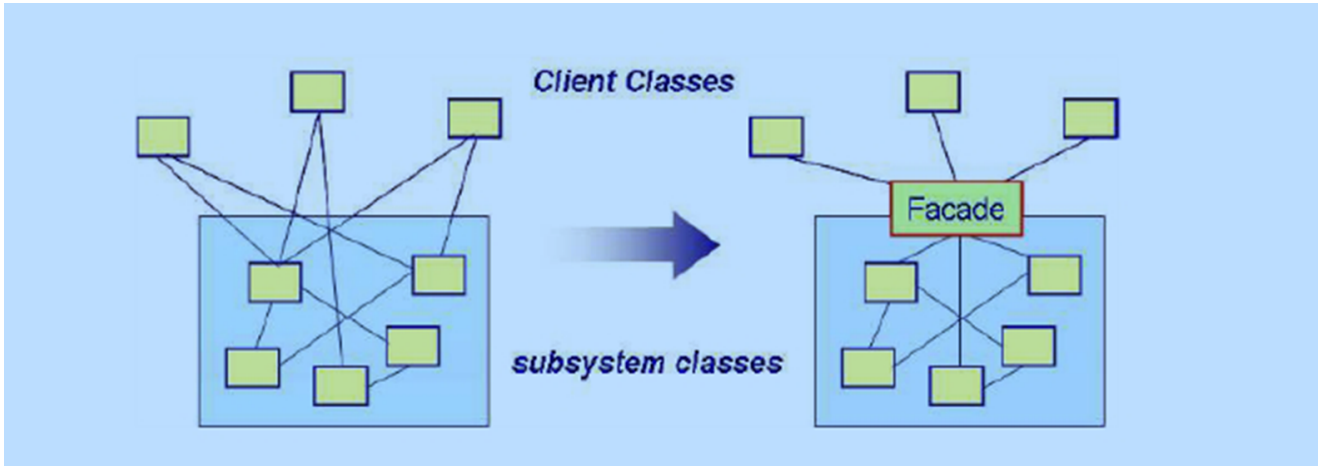
外观（Facade）模式

『 要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行。门面模式提供一个高层的接口，使得子系统更易于使用 』


Facade
Complex System


教务处
公安处
毕业证、学位证
学生证
派遣证
身份证、学生证
毕业生
饭卡
饭卡余额
借书证
借书证押金
后勤处
图书馆





(2.1)外观（Facade）模式的意图与适用性

（1）提供了一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用；
（2）解除客户程序与抽象类具体实现部分的依赖性，有利于移植和更改；
（3）当需要构建层次结构的子系统时，使用Façade模式定义每层的入口点。如果子系统间相互依赖，他们只需通过Façade进行通讯；
（4）外观模式的本质是让接口变得更简单.

(2.2)外观（Facade）模式的结构

```
Façade
        知道哪些子系统类负责处理请求
        将客户的请求代理给适当的子系统对象
Subsystem Classes
```

实现子系统的功能
处理由Façade 对象指派的任务
没有Façade的任何相关信息

(2.3)外观（Facade）模式的效果分析

对客户端屏蔽子系统组件，减少客户端使用对象数目
实现了子系统与客户之间松耦合的关系，使得子系统组件的变化不会影响到客户
不限制客户应用子系统类

行为型模式

着力解决的是类实体之间的通讯关系，希望以面向对象的方式描述一个控制流程

(1)模版（Template ） 模式：定义了一个算法步骤，并允许子类为一个或多个步骤提供实现。子类在不改变算法架构的情况下，可重新定义算法中某些步骤；

(2)观察者（Observer）模式：定义了对象之间一对多的依赖，当这个对象的状态发生改变的时候，多个对象会接受到通知，有机会做出反馈；

(3)迭代子（Iterator）模式：提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示；

(4)责任链（Chain of Responsibility）模式：很多对象由每一个对象对其下一个对象的引用而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。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使系统可以在不影响客户端的情况下动态的重新组织链和分配责任

(5)状态（State）模式：允许一个“对象”在其内部状态改变的时候改变其行为，即不同的状态，不同的行为

(6)访问者（Visitor）模式：表示一个作用于某对象结构中的各元素的操作。可以在不改变各元素的类的前提下定义作用于这些元素的新操作

(7)解释器(Interpreter) 模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子

(8)中介者（Mediator）模式：用一个中介对象来封装一系列的对象交互

(9)策略（Strategy）模式：定义一组算法，将每个算法都封装起来，并且使它们之间可以互换。策略模式使这些算法在客户端调用它们的时候能够互不影响地变化.

(10)备忘录（Memento）模式：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态；

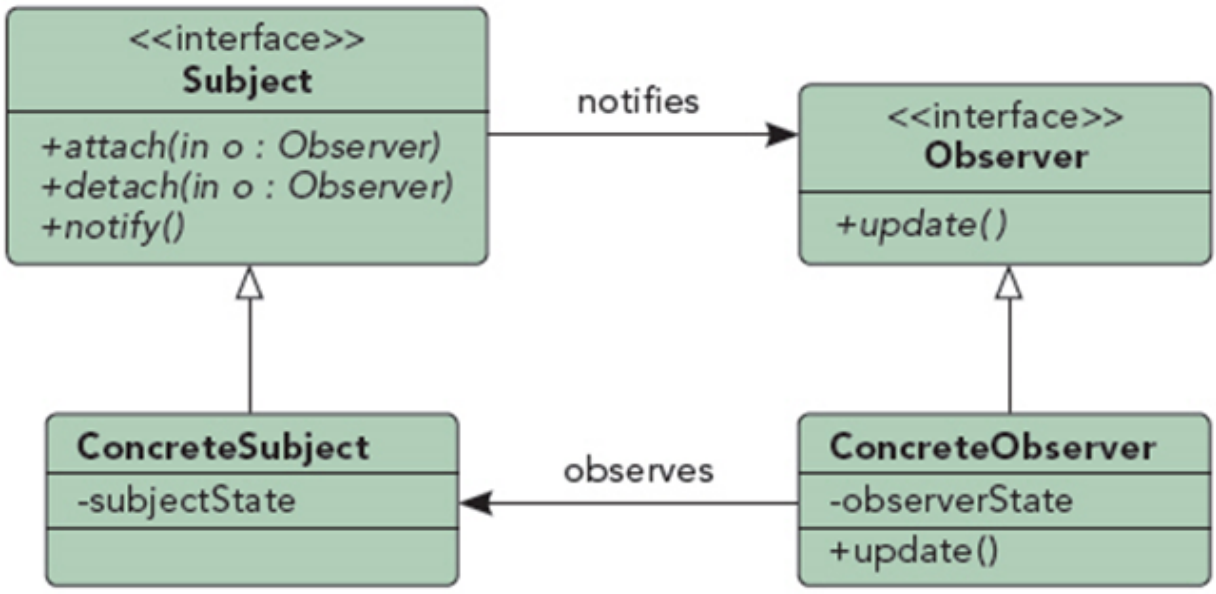
(11)命令（Command）模式：将请求及其参数封装成一个对象，作为命令发起者和接收者的中介，可以对这些请求排队或记录请求日志，以及支持可撤销操作

观察者（Observers）模式

『 定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新 』

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。



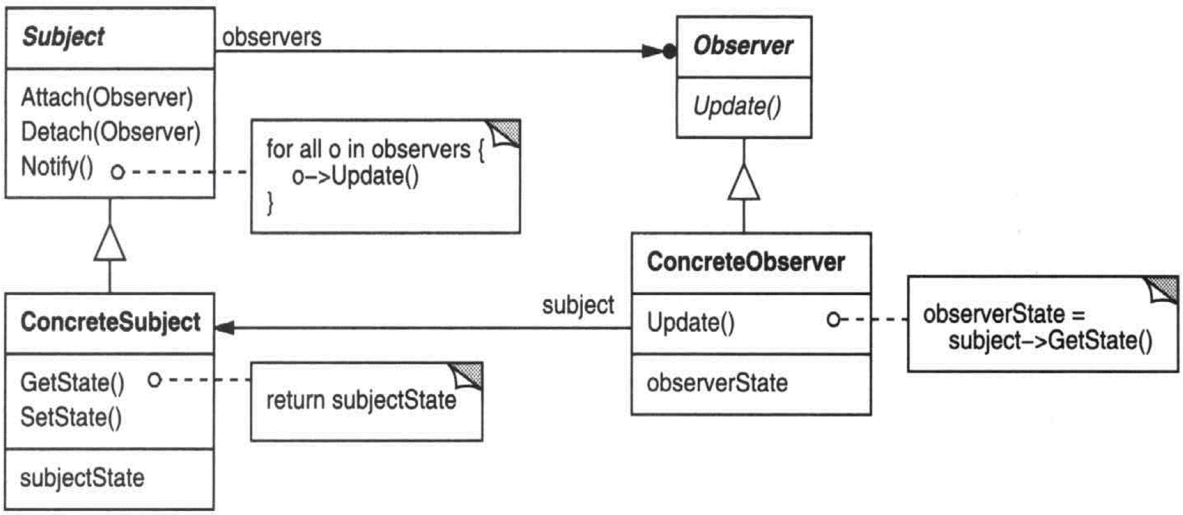
意图

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

（1）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；
（2）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；
（3）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；

(2.1)观察者（Observers）模式的结构



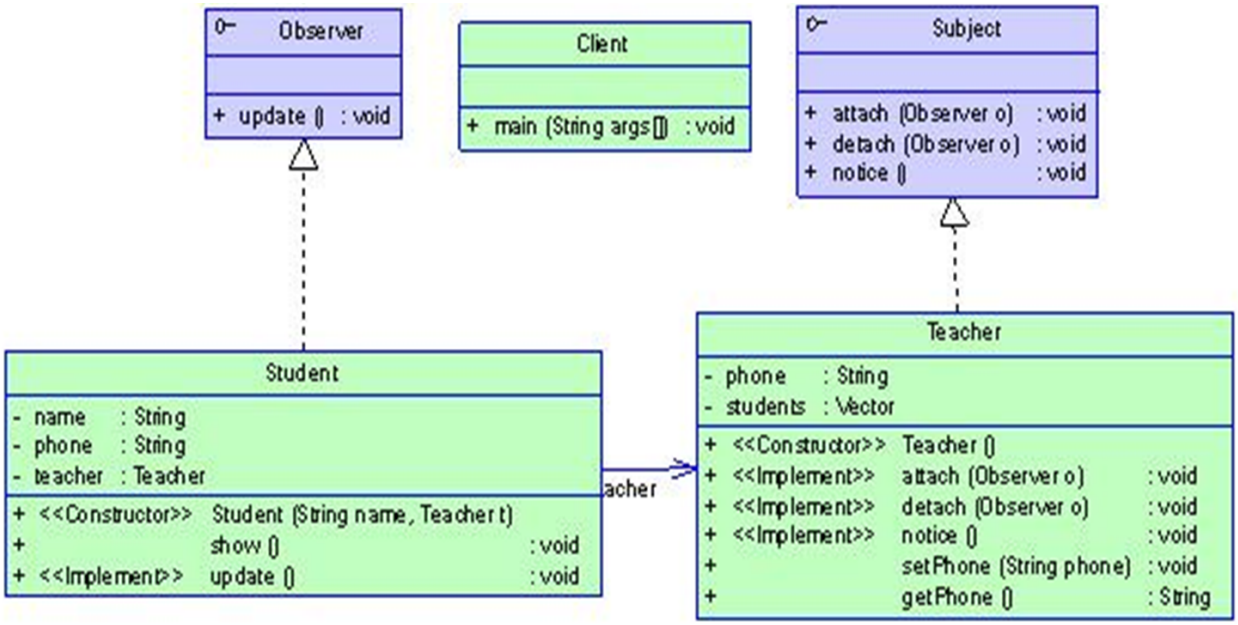
(2.2)观察者（Observers）模式的参与者

```
Subject：抽象的主题，即被观察的对象
Concrete Subject：具体被观察对象
Observer：抽象的观察者
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

(2.3)观察者（Observers）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。
```



(2.4)观察者（Observers）模式的效果分析

（1）应用场景对象同步更新，而且其他对象的数量动态可变
（2）对一个对象状态的更新，需要该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
（1）Subject和Observer之间是松偶合的，可以各自独立改变
（2）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
（3）高内聚、低偶合

缺陷
（1）松偶合导致代码关系不明显，有时可能难以理解
（2）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

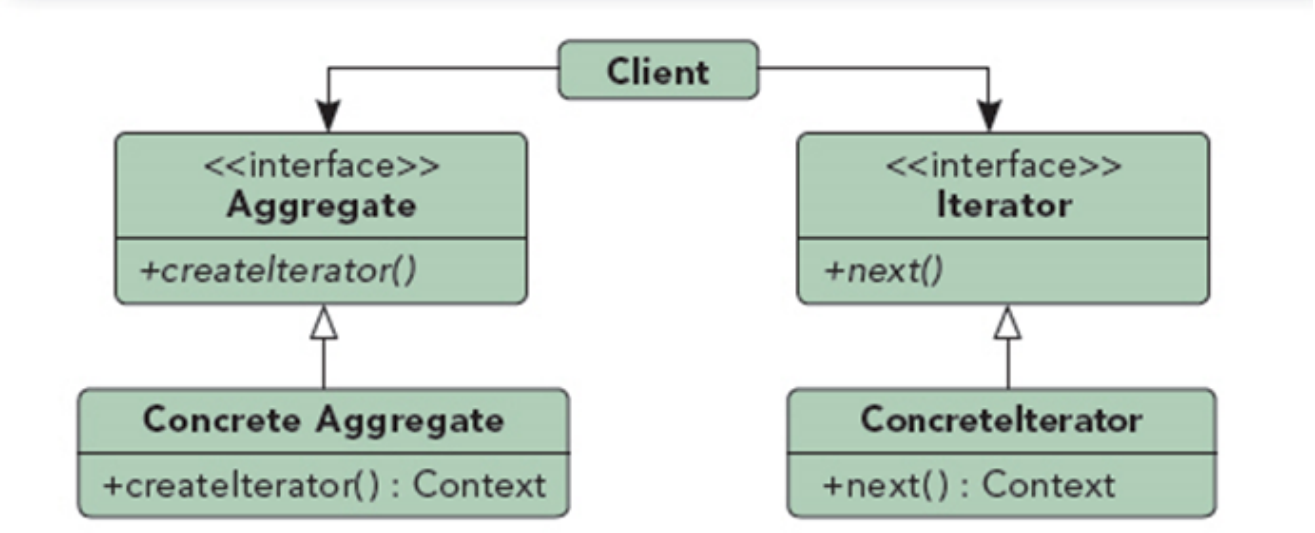
迭代子（Iterator）模式

『 它提供一种方法访问一个容器对象中各个元素，而又不需暴露该对象内部细节 』

由来
将对象职责分离，最大限度减少彼此之间的耦合程度，从而建立一个松散耦合的对象网络;

集合对象拥有两个职责：一是存储内部数据；二是遍历内部数据。从依赖性看，前者为对象的根本属性，而后者既是可变化的，又是可分离的。

可将遍历行为分离出来，抽象为一个迭代器，专门提供遍历集合内部数据对象行为。



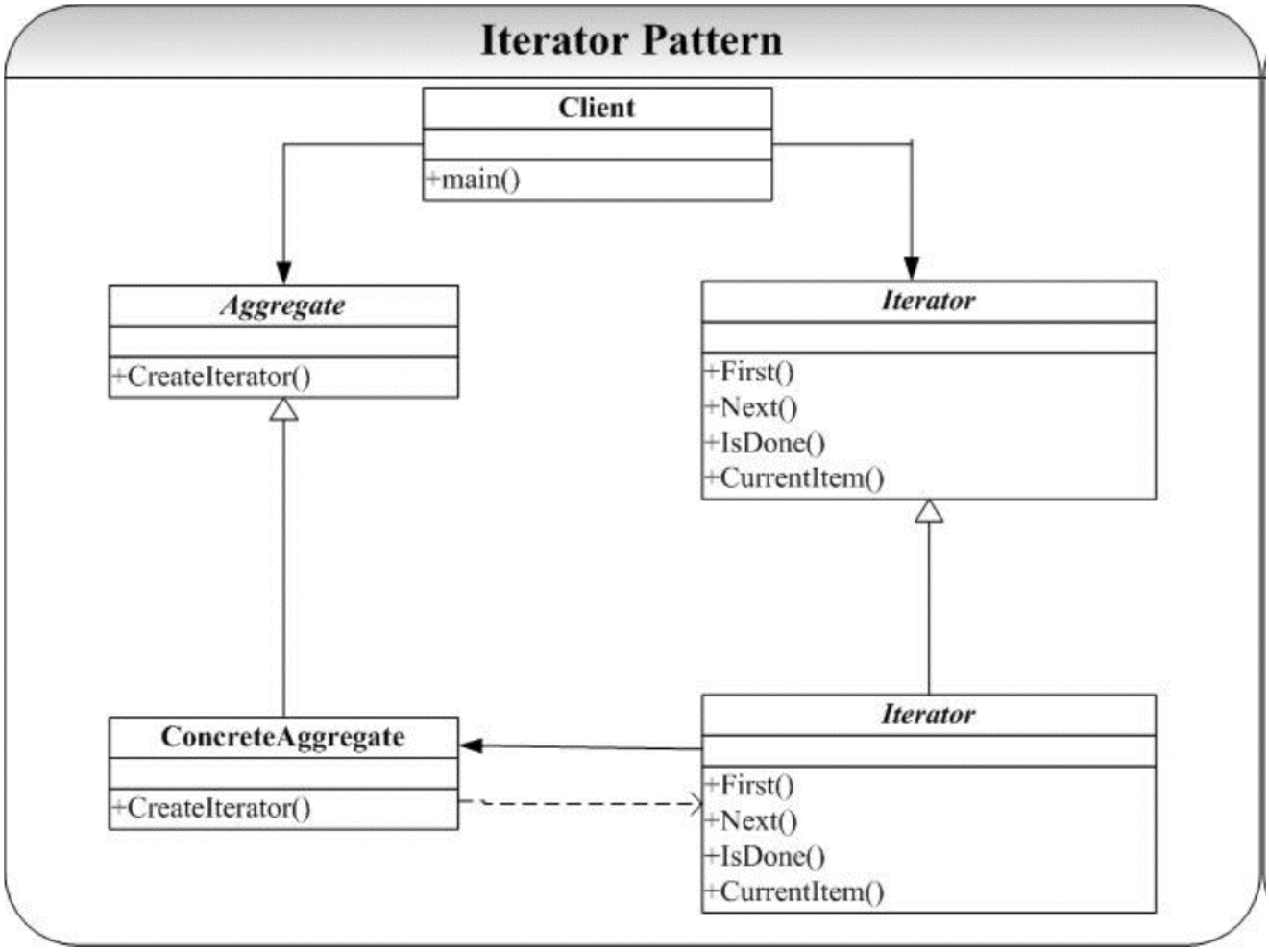
意图

迭代子模式的目的是设计一个迭代器，提供一种可顺序访问聚合对象中各个元素的方法，但不暴露该对象内部表示

适用场合

访问一个聚合对象的内容而无需暴露其内部表示
支持对聚合对象的多种遍历
为遍历不同的聚合结构提供一个统一接口（支持多态迭代）

(3.1)迭代子（Iterator）模式的结构



(3.2)迭代子（Iterator）模式的参与者

Iterator：定义访问和遍历元素的接口
Concrete Iterator：实现迭代器接口
Aggregate：定义创建迭代器对象的接口
Concrete Aggregate：实现创建迭代器的接口，返回具体迭代器的一个实例

常见一个课程集合，集合中每一个元素都是课程对象，然后编写一个迭代器，将每个课程对象的信息打印出来

```
public class Course
{   private String name;
    public Course(String name)
    {   this.name =name;       }
    public String getName()
    {   return name;              }
}
//创建自定义迭代器
public interface Iterator<E>
{  E next();
   boolean hasNext();
}
//实现迭代器接口
public class ConcreteIterator<E> implements Iterator<E>
{  private List<E> list;
   private int cursor;//光标
   private E element;
   public ConcreteIterator(List list)
   {  this.list = list;}
   public E next()
   {  System.out.print("当前位置"+cursor+" : ");
      element = list.get(cursor);
      cursor++;
      return element;
    }
   public boolean hasNext()
   {   if(cursor > list.size()-1)
       {  return fasle ;  }
        return true;
   }
}
//创新课程集合CourseAggregate接口
public interface CourseAggregate
{  void add(Course course);
   void remove(Course course);
   Iterator<Course> iterator();
}
```

```
//实现课程集合
public class ConcreteCourseAggegate implements CourseAggegate
{
     private List courseList;
     public ConcreteCourseAggegate()
     {  this.courseList= new ArrayList();  }
     public void add(Course course)
     {  courseList.add(course);  }
     public void remove(Course course)
     {  courseList.remove(course);  }
     public  Iterator<Course> iterator()
     {   return new ConcreteIterator(courseList);   }
}
public class test
{
   public static void main(String[] args)
   {    Course oo =new Course("面向对象");
        Course db =new Course("数据库");
        Course java =new Course("Java程序设计");
        Course se =new Course("软件工程");

        ConcreteCourseAggegate cca=new ConcreteCourseAggegate();
        cca.add(oo);   cca.add(java); cca.add(db); cca.add(se);
        printCourse(cca);

         cca.remove(db);
         printCourse();
   }
   public static void printCourse(ConcreteCourseAggegate tcca)
   {
        Iterator<Courese> iterator =tcca.iterator();
        while(!iterator.hasNext())
        {  Course course=iterator.next();
           System.out.println("课程名称"+course.getName())
         }
   }
}
```

(3.3)迭代子（Iterator）模式的效果分析

(1)每个聚集对象都可以有一个或者更多的迭代子对象，每一个迭代子的迭代状态可以彼此独立.

(2)遍历算简化了聚集的行为，迭代子具备遍历接口，聚集不必具备遍历接口.

(3)每一法客户端不必知道聚集对象的类型，通过迭代子就可以读取和遍历聚集对象。聚集内部数据发生变化不影响客户端程序被封装到迭代子对象中，迭代算法可以独立于聚集对象变化。

行为型模式总结

p模板模式：子类决定如何实现算法中的步骤

p观察者模式：让对象能够在状态改变时被通知

p迭代子模式：在对象的集合之中游走，而不暴露集合的实现

p责任链模式： 将请求的发送者和接收者解耦

p备忘录模式：保持关键对象的数据封装

p命令模式：封装请求成为对象

p状态模式：封装了基于状态的行为，并使用委托在行为间切换

p访问者模式：允许对组合结构加入新的操作而不改变结构本身

p中介者模式：集中相关对象之间复杂的沟通和通知方式

p策略模式：封装可以互换的行为，并使用委托来决定使用哪个

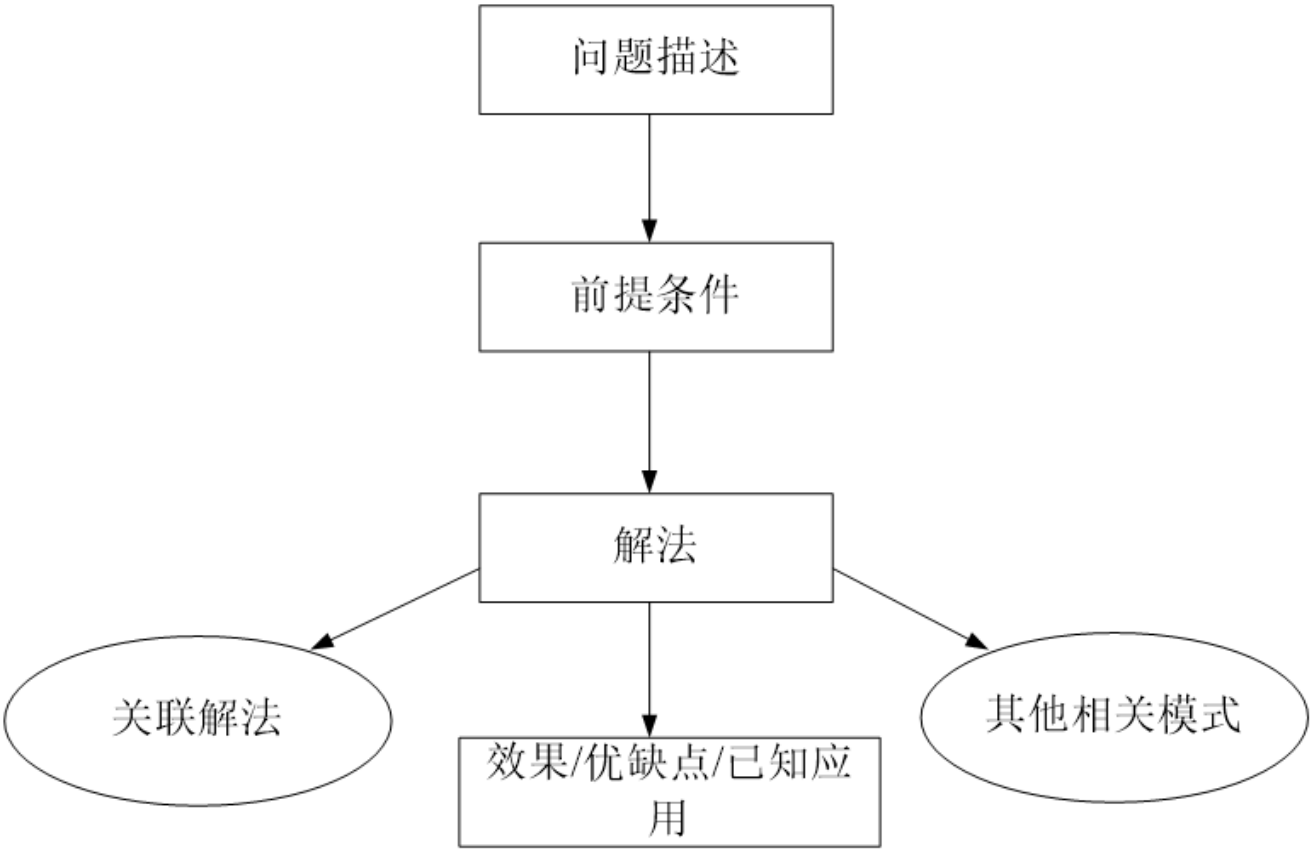
p解释器模式：定义语言文法，并解释该语言中的句子

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，即软件模式等于一定条件下的出现的问题以及解法。

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。



软件设计原则

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。

设计模式七大原则

程序员在编程时，应当遵守的原则，也是各种设计模式的基础（即设计模式为什么这样设计的依据）

1. 开闭原则 OCP

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。

遵循这个原则可以带来面向对象技术所声称的巨大好处（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。

OCP核心的思想是：

软件实体应该是可扩展，而不可修改的。

(两大特性)对扩展是开放的，而对修改是封闭的。

```
public interface Shape
 {
    public void draw();
}
public class Circle implements Shape
{
    public void draw()
    { … //从圆心开始画圆 }
}
```

画圆的方式发生变化?

```
public interface Shape
 {
    public void draw();
}
public class Circle implements Shape
{
    public void draw()
    { … //从圆正上方顶点顺时针开始画圆 }
}
```

```
public interface Shape
 {
    public void draw();
}
public class Circle implements Shape
{
    public void draw()
    { … //从圆心开始画圆 }
}
public class myCircle extends Circle
{
    public void draw()
    { … //从圆正上方顶点顺时针开始画圆 }
}
```

2. 里氏替换原则 LSP

定义：“若对于类型S的任一对象o1，均有类型T的对象o2存在，使得在T定义的所有程序P中，用o1替换o2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的使用其子类的对象。

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”

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，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。

```
public interface Shape
 {
    public void draw();
}
public class Circle implements Shape
{
    public void draw()
    { … //从圆心开始画圆 }
}
```

画圆的方式发生变化?

```
public interface Shape
 {
    public void draw();
}
public class Circle implements Shape
{
    public void draw()
    { … //从圆正上方顶点顺时针开始画圆 }
}
```

```
public interface Shape
 {
    public void draw();
}
public class Circle implements Shape
{
    public void draw()
    { … //从圆心开始画圆 }
}
public class myCircle extends Circle
{
    public void myDraw()
    { … //从圆正上方顶点顺时针开始画圆 }
}
```

18

3. 单一职责原则 SRP

定义：就一个类而言，应该只有一个导致其变化的原因。

单一职责包含两层含义：

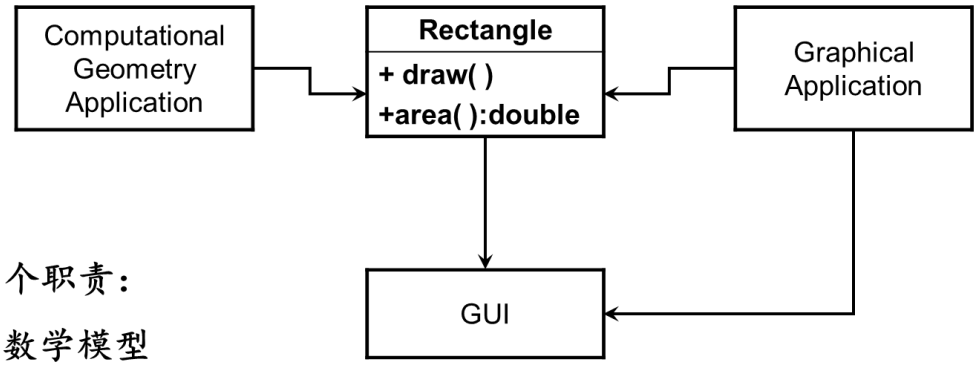
（1）一个模块只完成一个功能；

（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。



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

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

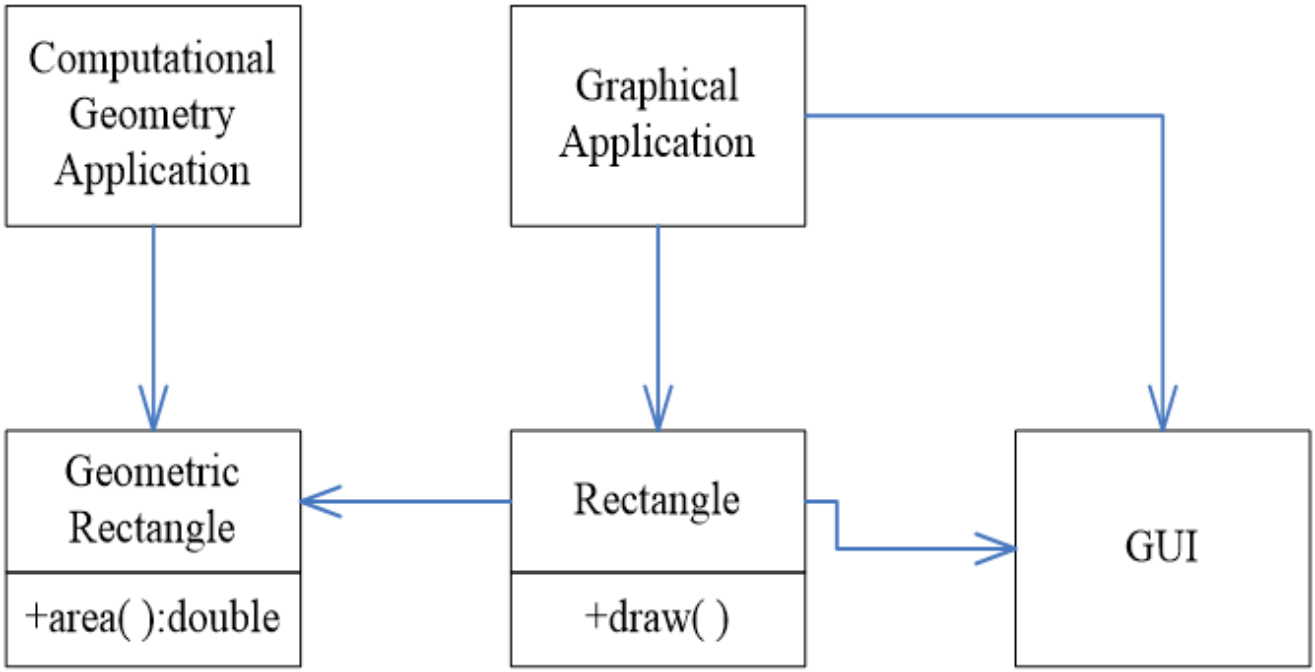
SRP是最简单地，但却是应用中最难处理和应用的一条法则。由于需要设计者发现类的不同职责，并结合实际考虑划分，经验与分析都需要。

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。

•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重新构建、测试以及部署·ComputationalGeometryApplication，否则其将莫名其妙的失败。

基于SRP原则的修改



4. 依赖倒置原则 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。
> 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。

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。

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。

```
public class Student  //底层
{
    public void selectOO()
    { System.out.println("选修面向对象课程"); }
    public void selectDB()
    { System.out.println("选修数据库课程");  }
}
public static void main(String[] arge) //高层
{
    Student zhanghua = new Student();
    zhanghua.selectOO();
    zhanghua.selectDB();
}
```

业务发生变化：
选修软件工程课程？

```
public class Student  //底层
{
    public void selectOO()
    { System.out.println("选修面向对象课程"); }
    public void selectDB()
    { System.out.println("选修数据库课程");  }
     public void selectSE()
    { System.out.println("选修软件工程课程"); }
}
public static void main(String[] arge) //高层
{
    Student zhanghua = new Student();
    zhanghua.selectOO();
    zhanghua.selectDB();
    zhanghua.selectSE();
}
```

```
public interface Course
{    public void selectCourse();   }
```

```
public class OOCourse implements Course
{
    public void selectCourse()
    { System.out.println("选修面向对象课程"); }
}
public class DBCourse implements Course
{
    public void selectCourse()
    { System.out.println("选修数据库课程"); }
}
public class SECourse implements Course
{
    public void selectCourse()
    { System.out.println("选修软件工程课程"); }
}
```

```
public class Student
{
    public void select(Course cc)
    { cc.selectCourse();  }
}
```

```
public static void main(String[] arge)
{
    Student zhanghua = new Student();
    zhanghua.select(new OOCourse());
    zhanghua.select(new DBCourse());
    zhanghua.select(new SECourse());
}
```

21

5. 接口隔离原则 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。

```
public interface Animal
{
    public void eat();
    public void fly();
    public void swim();
}
public class Bird implements Animal
{
    public void eat() { }
    public void fly() { }
    public void swim() { } //空函数体
}
public class Dog implements Animal
{
    public void eat() { }
    public void fly() { }  //空函数体
    public void swim() { }
}
```

小鸟不会游泳
小狗不会飞行?

```
public interface EatAnimal
{
    public void eat();
}
public interface FlyAnimal
{
    public void fly();
}
public interface SwimAnimal
{
    public void swim();
}
public class Bird implements EatAnimal,
FlyAnimal
{
    public void eat() { }
    public void fly() { }
}
public class Dog implements EatAnimal,
SwimAnimal
{
    public void eat() { }
    public void swim() { }
}
```



```
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）
为了实现上述新增功能，要求Door与一个已有的Timer对象进行交互

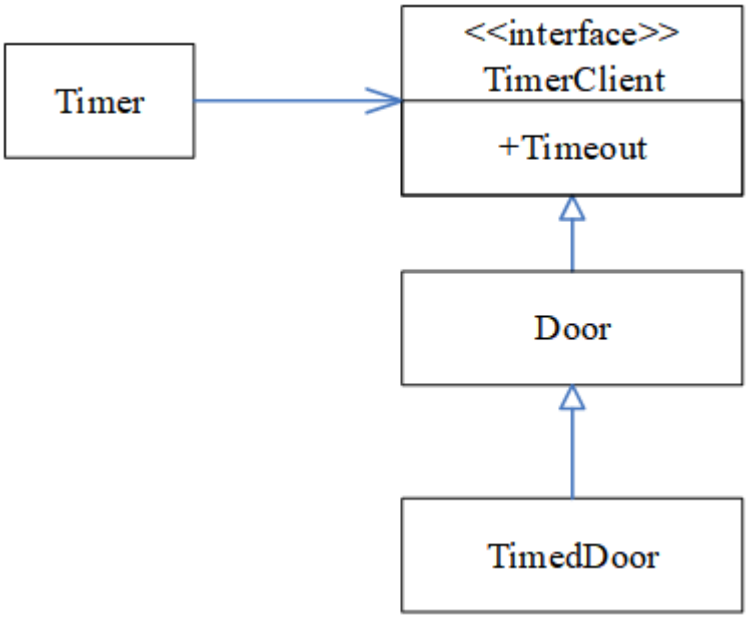
```
class Timer{
public:
    void Register(int timeout,TimerClient* client);
};

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：如何将TimerClient和TimedDoor联系起来?

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。



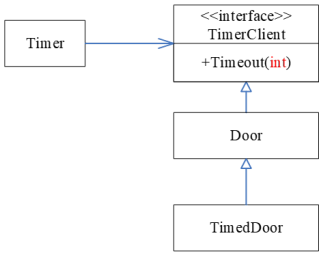
胖接口将导致SRP，LSP被违反，从而导致脆弱、僵化

•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 LoD

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。

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”。

狭义的迪米特原则

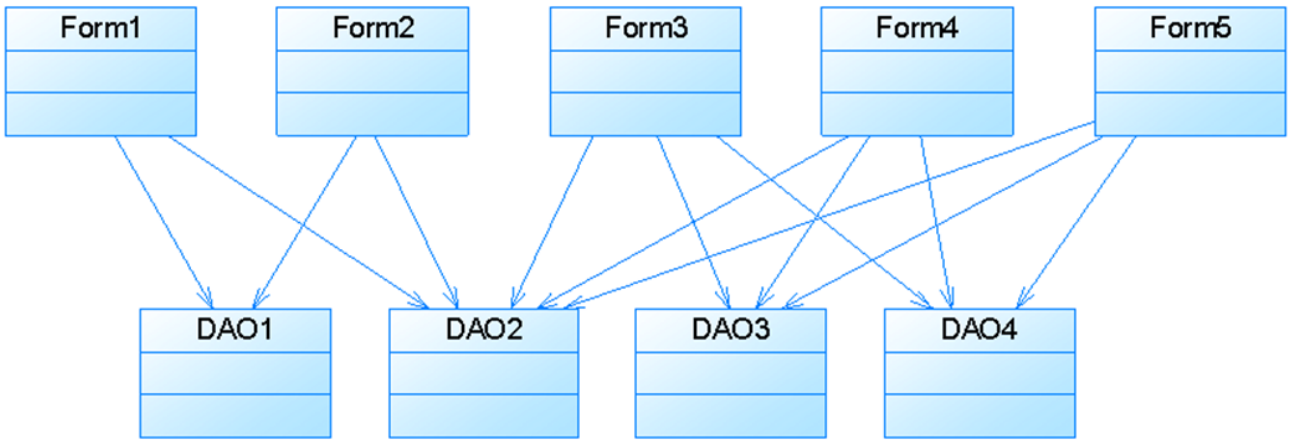
可以降低类之间的耦合，但是会在系统中增加大量的小方法并散落在系统的各个角落，它可以使一个系统的局部设计简化，因为每一个局部都不会和远距离的对象有直接的关联，但是也会造成系统的不同模块之间的通信效率降低，使得系统的不同模块之间不容易协调。

广义的迪米特原则

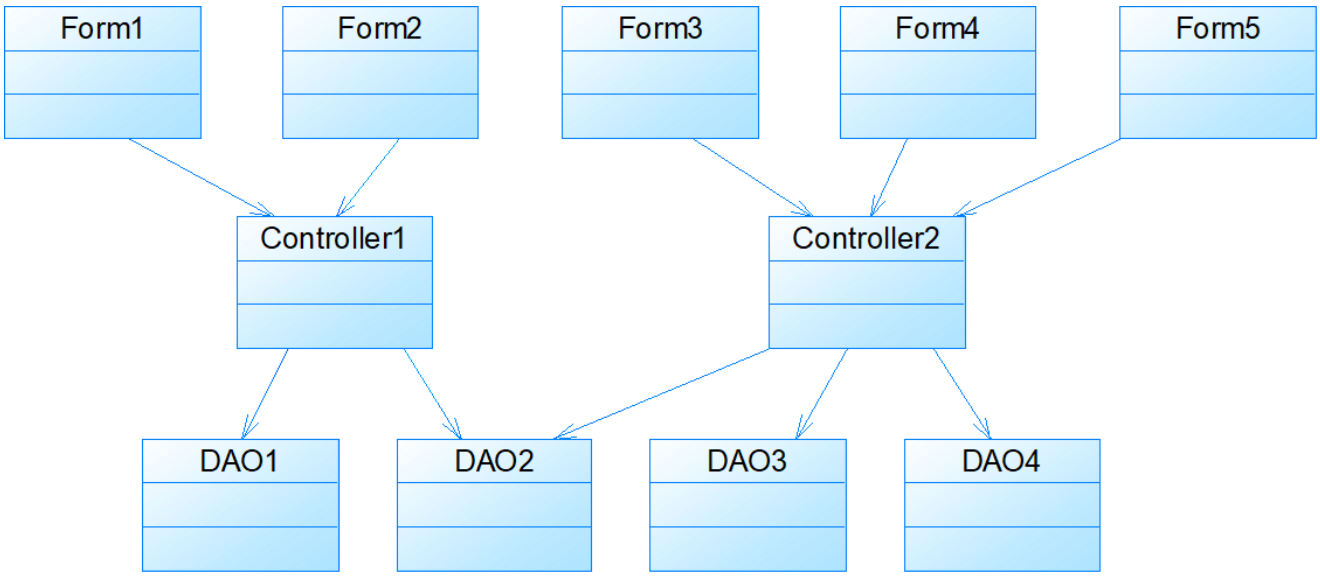
指对对象之间的信息流量、流向以及信息的影响的控制，主要是对信息隐藏的控制。信息的隐藏可以使各个子系统之间脱耦，从而允许它们独立地被开发、优化、使用和修改，同时可以促进软件的复用，由于每一个模块都不依赖于其他模块而存在，因此每一个模块都可以独立地在其他的地方使用。一个系统的规模越大，信息的隐藏就越重要，而信息隐藏的重要性也就越明显。

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。

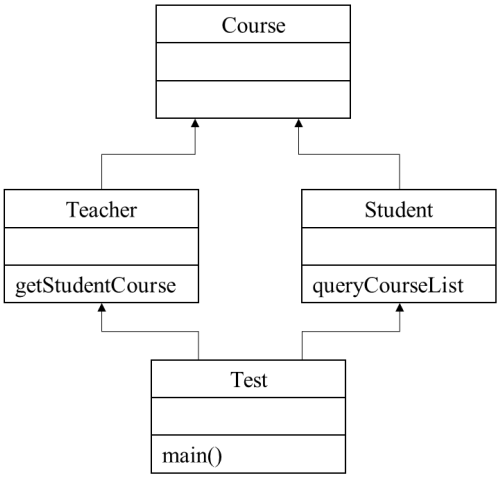


--



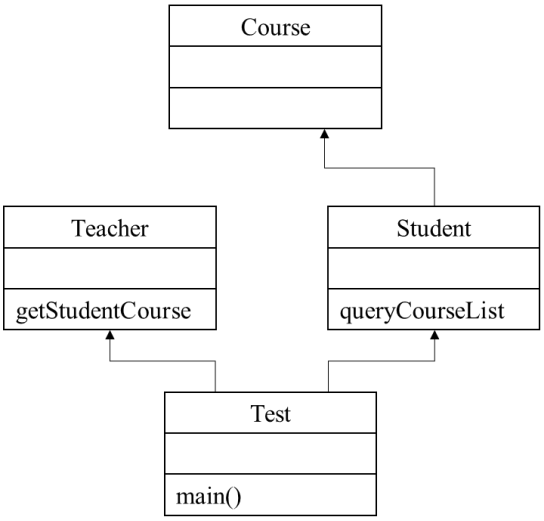

```
public class Course
{
 }
public class Student
{
    public void queryCouseList(List<Course> cl)
   { System.out.println("已选修"+cl.size()+"门课程")}
}
public class Teacher
{
    public void getStudentCourse(Student ts)
    {
      List<Course> studentCL=new ArrayList<Course>();
      //…….
      ts.queryCourseList(studentCL);
    }
}
public static void main(String[] arge)
{
    Teacher zhaoyun=new Teacher();
    Student zhanghua=new Student();
    zhaoyun.getStudentCourse(zhanghua);
}
```

赵云老师询问张华同学选修了多少门课程




```
public class Course
{
 }
public class Student
{
    public void queryCouseList(List<Course> cl)
   { System.out.println("已选修"+cl.size()+"门课程")}
}
public class Teacher
{
    public void getStudentCourse(Student ts)
    {
      List<Course> studentCL=new ArrayList<Course>();
      //…….
      ts.queryCourseList(studentCL);
    }
}
public static void main(String[] arge)
{
    Teacher zhaoyun=new Teacher();
    Student zhanghua=new Student();
    zhaoyun.getStudentCourse(zhanghua);
}
```

```
public class Course
{
 }
public class Student
{
    public void queryCouseList()
    {
      List<Course> studentCL=new ArrayList<Course>();
      //…….
      System.out.println("已选修"+studentCL.size()+"门课程")}
}
public class Teacher
{
    public void getStudentCourse(Student ts)
    {
      ts.queryCourseList();
    }
}
public static void main(String[] arge)
{
    Teacher zhaoyun=new Teacher();
    Student zhanghua=new Student();
    zhaoyun.getStudentCourse(zhanghua);
}
```




```
public class Course
{
 }
public class Student
{
    public void queryCouseList()
    {
       List<Course> studentCL=new ArrayList<Course>();
       //…….
       System.out.println("已选修"+studentCL.size()+"门课程")}
}
public class Teacher
{
    public void getStudentCourse(Student ts)
    {
       ts.queryCourseList();
    }
}
public static void main(String[] arge)
{
    Teacher zhaoyun=new Teacher();
    Student zhanghua=new Student();
    zhaoyun.getStudentCourse(zhanghua);
}
```

7. 合成复用原则 CARP（组合/聚合复用原则）

定义

尽量使用对象组合，而不是继承来达到复用的目的。

特点

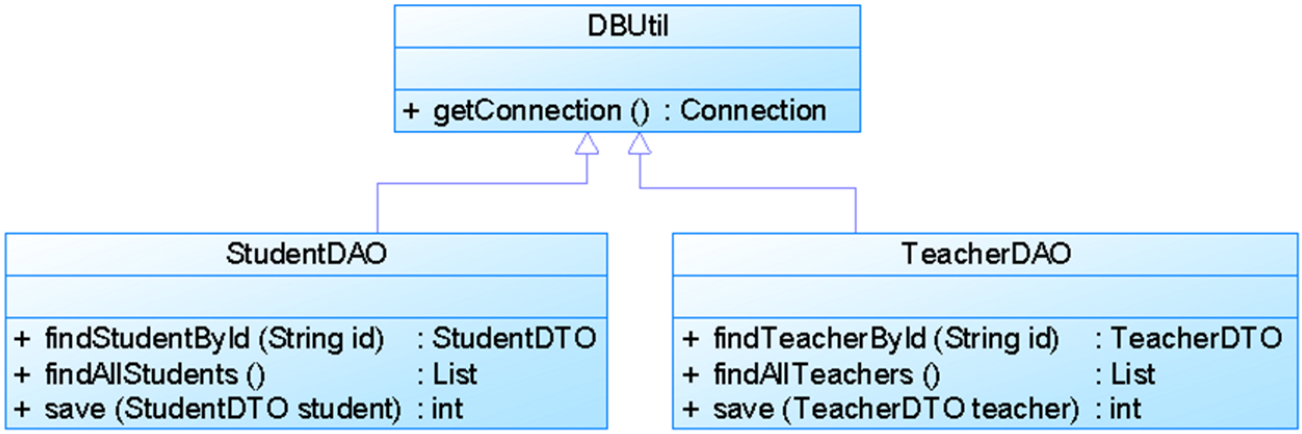
> 合成复用原则就是指在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用其已有功能的目的。
> 简言之：要尽量使用组合/聚合关系，少用继承。
> 在面向对象设计中，可以通过两种基本方法在不同的环境中复用已有的设计和实现，即通过组合/聚合关系或通过继承。
>
> 1、继承复用：实现简单，易于扩展。破坏系统的封装性；从基类继承而来的实现是静态的，不可能在运行时发生改变，没有足够的灵活性；只能在有限的环境中使用。（“白箱”复用 ）
>
> 2、组合/聚合复用：耦合度相对较低，选择性地调用成员对象的操作；可以在运行时动态进行。（“黑箱”复用 ）

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。

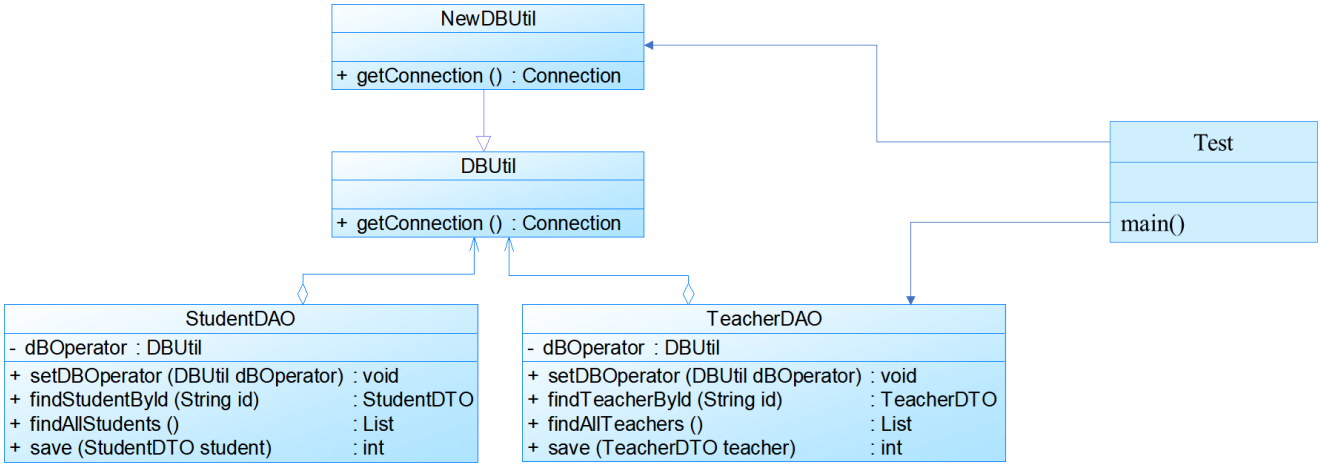
例子：



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大原则关系

开闭原则是总纲，要对扩展开放，对修改关闭。

里氏替换原则，不要破坏继承体系。

依赖倒置原则，要面向接口编程。

单一职责原则，类的职责要单一。

接口隔离原则，要在设计接口时精而简单。

迪米特法则，要降低耦合度。

合成复用原则，要多用组合或聚合关系，少用继承复用关系。

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	★★★★☆

软件项目管理

什么是项目

项目是指为创造移位产品获提供唯一服务所进行的临时性工作，它是以一套独特而相互联系的任务为前提，能够有效利用资源为了实现一个特定目标所进行的努力，它是在一定时间内满足一系列特定目标的多项相关工作的总称。

软件项目的确立

项目授权

项目章程

> 1. 项目章程是一个正式的文档，用于正式的认可一个项目的有效性，并指出目标和管理方向。
> 2. 授权项目经理来完成项目，从而保证项目经理可以组织资源用于项目活动。
> 3. 项目章程通常由项目发起人、出资人或高层管理人员签发。

要素：
1. 项目发起人及联系方式；
2. 项目经理及联系方式；
3. 项目目标；
4. 项目的业务情况（项目的开展原因）；
5. 项目的最高目标及可交付成果；
6. 团队开展工作的一般性描述；
7. 开展工作的基本事件安排；
8. 项目资源-预算-成员-供应商

软件生存期模型

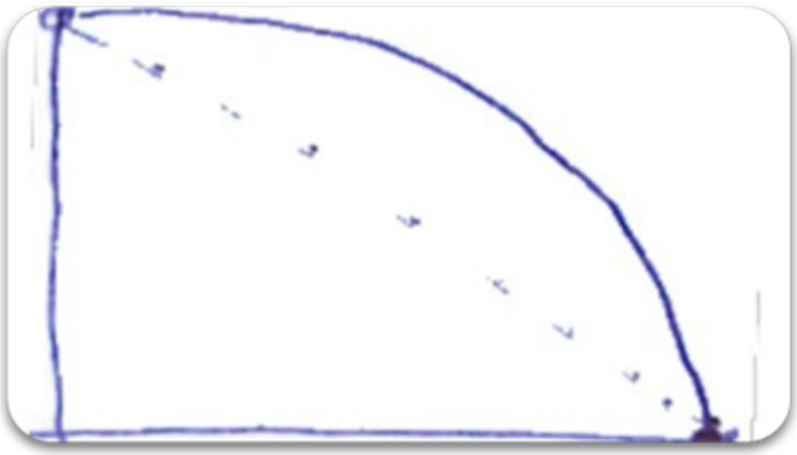
敏捷生存期模型

SCRUM方法

1．理想团队的燃尽图（小S）、
这种燃尽图说明该团队可以组织好工作。产品经理明白迭代的工作量，Scrum master 能够帮助团队完成任务。团队没有超负荷，并按时完成迭代工作。该团队可以正确估算自己的能力，迭代过程中也不需要改正。

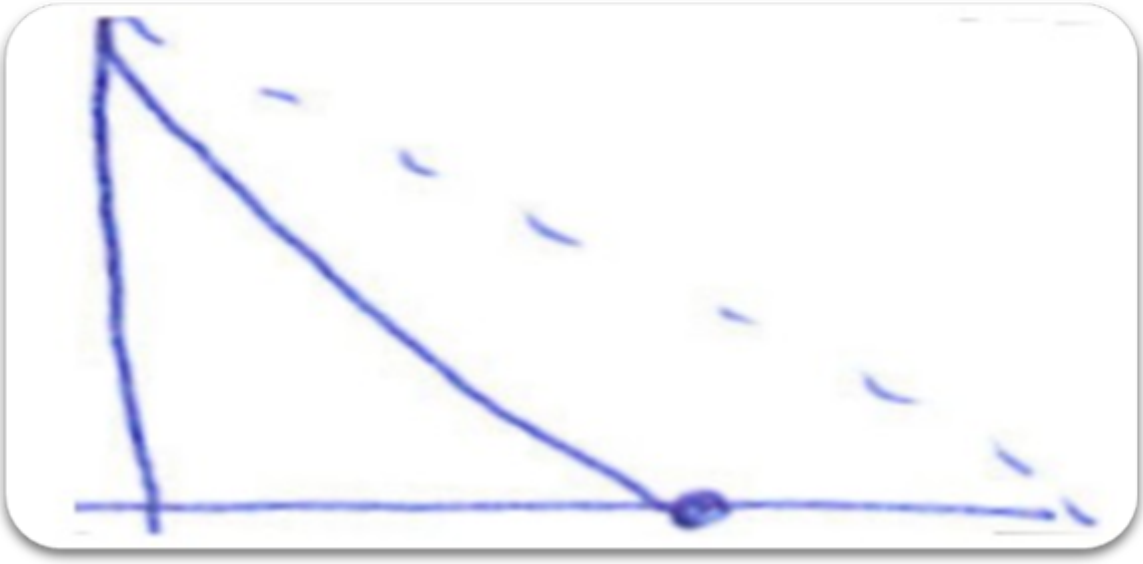
2．很好的团队的工作的燃尽图（大S）
这种燃尽图一般是具有丰富经验的团队的工作进度展示。团队按时完成工作，并且达到了迭代目标。这种团队可以完成工作，更重要的是可以适应迭代积压工作的情况。迭代后期，团队也有能力完成一些额外的工作。

3．不错的团队的工作的燃尽图
这是典型的工作进度燃尽图，在很多有经验的敏捷团队的工作中都可以看到。该燃尽图说明团队可以按时完成任务，调整以适应迭代中的积压任务，额外努力工作以完成任务。该团队需要自我反省，在迭代初期看到进度减慢就应该立即讨论如何变动计划、

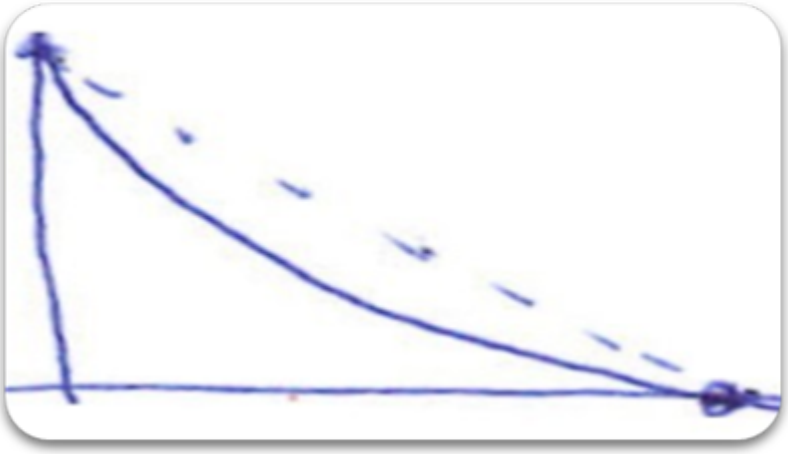


4．“太迟啦”团队的工作的燃尽图
这种燃尽图明显在说，“你们没有完成工作。”这种团队整个迭代过程都在迟到，没能合理调整工作。燃尽图还显示出团队没有完成需求，这些需求应该被进一步分解，或者挪到下一个迭代中

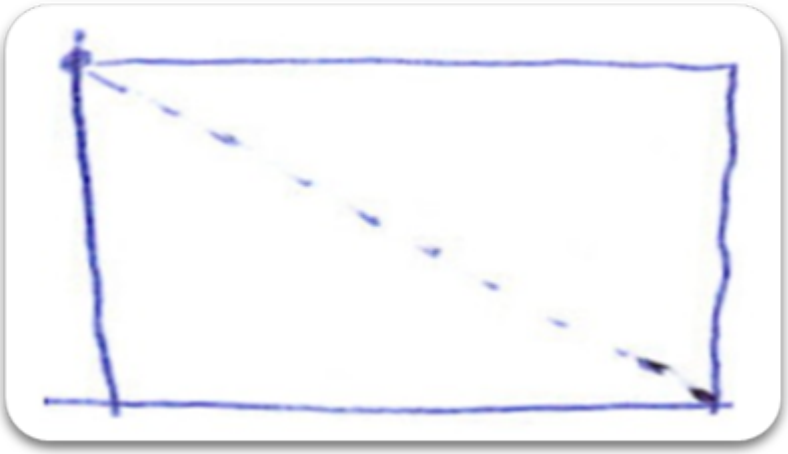
5．“太快啦”团队的工作的燃尽图
燃尽图显示团队比预期早很多完成任务。团队完成了需求，也没有继续完成其他任务即使团队有时间和精力这么做。这种情形下，需求可能被高估了，所以团队提前完成了任务。团队的工作速度没有被合理的估算。



6. “休息一下吧”团队的工作的燃尽图
团队的工作进度如果如该燃尽图所示，那么问题就来了。问题在于对工作复杂度的高估，这使得工作的完成比迭代初期预估的要早。Scrum master应该及早找出问题所在没要求产品经理给团队跟过的工作。即使需求被高估，团队至少应该继续做下一个迭代的任务

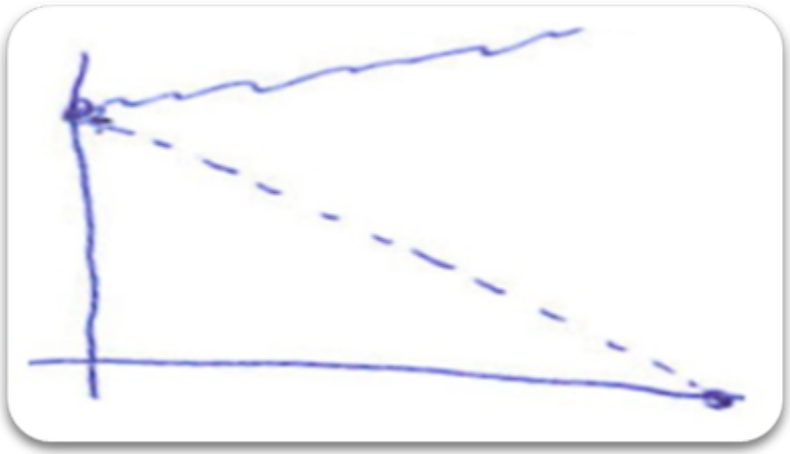


7. “管理层要来了”团队的工作的燃尽图
这种团队可能没有更新自己的工作进度。里一种情况可能是产品经理增加了一些已经完成的工作，所以燃尽图时机工作曲线是直线。



8．“你要上天吗”团队的工作的燃尽图

团队第一个迭代一般来说都是这种燃尽图。这种情况是成功之母，很明显团队没有完成任务。每天都有需求或任务添加到迭代工作中来却没有记录任何工作记录。另一个原因可能是迭代中的任务不断地被重新估算



软件需求概述

需求的3层次（需求规格说明书)

（1）业务需求

1．反映企业、组织对软件系统的高层次目标需求，也就是说是软件需求的建设目标。通常这一目标体现在两个方面。

2．问题：解决企业、组织运作过程中遇到的问题。

3．机会：抓住外部环节变化（业务、技术）所带来的机会，以便为企业带来新发展。

4．作用意义：业务需求的提出人通常是企业、组织的高层管理人员。业务需求是彻底从业务角度描述的，是指导软件开发的高层需求。明确地定义业务需求，将给整个团队指出努力地方向，这整个开发活动将有积极意义。

5．业务需求建立的时间：往往是在项目立项阶段完成，通常体现在战略规划报告或立项建议报告中。

（2）用户需求

指描述用户使用软件需要完成什么任务，怎么完成的需求。通常是在业务需求定义的基础上通过用户访谈、调查，对用户使用的场景进行整理，从而建立用户角度的需求。用户需求是需求捕获的结果。

1．用户需求理解

用户需求是从用户角度描述的系统功能需求和非功能需求，通常只涉及系统的外部行为，而不涉及系统的内部特性。

2．用户需求的描述

1．原则：应该易于用户的理解。一般不采用技术性很强的语言，而是采用自然语言和直观图形相结合的方式进行描述。

2．问题：自然语言表达容易含糊和不准确。

（3）软件需求（系统需求）

1．用户对软件系统行为的期望，一系列的行为联系在一起可以帮助用户完成任务，满足业务需求

2．软件需求可以直接映射为系统行为，定义了系统中需要实现的功能，描述了开发人员需

要实现什么

3．将用户需求转化为软件需求的过程是一个复杂的过程

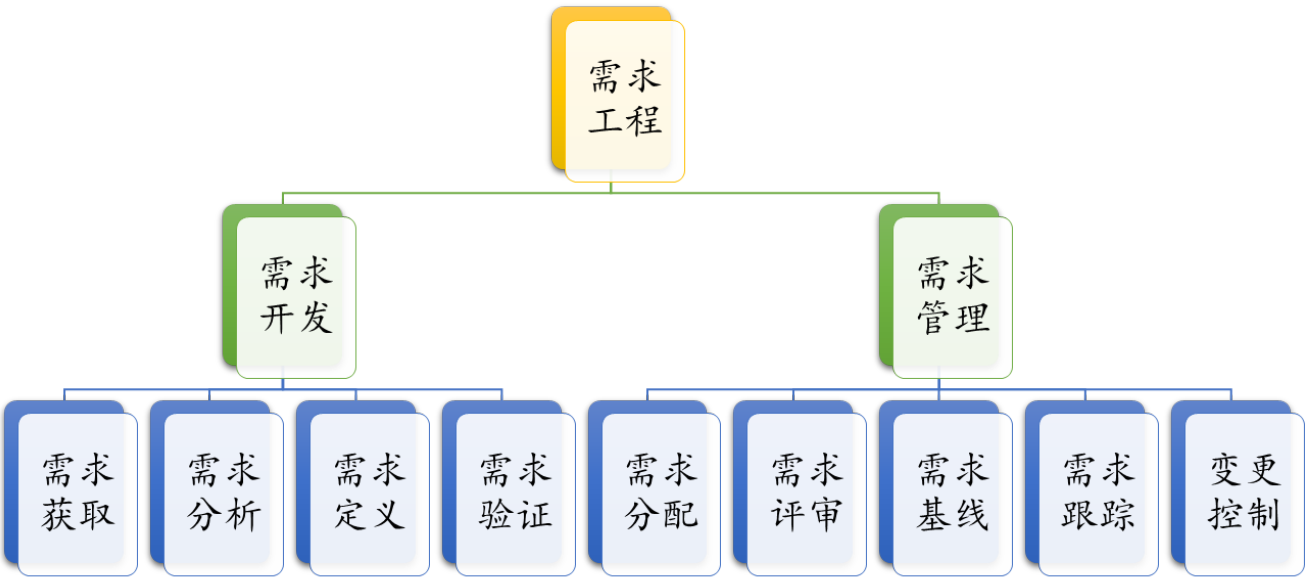
1．需要分析问题领域及其特性，从中发现问题域和计算机系统的共享知识，建立系统的知识模型；

2．将用户需求部署到系统模型，即定义系列的系统行为，让它们联合起来实现用户需求，每一个系统行为即为一个系统需求。

3．是需求工程当中最为重要的需求分析活动，又称建模与分析活动。

需求工程

需求工程基本活动



需求定义的任务与问题分析方法

问题分析的方法

1. 鱼骨图（石川图）

1．用于定性分析，可追溯问题根源

2．优点

1．使分析人员能将问题的原因而不是症状放在首位

2．提供了一种运用集体智慧解决问题的方法

3．直观、简单、易于操作

3．结合头脑风暴，由团队完成

帕累托分析

1．帕累托分析应用就是根据鱼骨图分析的结果，通过收集相关统计资料，以直方图的方式显示问题的相对频度或者大小高低等定量结果。

2．帕累托分析可以帮助我们做出这样的定量分析

需求定义的产物与需求范围

需求定义的产物和要素

产物：战略规划报告（前景及范围文档、项目综述与愿景文档）

需求获取

主要活动

1. 研究应用背景，建立初始的知识框架
2. 根据获取的需要，采用必要的获取方法和技巧
3. 先行确定获取的内容和主题，设定场景
4. 分析用户的高（深）层目标、理解用户意图
5. 进行涉众分析，针对涉众的特点开展工作

主要方法

（1）用户访谈

话题类型1：开放式话题
优点：
1. 让被会见者感到自在；
2. 会见者可以收集被会见者使用的词汇，这能反应他的教育、价值标准、态度和信念；
3. 提供丰富的细节；
4. 对没采用的进一步的提问有启迪作用；
5. 让被会见者更感兴趣；
6. 容许更多的自发性；
7. 会见者可以在没有太多准备的情况下进行面谈。

缺点：
1. 提此类问题可能会产生太多不相干的细节；
2. 面谈可能失控；
3. 开放式的回答会花费大量的时间才能获得有用的信息量；
4. 可能会使会见者看上去没有准备。

话题类型2：封闭式话题：
答案有基本的形式，被会见者的回答是受到限制的
优点：
1. 节省时间；
2. 切中要点；
3. 保持对面谈的控制；
4. 快速探讨大范围问题；
5. 得到贴切的数据

缺点：
1. 使得被会见者厌烦；
2. 得不到丰富的细节；

3. 出于上述原因，失去主要思想；
4. 不能建立和面谈者的友好关系。

（2）用户调查

采用用户调查的时机：

```
1．在市场调查时，由于很难深入接触到潜在的用户。所以总是先调查，后访谈。
2．在需求获取时，通常采用的策略是先访谈，后调查。
```

其实原因在于市场调查与需求获取有不同的应用背景。
1. 一般市场调查通常用于验证潜在客户对产品的接受程度。而需求获取的目标是要理解客户需要解决的问题。
2. 需求获取时你往往还没有产品，信息不够充分，所以很难设计出有效的调查问卷。

什么时候开展用户调查工作?

用户调查的目标是为克服用户访谈的片面性，当片面性矛盾比较突出时采用用户调查的方式。
从实际操作看：当出现以下情况时，可以采用用户调查。
（1）存在大样本用户：在操作层面上尤其突出。有些岗位在用户单位中从业人数非常大，进行访谈具有片面性，不可能一一访谈。通常在此情况下，需要采用用户调查的方式。
（2）存在跨地域的用户：用户单位分散在多个区域，所需要解决的问题会不尽相同。这时需要采用用户调查的方式获取这些差异性的问题。

（3）原型法

```
是什么：原型是一个系统，它内化了（capture）一个更迟系统（later system）的本质特征。原型系统通常被构造为不完整的系统，以在将来进行改进、补充或者替代。
为什么要建立：如果在最终的物件（final artifact）产生之前，一个中间物件（mediate artifact）被用来在一定广度和深度范围内表现这个最终物件，那么这个中间物件就被认为是最终物件在该广度和深度上的原型。
```

开发方式分类

```
1．抛弃式原型，探索式和实验式方法产生的原型产品
        花费最小的代价，争取最快的速度
        可能会使用简易的开发工具和不成熟的构造技术
        可能会忽略或简化处理原型目的不相关的功能特征
        要坚决的抛弃
2．演化式原型，演化式原型方法产生的原型产品
        质量要从一开始就能达到最终系统的要求
        要易于进行扩展和频繁改进，因此开发者必须重视演化式原型的设计
        仅应该被用于处理清晰的需求规格说明和技术方案
```

需求分析概述

1．需求分析是软件需求中最核心的工作，需求建模是需求分析的主要手段
2．需求分析是软件定义时期的最后一个阶段，他的基本任务是准确的回答“系统必须做什么”。需求分析的任务还不是确定系统怎样完成他的工作，而仅确定系统必须完成哪些工作。也就是对目标系统提出完整、准确、清晰、具体的要求。

建立分析模型

模型的描述

1．语法：语法严格同时又不太复杂
　　怎样使用模型的元素并且以什么方式组织、连接或关联这些元素；
2．语义：语义丰富
　　特定模型元素所具有的含义；
3．语用：语用复杂
　　模型元素的上下文，以及影响该模型元素意义的约束和假定

需求分析方法

UML

1．UML是一种语言，是一种表示方法，不是方法论。
2．是一种建模语言，不是编程语言。
3．是统一建模语言，是一种标准化的、统一的建模语言，OMG认可的工业标准。

面向对象分析

OO优点：

1．增加可复用性
2．增加可扩展性
3．改进质量
4．财务利益
5．增加项目成功机会
6．减少维护负担
7．减少应用积压
8．可管理的复杂性

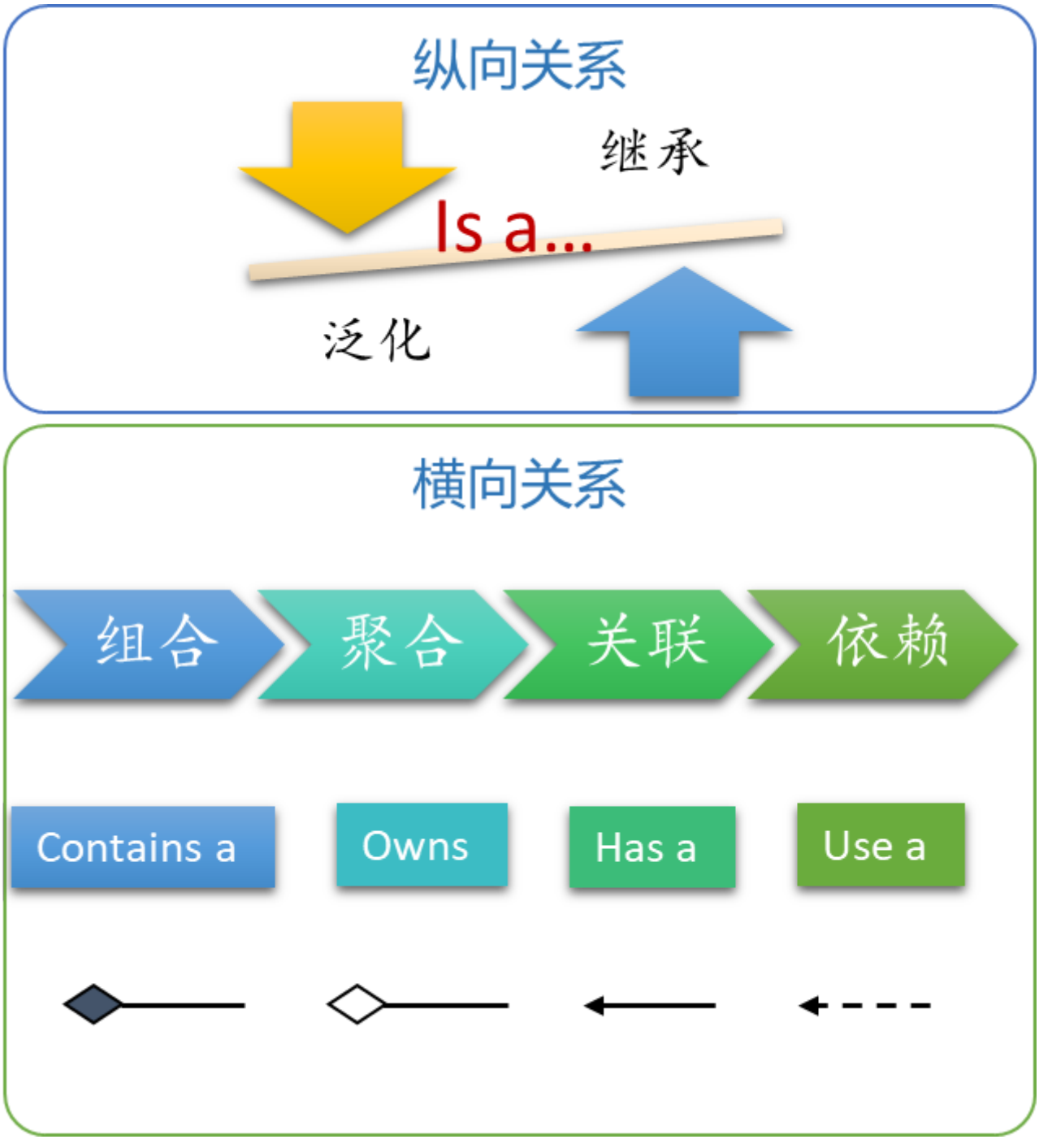
OO基础：

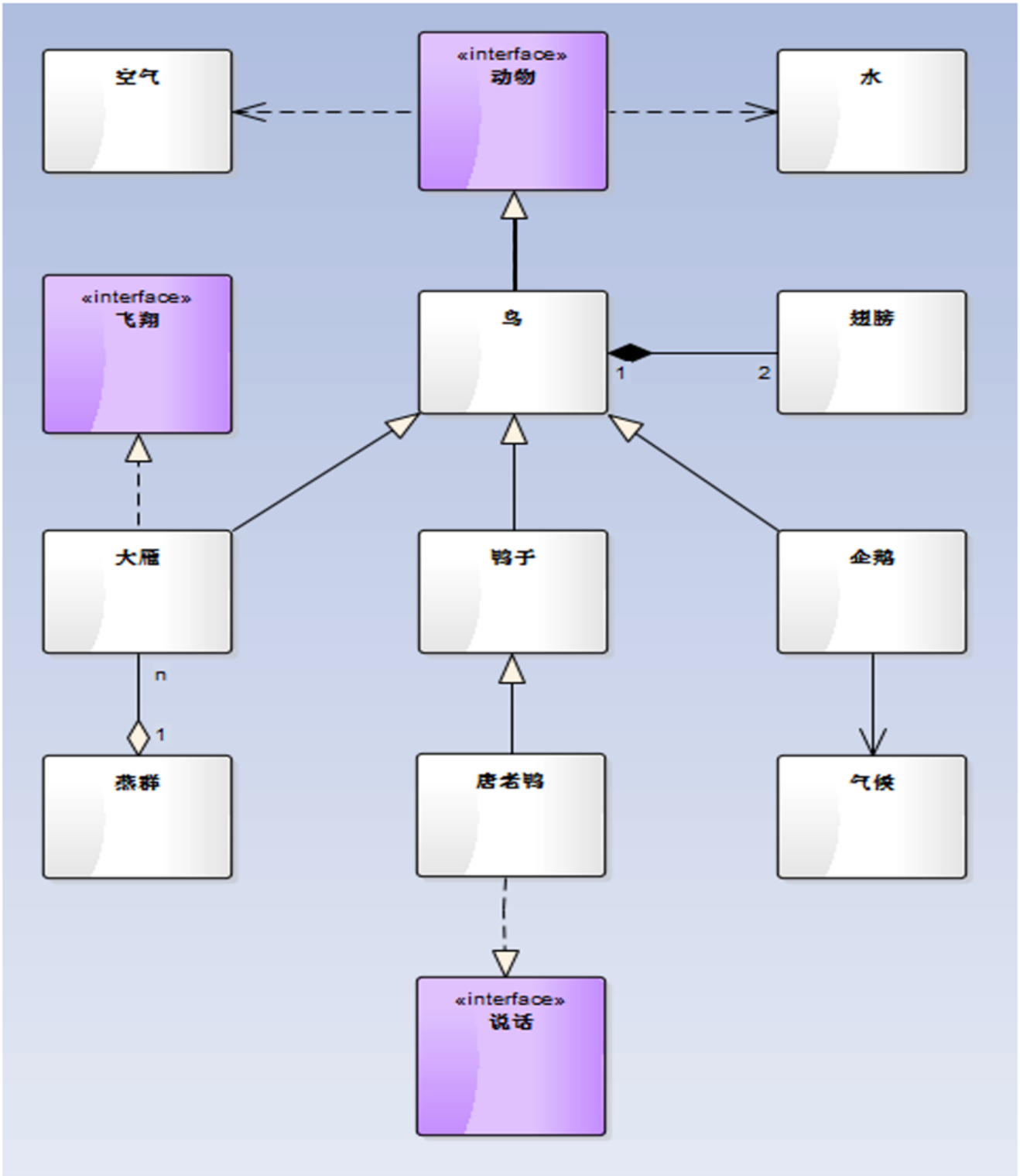
1．多态
　　多态性是面向对象程序设计的重要特性之一。指当不同对象收到相同消息时，产生不同的动作。

C++的多态性具体体现在运行和编译两个方面，在程序运行时的多态性通过继承和虚函数来体现，而在程序编译时多态性体现在函数和运算符的重载上。

对象模型构建-类图

建模概念：类间关系



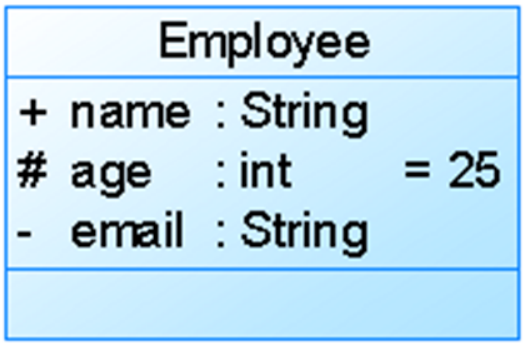


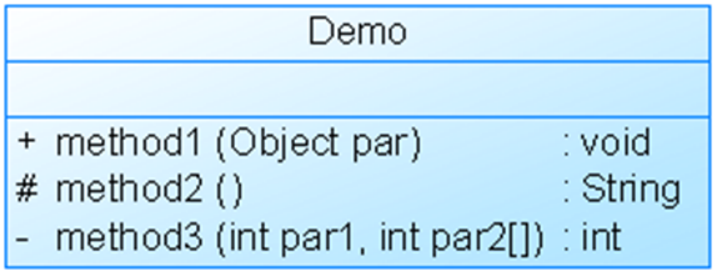
类与类图

•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。

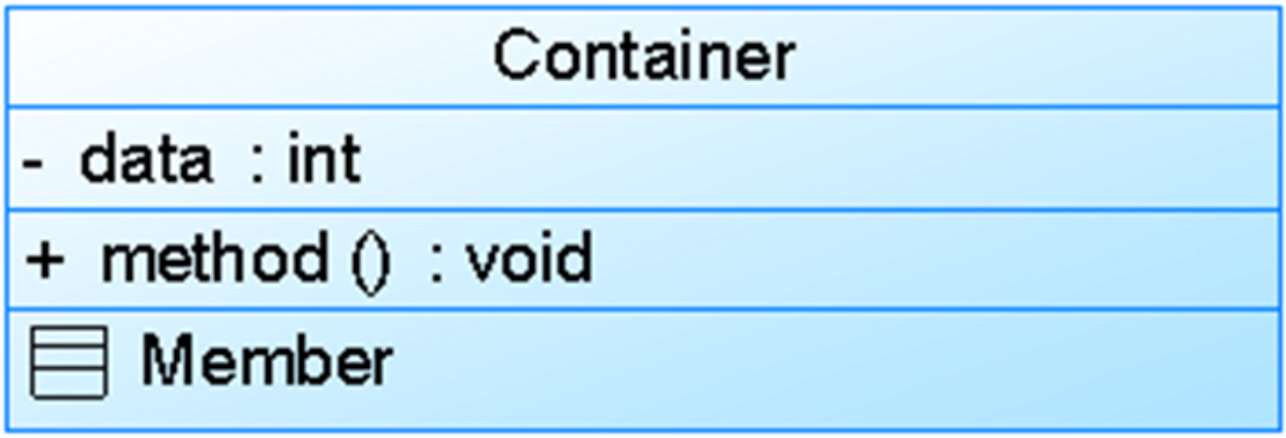
•在系统中，每个类具有一定的职责，职责指的是类所担任的任务，即类要完成什么样的功能，要承担什么样的义务。一个类可以有多种职责，设计得好的类一般只有一种职责，在定义类的时候，将类的职责分解成为类的属性和操作（即方法）。

类的属性即类的数据职责，类的操作即类的行为职责。



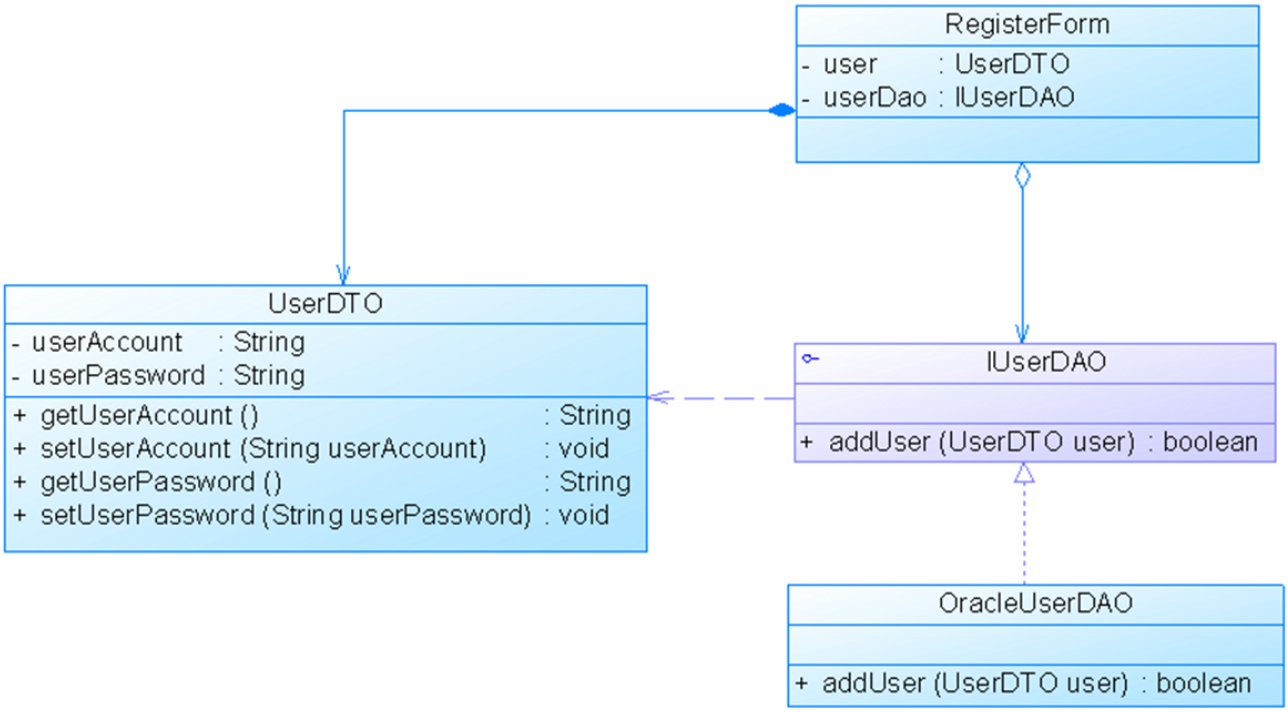


•在UML类图中，类一般由三部分组成：
•类名：每个类都必须有一个名字，类名是一个字符串。
•属性(Attributes)：属性是指类的性质，即类的成员变量。类可以有任意多个属性，也可以没有属性。可见性 名称:类型 [= 默认值]
•操作(Operations)：操作是类的任意一个实例对象都可以使用的行为，操作是类的成员方法。



画类图

实例
•某基于Java语言的C/S软件需要提供注册功能，该功能简要描述如下：
•用户通过注册界面(RegisterForm)输入个人信息，用户点击“注册”按钮后将输入的信息通过一个封装用户输入数据的对象(UserDTO)传递给操作数据库的数据访问类(DAO)，为了提高系统的扩展性，针对不同的数据库可能需要提供不同的数据访问类，因此提供了数据访问类接口，如IUserDAO，每一个具体数据访问类都是某一个数据访问类接口的实现类，如OracleUserDAO就是一个专门用于访问Oracle数据库的数据访问类。
•根据以上描述绘制类图。为了简化类图，个人信息仅包括账号(userAccount)和密码(userPassword)，且界面类无须涉及界面细节元素。



需求验证概述

需求验证严格说就是检验软件需求规格说明，是需求开发的最后一项活动。

目标

需求验证的目标是尽可能发现存在的错误，以减少因为需求错误而带来的工作量的问题。

主要手段：

需求评审

项目范围管理

范围定义

PMBOK定义，范围是指产生项目产品所包含的所有工作及产生这些产品的过程。

```
1. 范围管理原则：确保做且只做所需的全部工作。
2. 不遵守原则带来的问题：范围蔓延和范围镀金。
```

范围蔓延

客户在项目过程中提出新要求，导致我们做了很多范围之外的事情，最后我们可能没法按时完成项目或者需要多投入更多资源才能把项目做完。

范围镀金

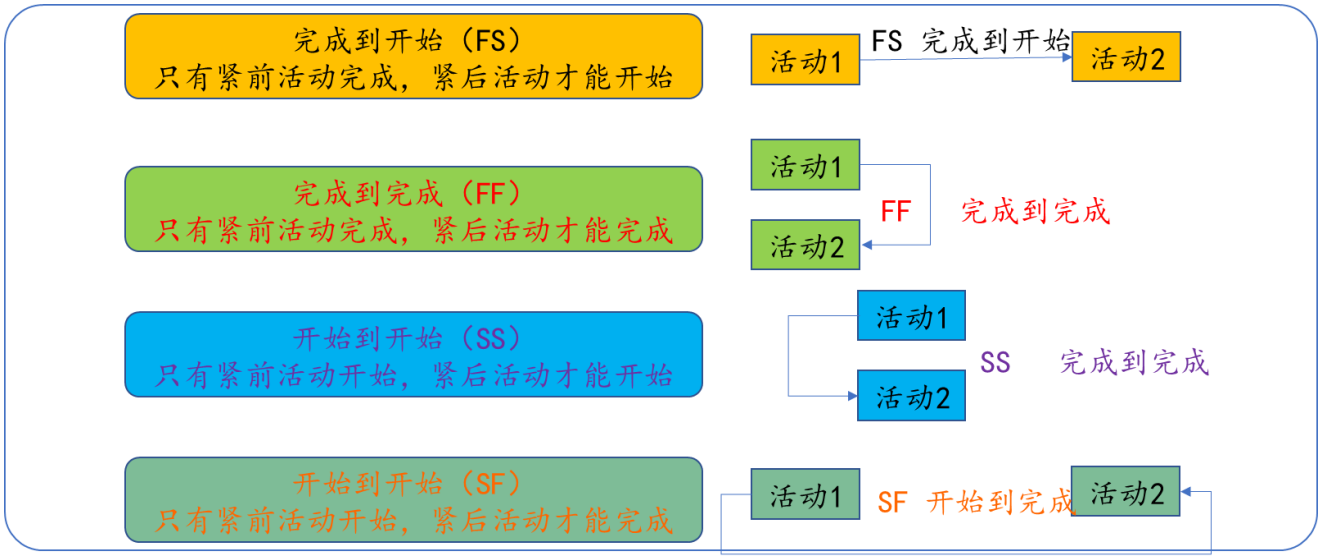
客户没有给项目提出什么新的要求，我们自己却做了很多项目范围之外的事情，客户还不一定满意，这其实是我们更不愿意看到的现象。

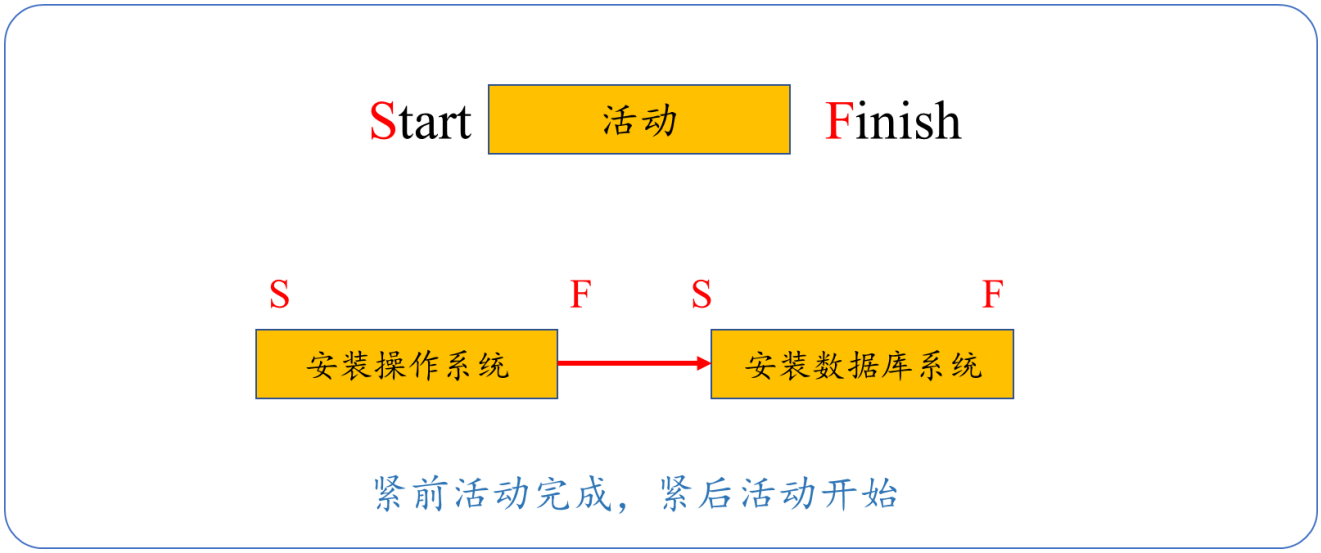
不管是范围蔓延还是范围镀金，本质上都是因为我们做了很多范围之外的工作或者干了我们不该干的事情，作为项目经理，应该杜绝这两种情况出现。
客户提出来新的要求，最好让他走一个正式的审批和变更流程，不能让客户随意改变我们项目的范围。

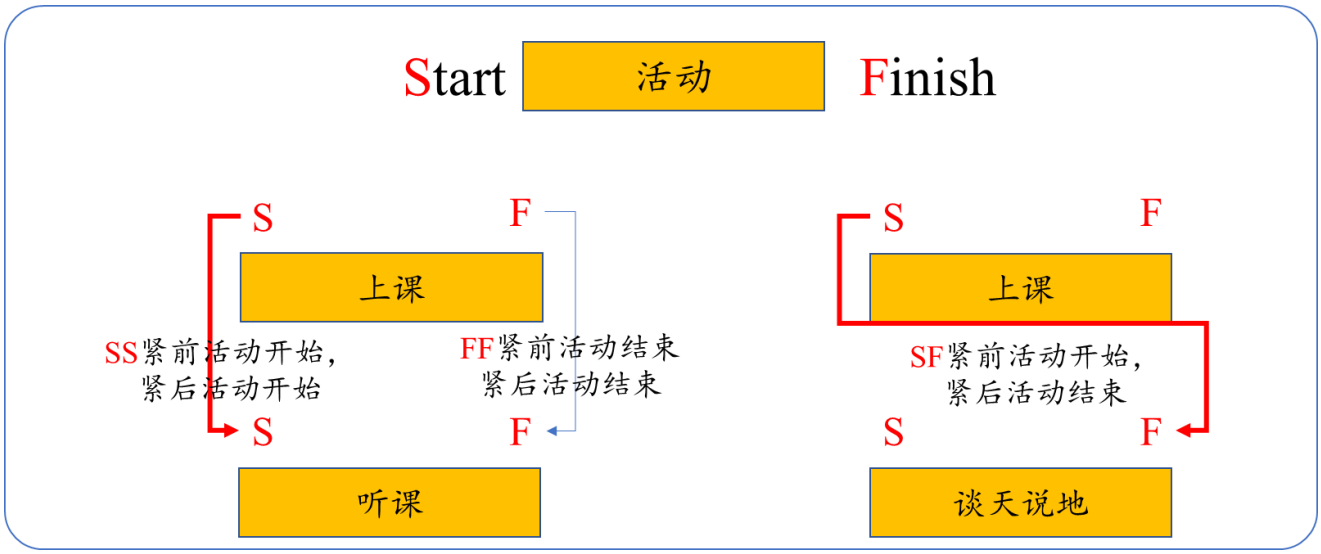
项目进度管理

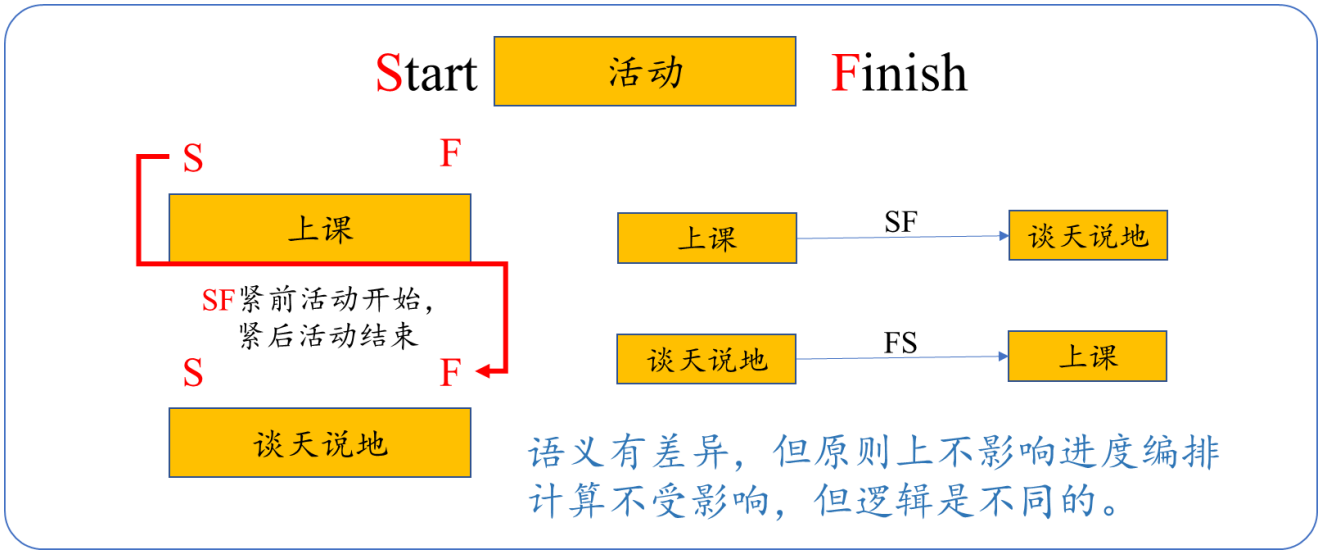
项目任务间的关系

1. 关联关系









2．确定和整合依赖关系

（1）外部依赖关系

项目活动与非项目活动之间的依赖关系（项目团队不可控）

（2）内部依赖关系

是项目活动之间的紧前关系（项目团队可控）

（3）强制性依赖关系（硬逻辑、硬关系）

法律或合同要求的或工作内在性质决定的依赖关系（项目团队不能违反）。

（4）选择性依赖关系（软逻辑、优先逻辑）

基于最佳实践建立的、或基于项目的某些特殊性质而采用的依赖关系（项目团队可自由选择）。

排列活动顺序

网络图

1．网络图是活动顺序的一个输出

2．展示项目中各个活动以及活动之间的逻辑关系

PDM

优先图法，节点法（单代号）网络图

ADM

箭线法 (双代号)网络图

甘特图

> 定义：甘特图是以图示的方式通过活动列表和时间刻度形象地表示出任何特定项目的活动顺序与持续时间。它能够直观地表明任务计划在什么时候进行，以及实际发展与计划要求的对比。它是活动计划拟定阶段的必备工具。
> 结构：甘特图分为三个部分，横坐标表示时间，纵坐标表示工作顺序或活动内容，线条表示在整个工作期间内计划和实际的完成情况。

估算活动持续时间

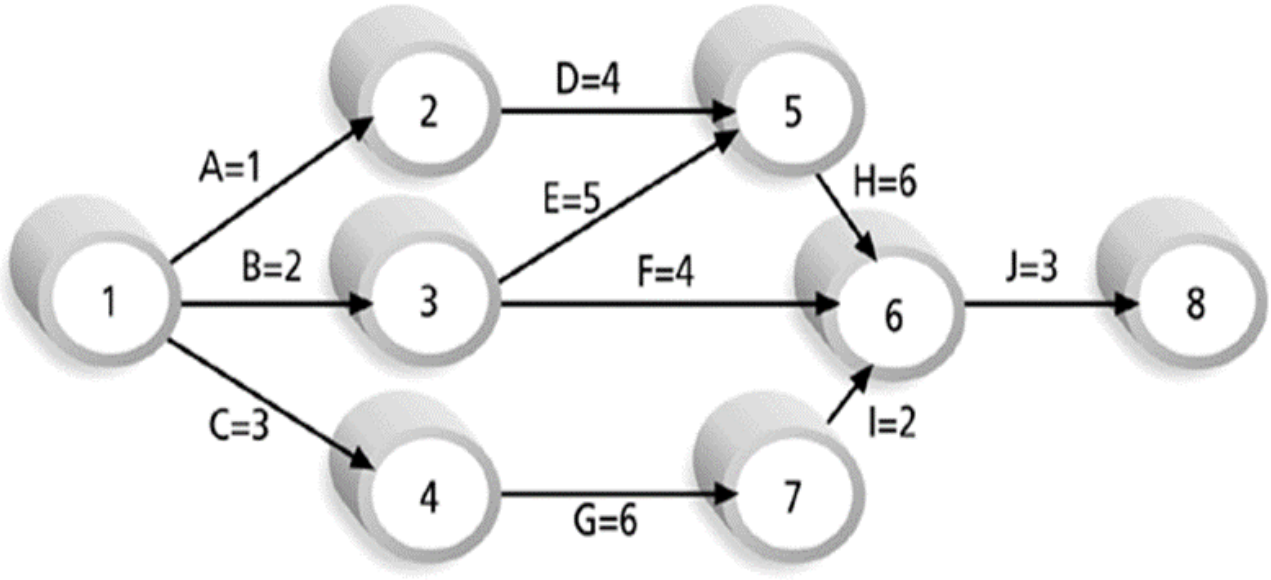
•历时估算的基本方法——定额估算法
T=Q/(R*S)
T:活动历时
Q:任务工作量
R:人力数量
S:工作效率（贡献率）
•历时估算的基本方法——工程评估评审技术
•加权算法

- 它是基于对某项任务的乐观，悲观以及最可能的概率时间估计
- 采用加权平均得到期望值E= (O+4m+P)/6,
  - O是最小估算值:乐观(Optimistic),
  - P是最大估算值:悲观(Pessimistic),
  - M是最大可能估算(Most Likely)。

制订进度计划

关键路径法

> 关键路径
>
> 时间浮动为0（Float=0）的路径
> 网络图中最长的路径
> 关键路径是决定项目完成的最短时间。
> 关键路径上的任何活动延迟，都会导致整个项目完成时间的延迟
> 关键路径可能不止一条



Note: Assume all durations are in days.

Path 1:	A-D-H-J	Length = 1+4+6+3 = 14 days
Path 2:	**B-E-H-J**	**Length = 2+5+6+3 = 16 days**
Path 3:	B-F-J	Length = 2+4+3 = 9 days
Path 4:	C-G-I-J	Length = 3+6+2+3 = 14 days

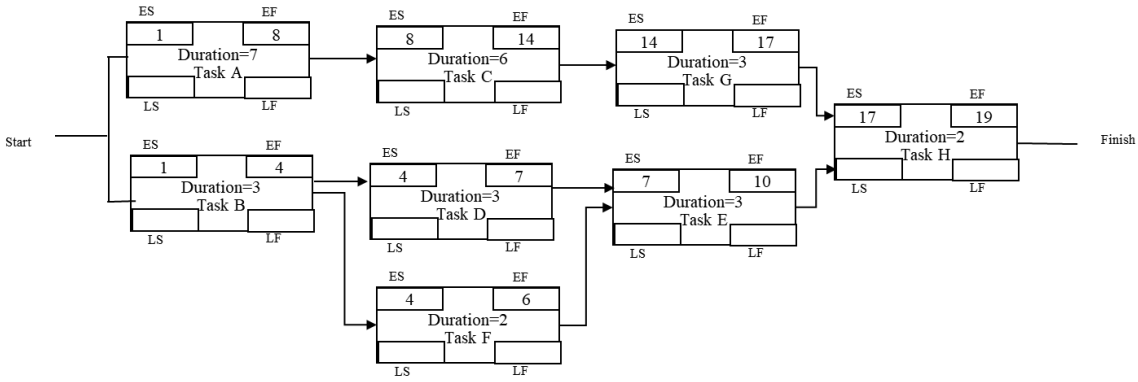
Since the critical path is the longest path through the network diagram, Path 2, B-E-H-J, is the critical path for Project X.

项目网络图——正推法

- 按照时间顺序计算最早开始时间和最早完成时间的方法,称为正推法
  1）确定项目的开始时间。
  2）从左到右，从上到下、
  3）计算每个任务的最早开始时间ES和最早完成时间EF

网络图中最后一个任务最晚完成时间是项目的结束时间；
LF-Duration=LS
LS-Lag=LF（p）
当一个任务有多个后置任务时，选择其后置任务中最小LS减Lag作为其LF

实例



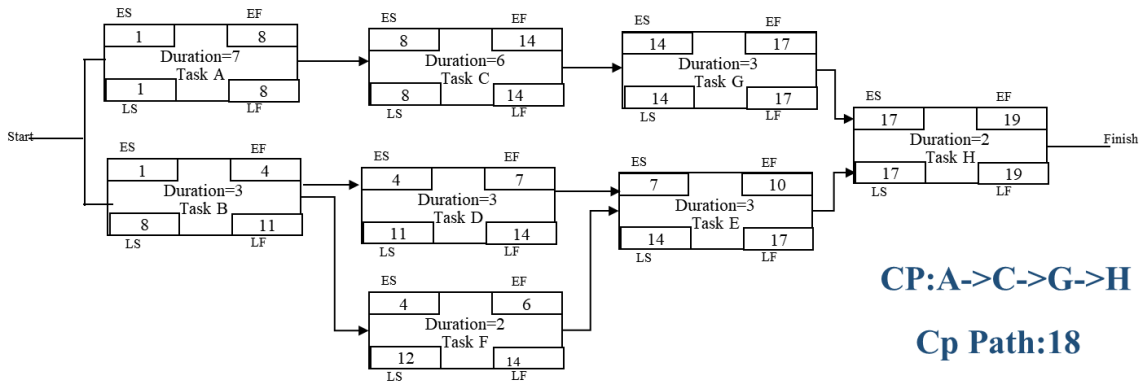
当一个任务有多个前置任务时，选择前置任务中最大的EF加上Lag作为其ES。

项目网络图——逆推法

1)确定项目的结束时间
2)从右到左，从上到下
3计算每个任务的最晚开始时间LS和最晚完成时间LF

1)确定项目的结束时间
2)从右到左，从上到下
3计算每个任务的最晚开始时间LS和最晚完成时间LF

实例



当一个任务有多个后置任务时，选择其后置任务中最小LS减Lag作为其LF。

时间压缩法

应急法——赶工

进度压缩单位成本=（压缩成本-正常成本）/(正常进度-压缩进度)

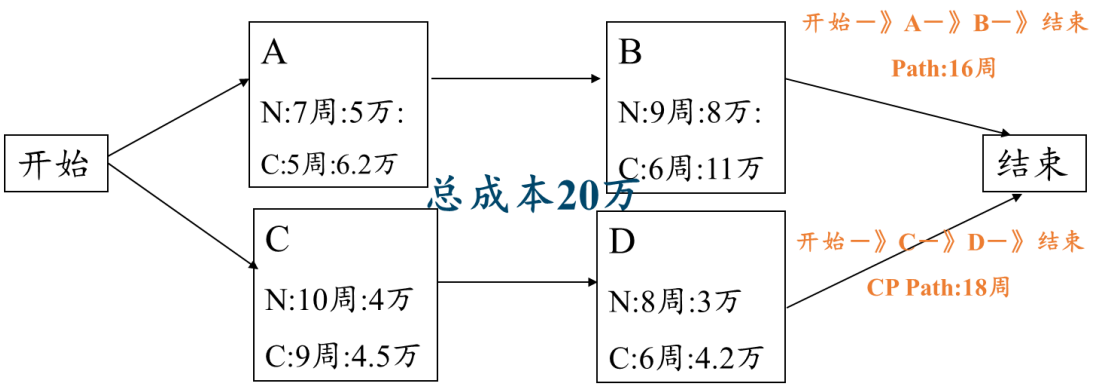
实例
任务A:正常进度7周,成本5万；
压缩到5周的成本是6.2万；

进度压缩单位成本=(6.2-5)/(7-5)=6000元/周
如果压缩到6周的成本是：5.6万

时间压缩法
应急法-赶工
（Crash）

实例

下图给出了各个任务可以压缩的最大限度和压缩成本，请问如果将工期压缩到17周，16周，15周时应该压缩的活动和最后的成本？



先计算单位压缩成本，每周多少钱，再确定需要压缩多少才能成本最低，先压缩关键路径上单周成本低的。

时间压缩法
应急法-赶工
（Crash）

实例

下图给出了各个任务可以压缩的最大限度和压缩成本，请问如果将工期压缩到17周，16周，15周时应该压缩的活动和最后的成本？

结果

压缩任务及成本 / 完成周期（单位:周）	压缩的任务	成本计算 (单位:万)	项目成本 (单位:万)
18		5+8+4+3	20
17	C	20+0.5	20.5
16	D	20.5+0.6	21.1
15	A,D	21.1+0.6+ 0.6	22.3

成本估算

总成本 = 直接成本 + 间接成本
直接成本：与具体项目相关的成本：参与项目的人员成本
间接成本：可以分摊到各个具体项目中的成本

制订预算

分配项目成本预算三种情况：
1. 给任务分配资源成本：资源的基本费率
2. 给任务分配固定资源成本：固定数量资金
3. 给任务分配固定成本：成本与工期无关：外包任务等

软件构造

软件构造加强软件的质量

如果不关心软件的质量，那么软件构造与一门语言课程无异

为了实现高质量的软件产品，则不仅需要有编码工作，还必须遵守软件构造的各种原则，具有良好的软件设计，遵守一定的编码原则以及规范的编码过程，具有软件测试的过程等，这些都是属于软件构造的内容

目的

在了解软件构造的原理、规则和标准之后，利用学到的软件构造知识，高质量高效率地构造软件项目。
从软件的变量、语句、注释、布局、子程序、类以及结构等不同层次保证软件的优秀性。
ADT 防御性编程 （断言 异常 隔离 ）

概念

软件构造（Software Construction）指的是通过编码、验证、单元测试、集成测试和调试的组合，详细地创建可工作的，有意义的软件。

软件构造知识域与其它所有的知识域都有联系，特别是软件设计、软件测试，另外还涉及到配置管理，软件工具等内容。

软件开发过程活动

1.定义问题（problem definition）
2.需求分析（requirements development）
3.软件构架（software architecture）
4.规划构建（construction planning）
5.详细设计（detailed design）
6.编码与调试（coding and debugging）
7.单元测试（unit testing）
8.集成（Integration）
9.集成测试（Integration testing）
10.系统测试（system testing）
11.系统维护（corrective maintenance）

构造活动的具体任务

1.验证有关的基础工作已完成，因此构造活动可以顺利进行下去
2.确定如何测试所写的代码
3.设计并编写类和子程序
4.创建并命名变量和具名常量
5.选择控制结构，组织语句块
6.对代码进行单元和集成测试，排除错误
7.评审开发团队其他成员的底层测试和代码
8.润饰代码，仔细进行代码的格式化和注释
9.将单独开发的多个软件组件集成为一体
10.调整代码，让它更快、更省资源

构造的重要性

1.构造活动是软件开发的主要组成部分，根据项目规模的不同，构造活动占据30%-80%的开发时间
2.构造活动是软件开发的核心活动
3.把主要精力集中于构造活动，可以大大提高程序员的生产率（相差10倍）
4.构造活动的产物源代码往往是对软件的唯一精确的描述
5.构造活动是唯一一项确保完成的工作

软件构造内容

1.前期准备的重要性

什么时期强调质量最为重要?

1.如果在项目的末期强调质量，则会强调系统测试
2.如果在项目的中期强调质量，则会强调构造实践
3.如果在项目的前期强调质量，则会计划、要求并且设计一个高质量的产品

前期准备的目的

1. 使用高质量的实践方法是那些能创造高质量软件程序员的共性
2. 这些高质量实践方法在项目的初期、中期、末期都强调质量
3. 前期准备工作的中心目标就是降低项目的风险，以使项目平稳进行

前期出错会花费更高的代价

•需求中引入的缺陷到发布时发现会引入上百倍的修复成本

表1 修复缺陷的平均成本与引入时间之间的关系

引入缺陷时间	需求	构架	构建	测试	发布后
需求	1	3	5-10	10	10-100
构架	-	1	10	15	25-100
构建	-	-	1	10	10-25

前期准备不足的原因

尽管前期准备非常重要，但很多时候项目还是前期准备不足，为什么?
1.做前期准备的开发人员不具备相应的专业技能（能力）
2.开发人员不能抵抗“尽快开始编码”的欲望（态度）
3.管理者们对那些“花时间进行构造的前期准备”的冷漠态度（态度）
找到原因的目的是克服准备不足的问题

2.软件类型的影响

Caper Jones指出不同类型的软件项目，需要在准备工作和构造活动之间做出平衡。
每一个项目都是独特的，但是项目可以归入若干种开发风格，最常见的三种软件类型是：商业系统、使命攸关系统和性命攸关系统。

表2 不同类型软件的典型应用

商业系统	使命攸关系统	性命攸关系统
Internet站点 Intranet站点 库存管理 游戏 管理信息系统 工资系统	嵌入式软件 游戏 Internet站点 盒装软件 软件工具 Web Services	航空软件 嵌入式软件 医疗设备 操作系统 盒装软件

表3 不同类型软件的开发模型

商业系统	使命攸关系统	性命攸关系统
敏捷开发（极限编程、Scrum、Time-box开发等） 渐进原型（prototyping）	分阶段交付 渐进交付 螺旋型开发	分阶段交付 渐进交付 螺旋型开发

表4 不同类型软件的计划与管理

商业系统	使命攸关系统	性命攸关系统
增量式项目计划 随需测试与QA计划 非正式的变更控制	基本的预先计划 基本的测试计划 随需QA计划 正式的变更控制	充分的预先计划 充分的测试计划 充分的QA计划 严格的变更控制

表5 不同类型软件的需求形式

商业系统	使命攸关系统	性命攸关系统
非正式的需求规格	半形式化的需求规格 随需的需求评审	形式化的需求规格 形式化的需求检查

表6 不同类型软件的设计方法

商业系统	使命攸关系统	性命攸关系统
设计与编码是结合的	构架设计 非形式化的详细设计 随需的设计评审	构架设计 形式化的构架检查 形式化的详细设计 形式化的详细设计检查

表7 不同类型软件的构造方法

商业系统	使命攸关系统	性命攸关系统
结对编程或独立编码 非正式的Check-In手续 或没有Check-In手续	结对编程或独立编码 非正式的Check-In手续 随需代码评审	结对编程或独立编码 正式的Check-In手续 正式的代码评审

结对编程（Pair programming）是一种敏捷软件开发的方法，两个程序员在一个计算机上共同工作。一个人输入代码，而另一个人审查他输入的每一行代码。输入代码的人称作驾驶员，审查代码的人称作观察员（或导航员）。两个程序员经常互换角色。

表9 不同类型软件的部署方式

商业系统	使命攸关系统	性命攸关系统
非正式的部署过程	正式的部署过程	正式的部署过程

3.顺序开发还是迭代开发

(1)通常而言，开发商业系统往往受益于高度迭代的开发方法，这种方法的计划、需求、构架、构建、测试与质量保证的活动往往交织在一起

(2)性命攸关的系统往往要求采用更加序列式的方法，需求稳定是确保超级可靠性的必备条件之一

¨无论采用什么样的开发方法，前期准备都是重要的

¨对顺序方法可以先明确80%的需求，对于迭代方法可以先明确20%的需求

对于其他的项目，各种活动在项目开发期间会重叠起来。成功“构建”的关键之一是理解前期准备工作的完成程度，并据此调整项目的开发方法。

选择顺序开发的原因如下：

1.需求相当稳定

2.设计直截了当，而且理解透彻

3.开发团队对于这一应用领域非常熟悉

4.项目的风险很小

5.长期可预测性很重要
6.后期改变需求，设计和代码的代价很可能较昂贵

选择迭代开发的原因如下：

1.需求并没有被理解透彻，或者出于其他理由认为它是不稳定的
2.设计很复杂，具有挑战性，或两者兼具
3.开发团队对这一应用领域不熟悉
4.项目包含很多风险
5.长期可预测性不重要
6.后期改变需求、设计和编码的代价很可能较低

4.问题定义的先决条件

（1）在开始构建之前，首要要满足的一项先决条件是，对这个系统要解决的问题作出清楚地定义
（2）问题定义只定义问题是什么，而不涉及到任何可能的解决方案

问题的定义是开发的方向

（1）问题相当于方向，方向错误开发一定不能成功
（2）问题应该用客户的语言来书写，即从客户的角度去描述问题

5.需求的先决条件

为什么要有正式的需求

1.明确的需求有助于确保是用户驾驭系统的功能
2.明确的需求有助于避免争论，因为需求明确了系统的范围
3.重视需求有助于减少开始编程开发之后的系统变更情况

稳定的需求是不可能的

（1）稳定的需求是软件开发的圣杯
（2）IBM和其它公司的研究发现，平均水平的项目在开发过程中，需求会有25%的变化
（3）在典型的项目中，需求变更导致的返工占返工总量的75%到85%

在构建期间如何处理需求变更

1.使用需求核对表来评估需求的质量
2.确保每一个人都知道需求变更的代价
3.建立一套变更控制程序
4.使用能适应变更的开发方法

5.放弃这个项目
6.注意项目的商业案例

需求检查核对表

对功能需求的检查

1.是否详细定义了系统的全部输入
2.是否详细定义了系统的全部输出
3.是否详细定义了所有输出格式
4.是否详细定义了所有硬件和软件外部接口
5.是否详细定义了全部外部通信协议，包括握手协议、纠错协议，通信协议等
6.是否列出了用于想要的全部事情
7.是否定义了每个任务所用的数据，以及每个任务得到的数据

针对非功能需求的检查

1.是否为全部必要的操作，从用户的视角，详细描述了期望响应时间吗？
2.是否详细描述了其他与计时有关的考虑，例如处理时间，数据传输率，系统吞吐量等
3.是否详细定义了安全级别
4.是否详细定义了可靠性，包括软件失灵后果，发生故障时需要保护的重要信息，错误检查和恢复策略等
5.是否详细定义了机器内存和剩余磁盘最小值
6.是否详细定义了系统的可维护性
7.是否包含对“成功”和“失败”的定义

需求质量的检查

1.需求是用用户的语言书写的吗
2.每条需求都不与其它需求冲突吗
3.是否详细定义了相互竞争的特性之间的权衡
4.是否避免在需求中规定设计
5.需求是否在详细程度上保持相当一致的水平
6.需求是否足够清晰，都容易理解
7.每个条款都与待解决的问题相关吗
8.是否每条需求都可以测试
9.是否详细描述了所有可能的对需求的改动

需求的完备性检查

1.对于在开始开发之前无法获得的信息，是否详细描述了信息不完全的区域
2.需求的完备度是否能达到这种程度；如果产品满足所有需求，那么它就是可接受的
3.对全部需求都感到很舒服吗？是否已经去掉了那些不可能实现的需求

6.构架的先决条件

（1）软件构架是软件设计的高层部分，是用户支撑更细节的设计框架
（2）构架的质量决定了系统的“概念完整性”，而概念完整性决定了系统的最终质量
（3）好的构架使得构造活动能够变得更容易，因此需要在构造前评估构架

构架的总体质量

(1)优秀的构架的特点在于：讨论了系统中的类、讨论了每个类背后的信息隐藏，讨论了采纳或排斥所有可能的设计替代方案及其理由;
(2)构架应该描述所有主要决策的动机
(3)构架在很大程度上是与机器和编程语言无关的
(4)构架应该明确地指出了风险的区域，并使风险最小化

7.前期准备上花费的时间比例

软件架构概述

定义

软件系统的一个或多个结构，包括软件组件、这些组件的外部可见特性，以及这些组件之间的相互关系。

包含：

组件
组件的外部可见特性
组件之间的相互关系

理解：

（1）架构是一个或多个系统的抽象
（2）是由抽象的组件来表示的
（3）组件具有外部的可见特性
（4）组件相互之间是有联系的

系统抽象屏蔽了组件内部特有的细节

系统抽象：

组件与联系
用视图的方式表示

常见的软件架构

> 由结构和功能各异、相互作用组件的集合，按照层次构成。

软件架构包含了

> （1）系统的基础构成单元（组件）
> （2）它们之间的作用关系（连接/连接件）
> （3）在构成系统时，它们的集成方法以及对集成约束的描述。

架构={ 组件、连接件、约束 }

> 组　　件：一组程序代码、进程、接口
> 连接件：调用、管道、消息
> 约　　束：组件之间的连接关系

架构是关于软件系统组织的配置定义
定义了：
组成系统的结构（结构元素/组件）
结构或组件间特定的构成和协作关系
系统的集成方法和约束

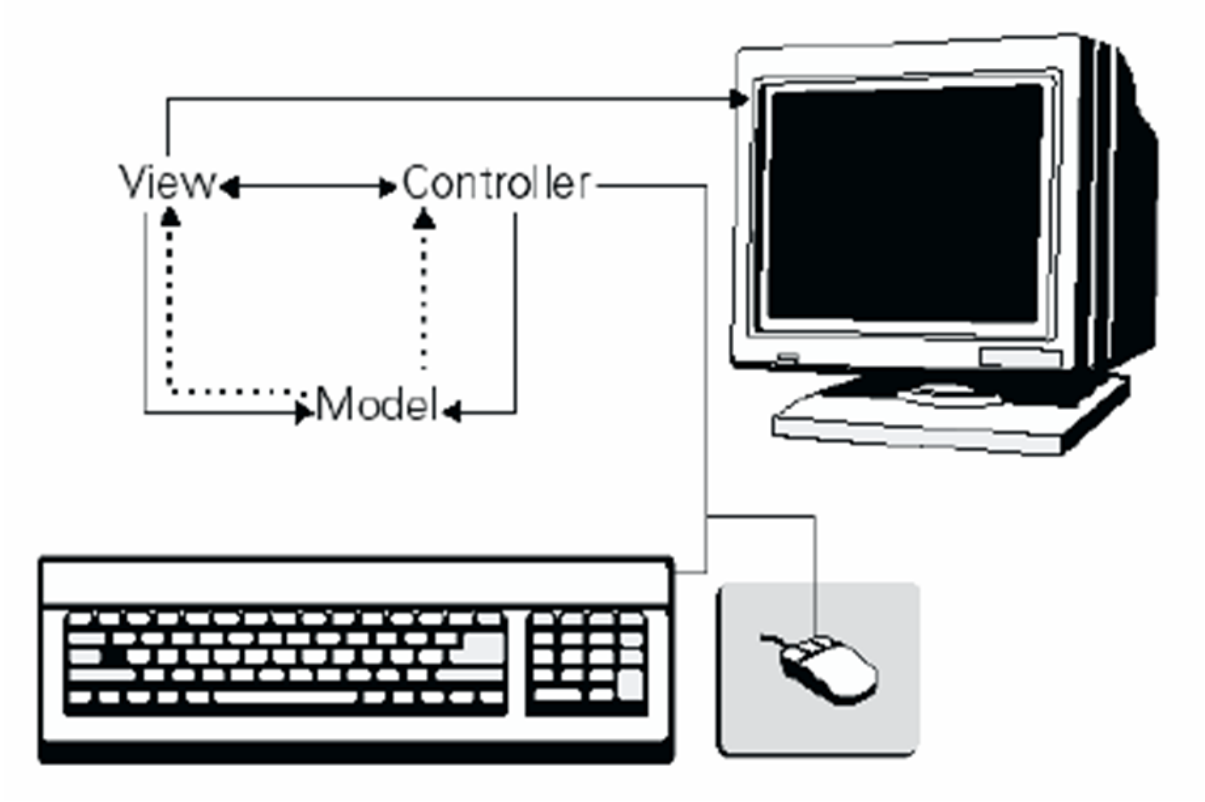
所以，软件架构不但决定了系统的物理构成，也支配了开发的组织行为：需求分配、范围和任务定义、进度计划、测试方案、集成方法、配置项与基线管理。这是现代软件工程更关注的地方。

软件架构作用

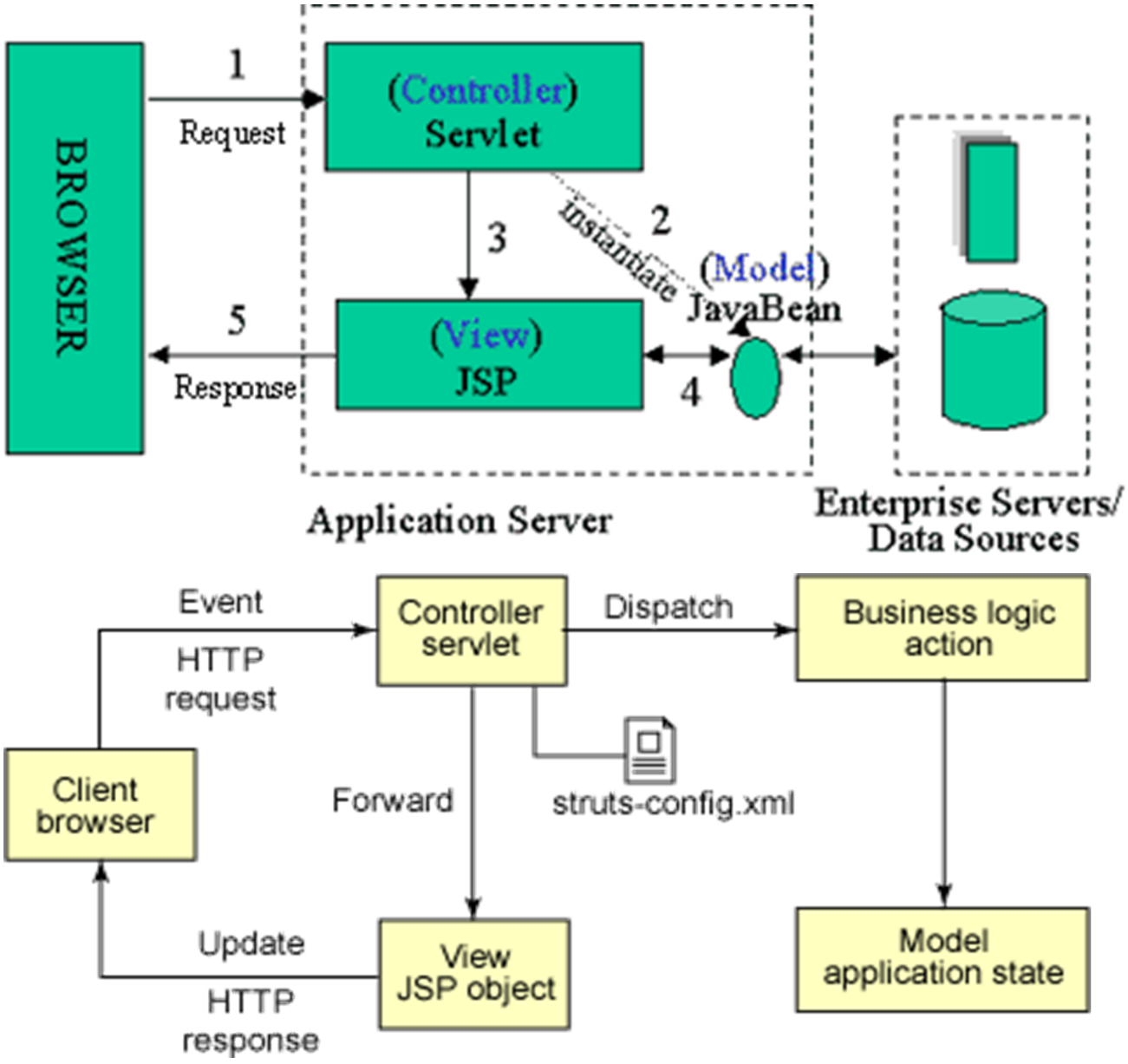
（1）软件架构定义了软件计算的组件、局部和总体的构成关系、以及这些组件之间的相互作用
•计算组件：
•客户、服务器、数据库、中间件、程序包、过程、子程序、进程等——切碎、再切碎（粒度）
•相互作用关系：
•过程调用、共享变量访问、信号灯、进程通信、消息传递、访问/网络协议等
•《操作系统原理》课程其实最主要的意义和价值就是介绍一个实际的软件系统构架的案例
（2）除了描述系统的构成和结构关系外，软件构架还表达了系统关键需求与系统构成之间的对应关系，这为系统的设计，提供了分析和评价的依据
因为软件构架比需求更进一步要面对或满足系统非功能性的内容，如：容量、数据吞吐量、一致性、兼容性、安全性、可靠性

软件架构的表示方法

1. 最简单的架构视图：物理视图 （计算机系统构架）



2. MVC构架的并发视图



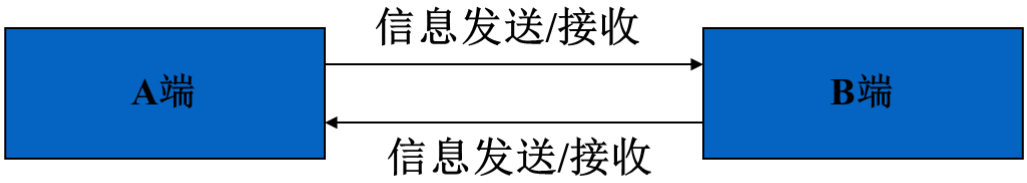
一个由Struts实现的MVC构架

架构与架构师的作用

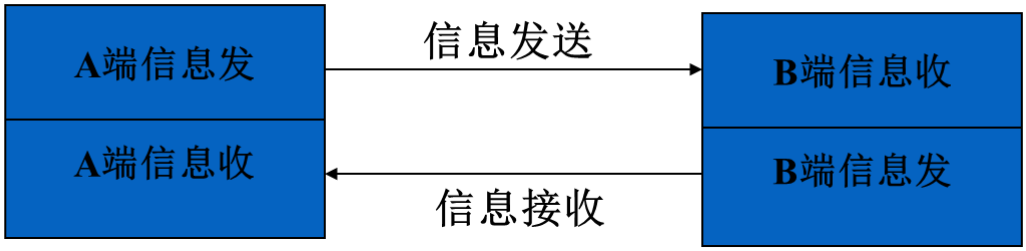
架构是需求将如何被实现的描述

文件传输软件的架构描述与分析

①



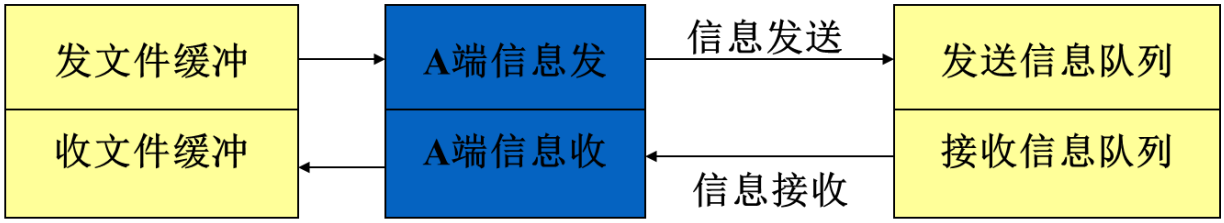
②



A/B端信息收或发各是2个不同的进程

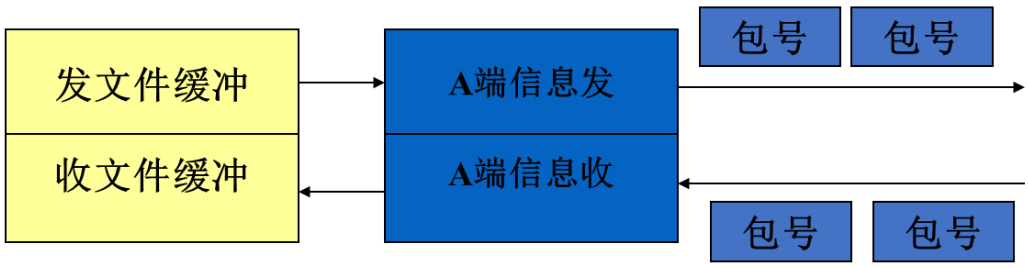
A发B收/B发A收

③



A/B端信息收或发与文件缓冲之间需要进行打包解包、队列缓冲控制等

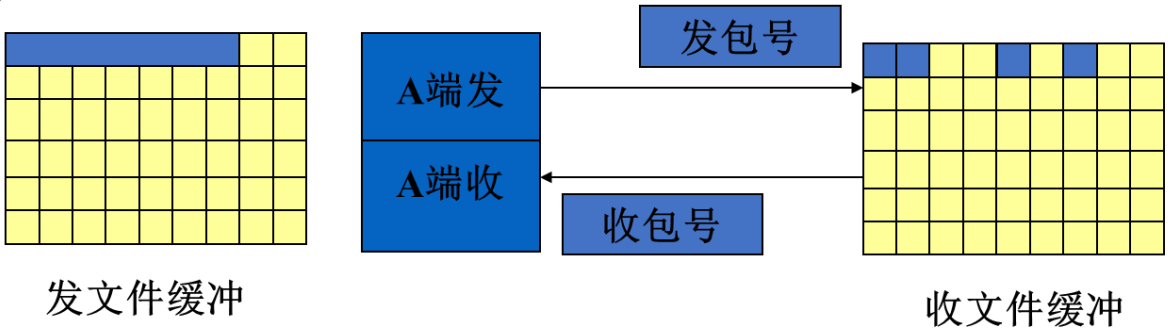
④



文件缓冲按包重发进行控制

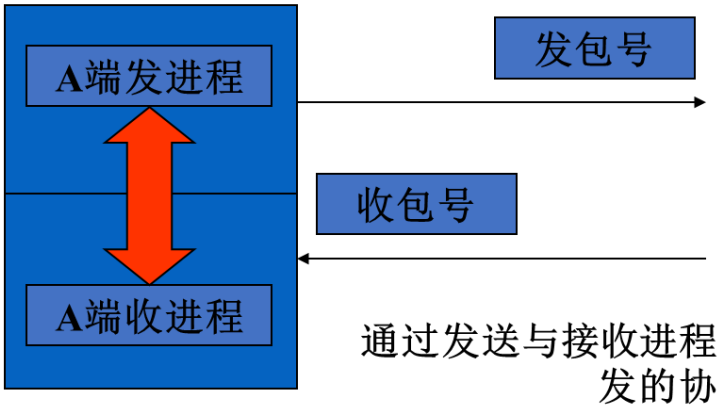
信息缓冲按栈先进先出、队列满暂停方式控制

⑤



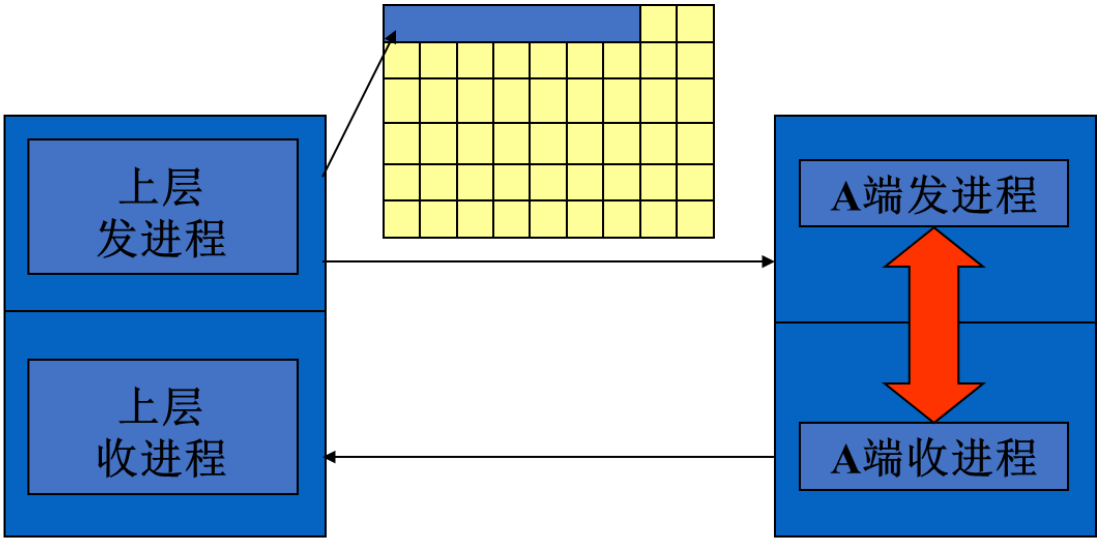
用包号控制发送与接收之间的窗口大小

⑥



通过发送与接收进程之间的通讯，实现收发的协同工作

⑦



上层发进程：发送命令/文件指针

上层收进程：文件发送成功/失败

上层与下层的关系：

同步方式：超时中断进程

异步方式：等待信号

软件构架构成的二个方面

部件和规则（静态、动态）

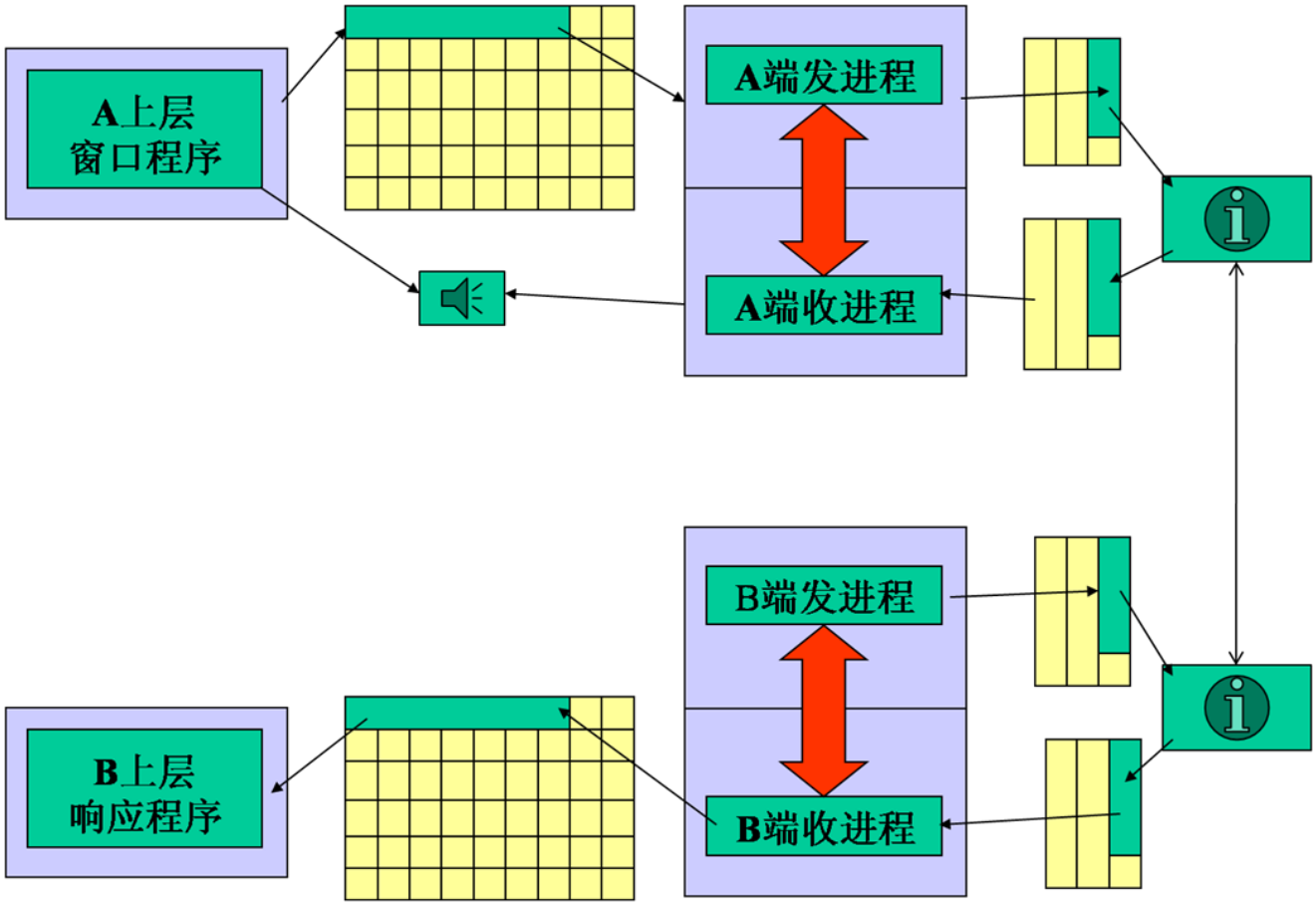
静态

> （1）构成系统的原始的或集成的部件：语句、程序模块、数据结构
> （2）由部件集成为系统的静态集成规则：模块接口、连接件

动态

> 为系统提供语义的动态行为规则：模式控制方式、连接方式、并行/同步模式等

文件传输软件的新需求及其改进方案
新需求：在现有的线路和传输系统基础上，实现“实时性”更好的“即时”短通信。



文件发送接收程序的架构描述

改进方案：
1、在上层应用程序和下层发送接收进程之间，协商建立所谓“优先级”制度——组件、连接都没有变化，连接方式上，增加一条“协议”内容，当出现“优先级”高的文件时，发送和接收进程优先处理。
2、在文件缓冲、发送和接收队列等需要排队的地方，增加一个新的缓冲和队列，它们的优先级比其他已有缓冲和队列的优先级要高
3、在短消息发送和接收过程中，是否可以不采用已有的“打包”、“解包”模式。由于这类信息一般都很短，因此，无需打包、解包。
4、不采用已有的纠错方式，不是不会发送错误，而是不用报告出错包号，再重传出错包的方式，报告和纠正错误，而是对此类信息，采用直接应答方式，既所谓“命令”模式。

这些新扩展，从架构的角度看，就是通过增加新组件（缓冲和队列）、建立新连接（缓冲和队列），更主要的是通过建立新的连接关系，来实现了“即时性”的改善。

架构描述表达了系统要实现的需求

软件系统的“需求”

> 架构描述的第一个目的，就是在架构设计层次上，而不是需求层次上，表达系统的需求。

良好的架构设计，最直接的目的是：

> 使软件系统能够达到为用户提供最佳的功能和服务状态；
> 使软件与系统的结合达到最佳运行性能；
> 合理和最佳地利用系统的各项资源。
> 在软件的开发、部署、运行、维护、升级换代上，为系统提供最大的灵活性；
> 为系统提供最大的安全性、稳定性、可靠性，以及其他各项质量要素；

架构描述的第一个作用和意义，就是要站在系统构成的角度（注意：不是业务需求的角度）。

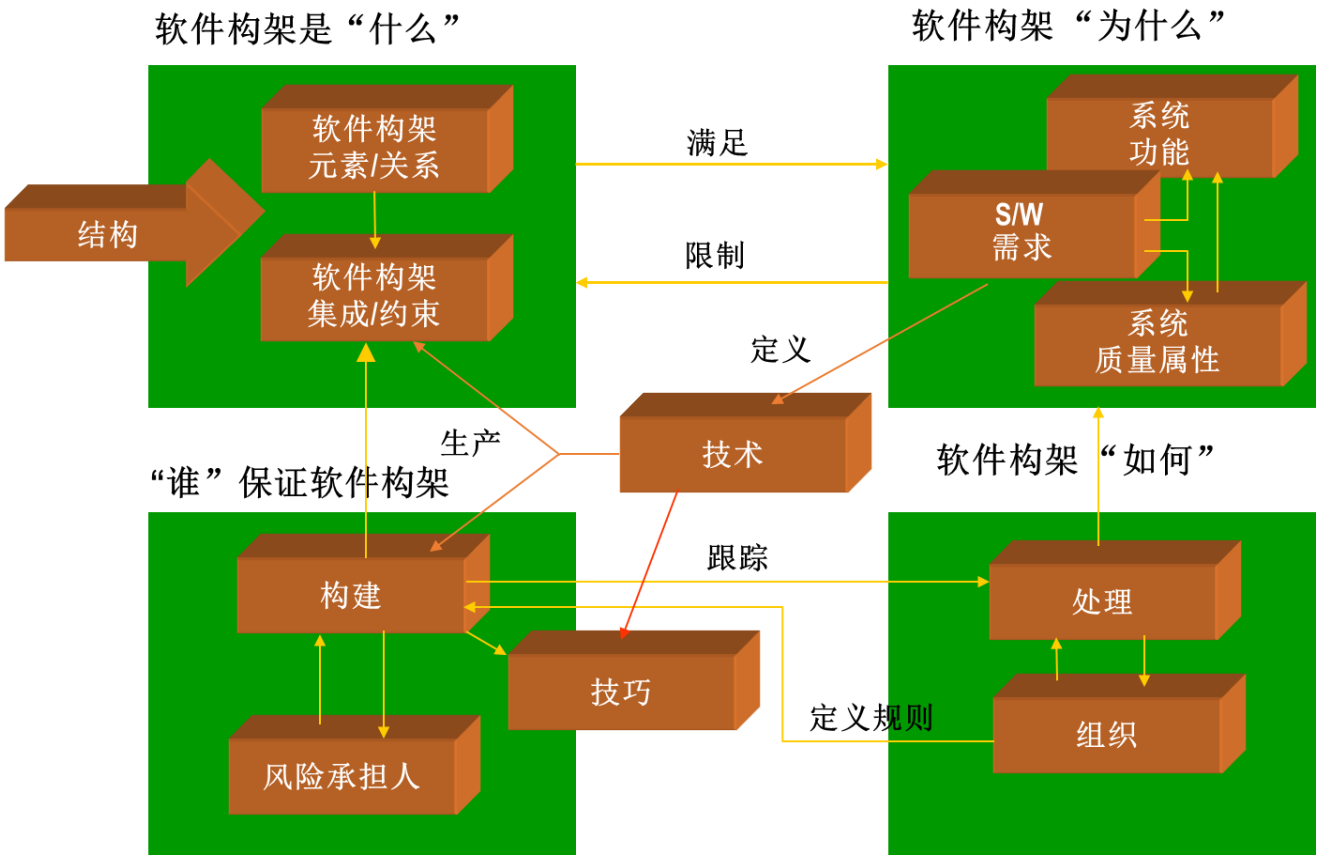
架构描述表达了软件系统的实现结构

在这个案例中，架构（确切地说，应该是有关架构的描述）的作用，是为系统设计者提供了一个记录、分析、比较、提出改进系统设计方案的一个“蓝图”，一个可讨论的“方案”，也是以后实现、测试、验证，以及进行过程管理的“承诺”和依据。

在上节描述的系统架构中，可以看到的“蓝图”和“方案”是：
p1、有关组件：
p2、有关组件之间的连接：
p3、有关连接关系：

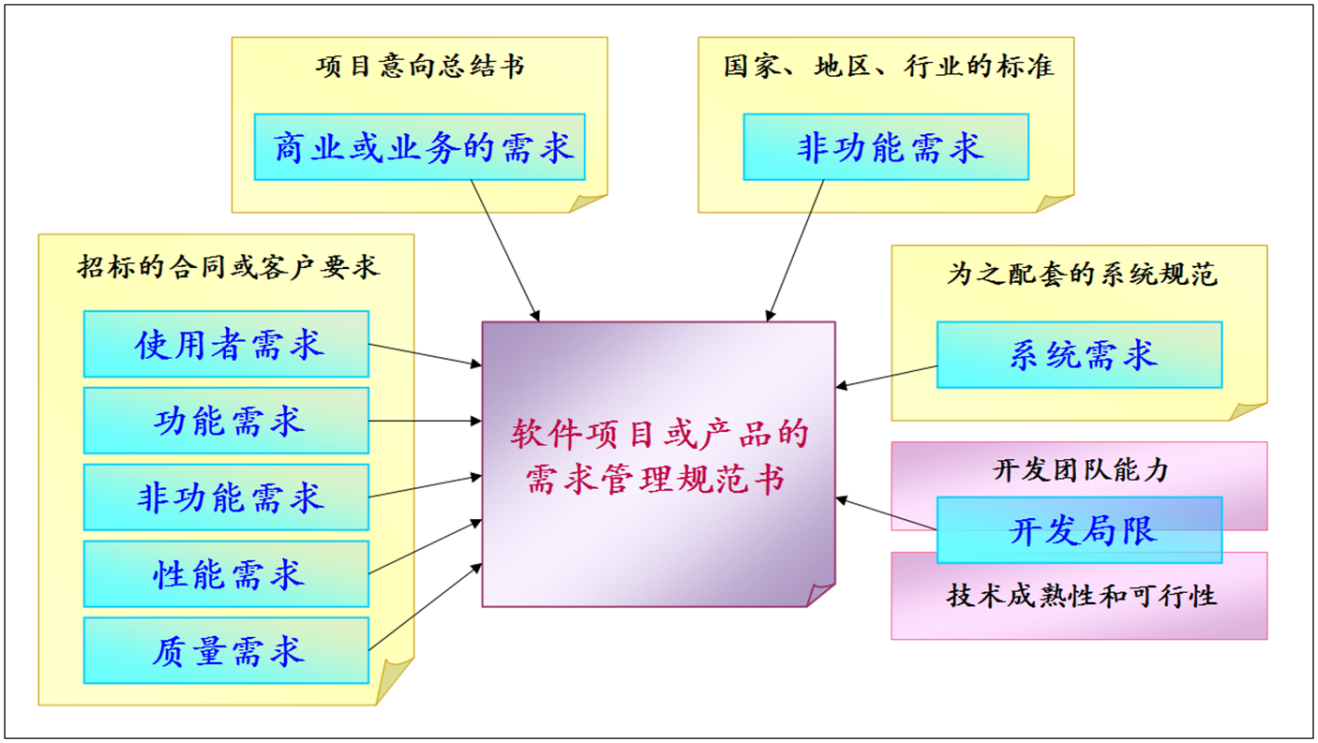
架构描述的最主要的任务是，记录了系统的设计想法，具体表现为，将这些想法，按实现模块（而不是按功能），分解为具体的组件，并将这些组件的连接方式、连接关系，描述出来，让使用者知道，未来系统的架构。

架构描述对需求实现的作用和意义



架构是系统关键质量属性的描述

软件系统的关键质量属性需求



用质量属性的选择帮助确定项目的需求范围

利用质量标准的互相对立和制约因素来决定开发工作的侧重点和优先权

对客户重要的质量标志 对开发者重要的质量标志	可靠性	效率性	灵活性	完整性	兼容性	可维护性	多用转换性	稳定性	重复使用性	健全性	可测性	可用性
可靠性 (Availability)								+		+		
效率性 (Efficiency)			–		–	–	–	–		–	–	–
灵活性 (Flexibility)		–		–		+	+	+			+	
完整性 (Integrity)		–			–				–		–	–
兼容性 (Interoperability)		–	+	–			+					
可维护性 (Maintainability)	+	–	+					+			+	
多用转换性 (Portability)		–	+		+	–			+		+	–
稳定性 (Reliability)	+	–	+			+				+	+	+
重复使用性 (Reusability)		–	+	–	+	+	+	–			+	
健全性 (Robustness)	+	–						+				+
可测性 (Testability)	+	–	+			+		+				+
可用性 (Usability)		–								+	–	

影响构架的关键需求

软件构架师所考虑的设计需求

•与系统设计本身直接有关的设计需求：

–概念完整性

–理解正确性

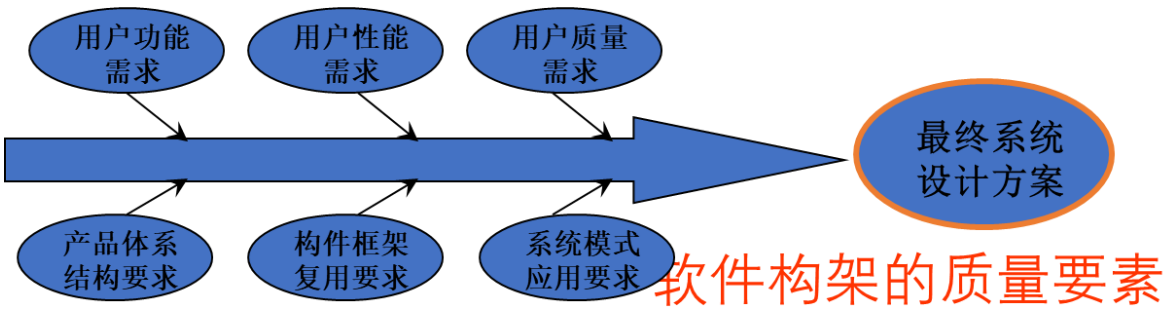
–设计完备性
–可构建性

•高质量软件系统设计的质量需求：
–用户需求目标的适应性
–可维护性、可移植性、可测试性、可追踪性、正确性、健壮性（鲁棒性）

决定软件构架的关键需求

开发期质量要素：
•易理解性
•可扩展性
•可重用性
•可修改性
•可移植性
•可集成性
•可测试性

运行期质量要素：
•功能
•性能
•安全性
•易用性
•持续可用性
•可伸缩性
•互操作性
•可靠性
•健壮性



软件构架的质量要素

1985年，国际标准化组织（ISO）建议，软件质量度量模型由三层组成。高层称软件质量需求评价准则（SQRC），中层称软件质量设计评价准则（SQDC），低层称软件质量度量评价准则（SQMC）

ISO 的质量要素与评价准则

关系 / 要素 / 准则	正确性	可容性	有效性	安全性	可用性	可维护性	灵活性	可互操作性
可追踪性	√							
完全性	√							
一致性	√							
准确性		√						
容错性		√						
简单性		√				√		
模块性						√	√	
通用性							√	
可扩充性								
检测性						√		
自描述性						√	√	
执行效率			√					
存储效率			√					
存取控制				√				
存取审查				√				
可操作性					√			
易培训性					√			
通信性					√			
软件系统独立性							√	
硬件独立性							√	
通信通用性								
数据通用性								
简明性						√		

软件质量需求评价准则（SQRC）

软件质量设计评价准则（SQDC）

好的软件构架的特征

灵活、具有可伸缩性
考虑全面并可扩展
思路简单明了、直接可以理解
结构划分和关系定义清楚
模块职责明确和分布合理
效益和技术综合平衡

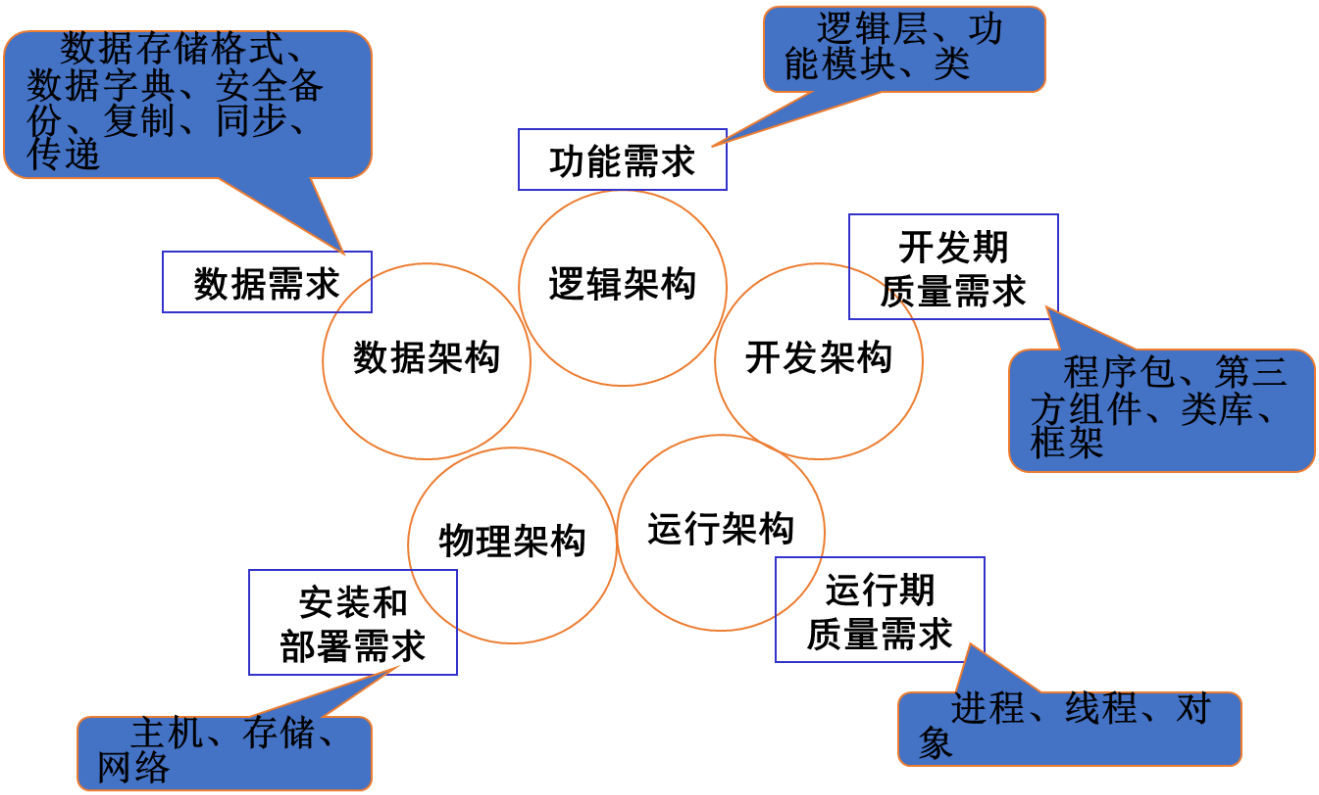
差劲的软件构架的特征

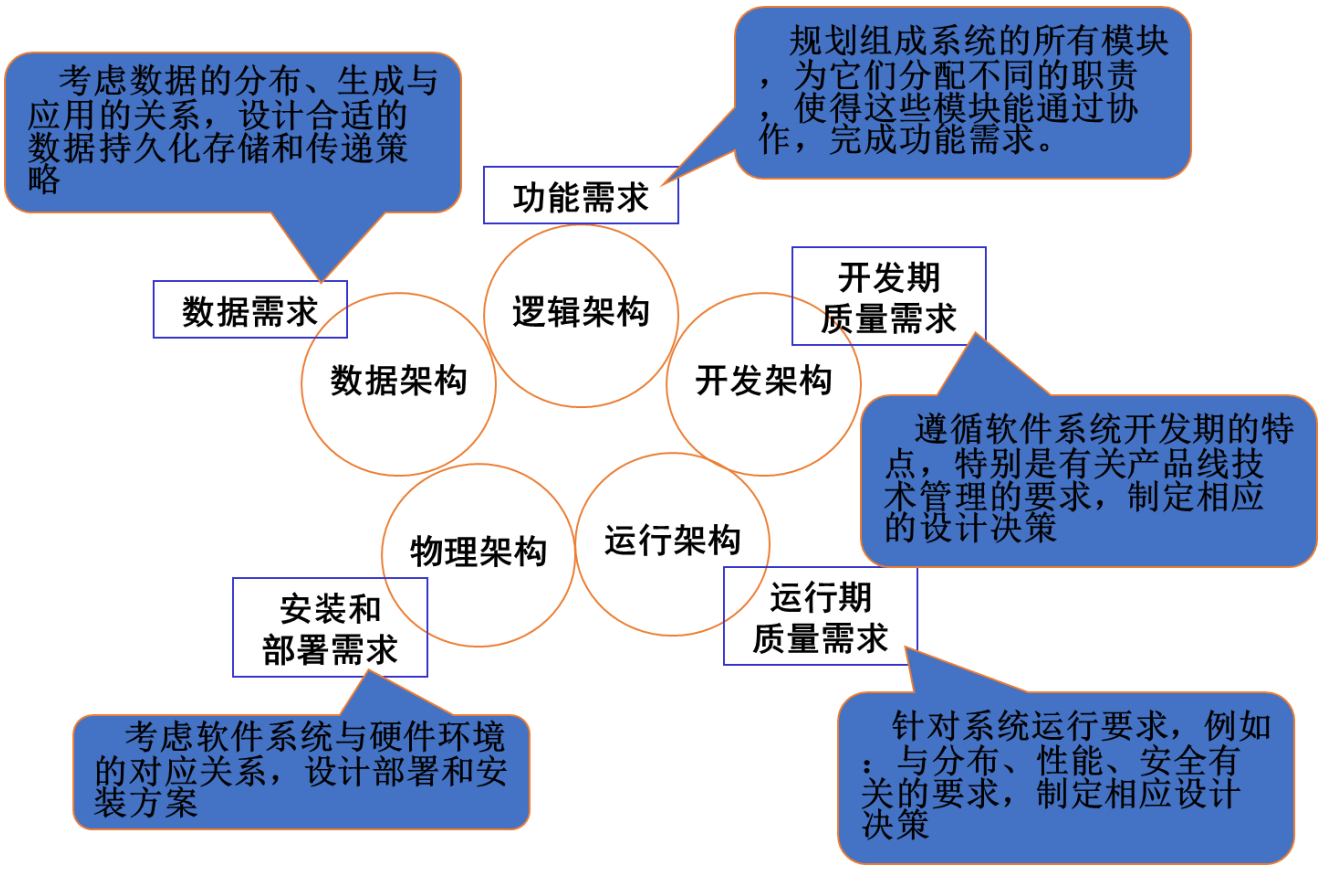
为什么许多大型软件应用系统推出不久，就面临需要重新设计（美其名曰：架构重组）？往往不是系统不能用（不是功能性问题）,而是系统性能问题：
(非功能性需求)
系统业务处理逻辑缺乏灵活性
维护困难
功能无法扩展
运行速度太慢
稳定性差、甚至经常宕机

软件架构的描述与可视化分析

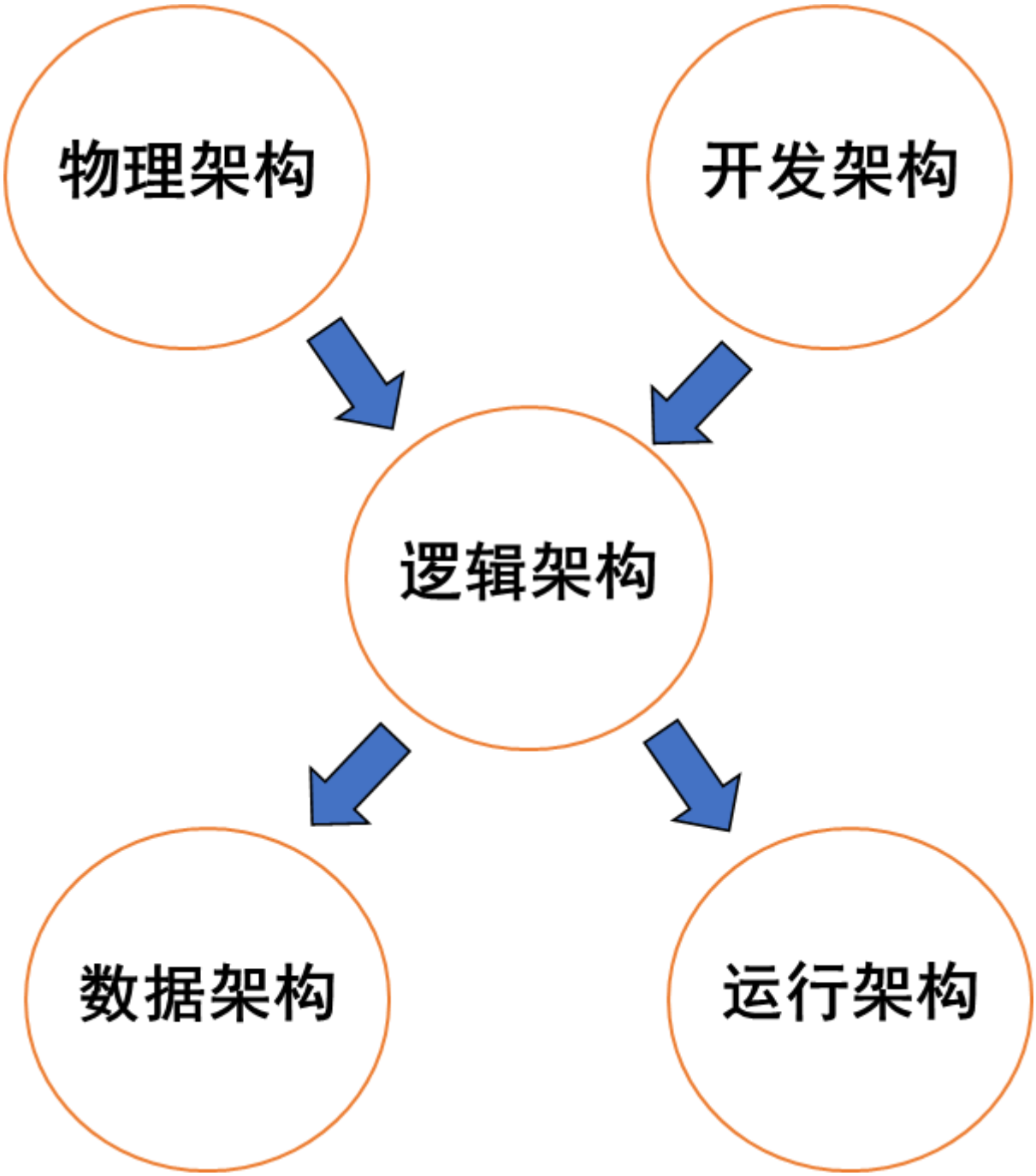
架构描述与UML架构视图

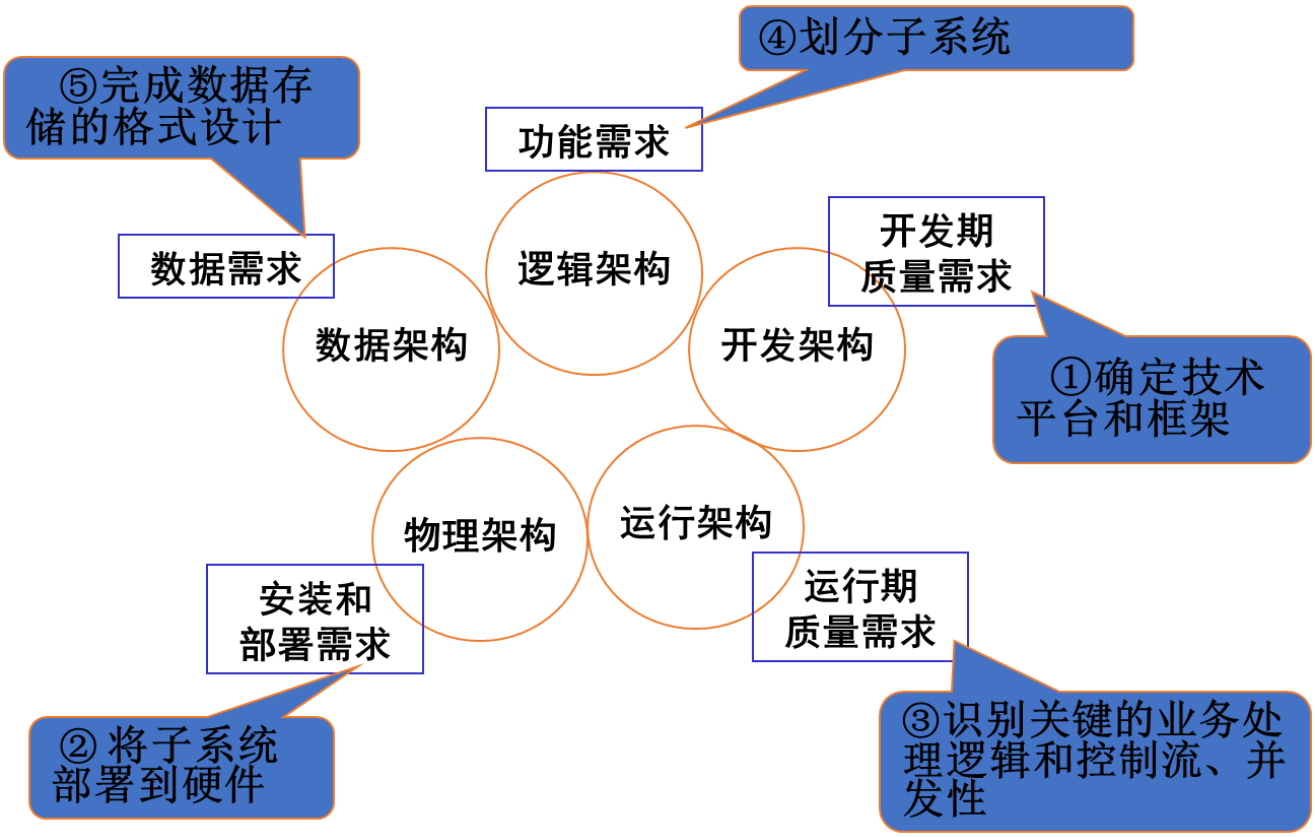
五个基本架构





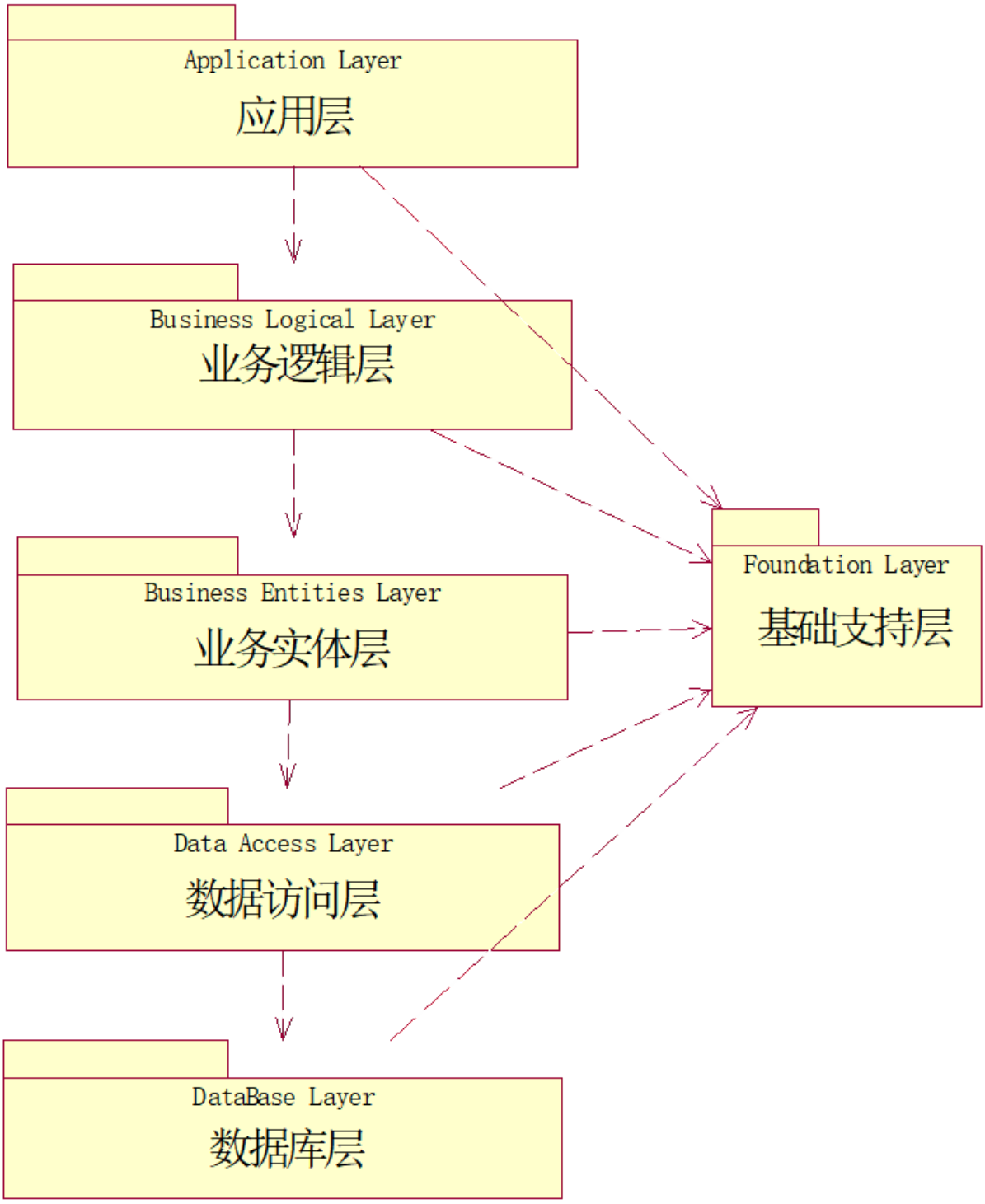
软件构架设计的多重考虑


物理架构
开发架构
逻辑架构
数据架构
运行架构

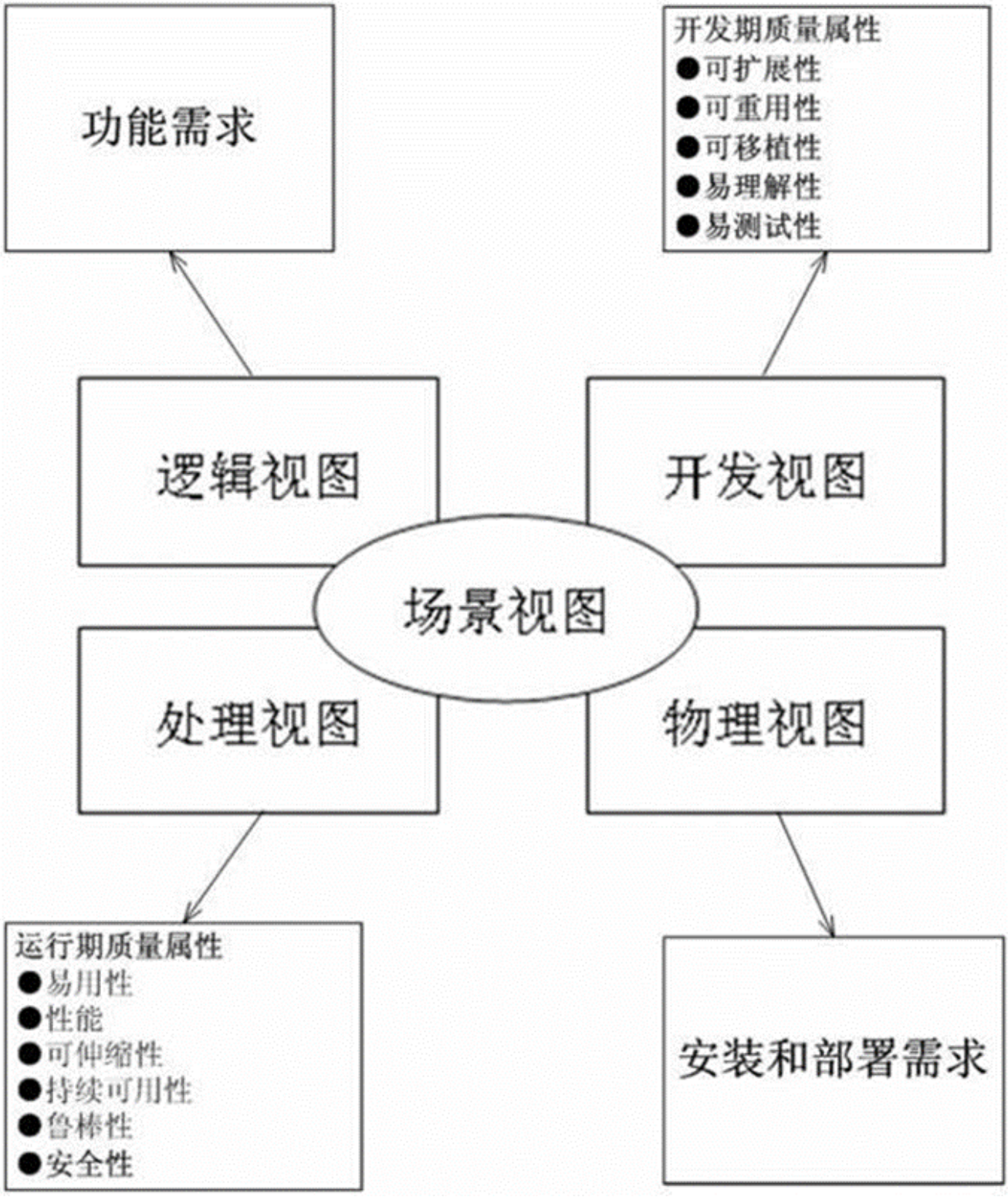


•面向应用的系统构架从上而下，分为：
•用户界面层
•业务逻辑层
•业务实体层
•数据访问层

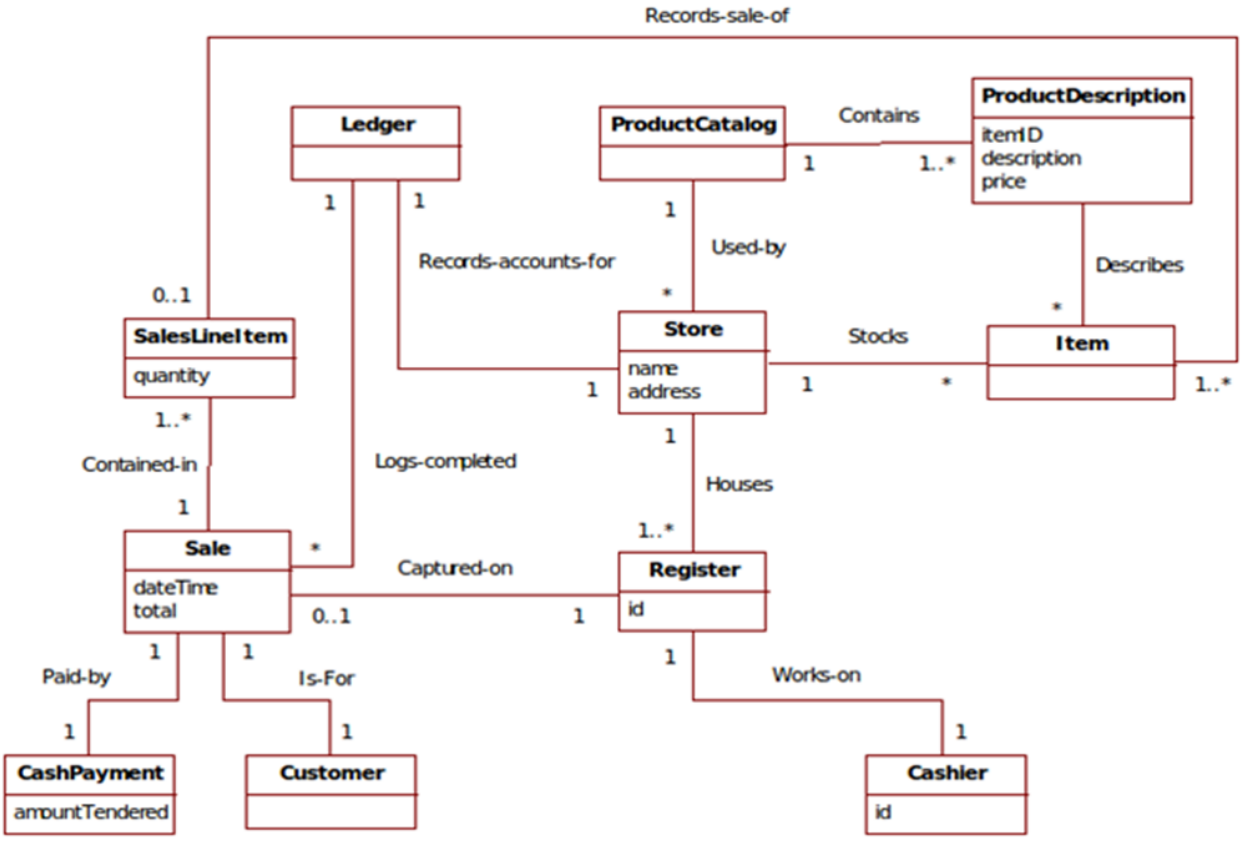
•数据存储层



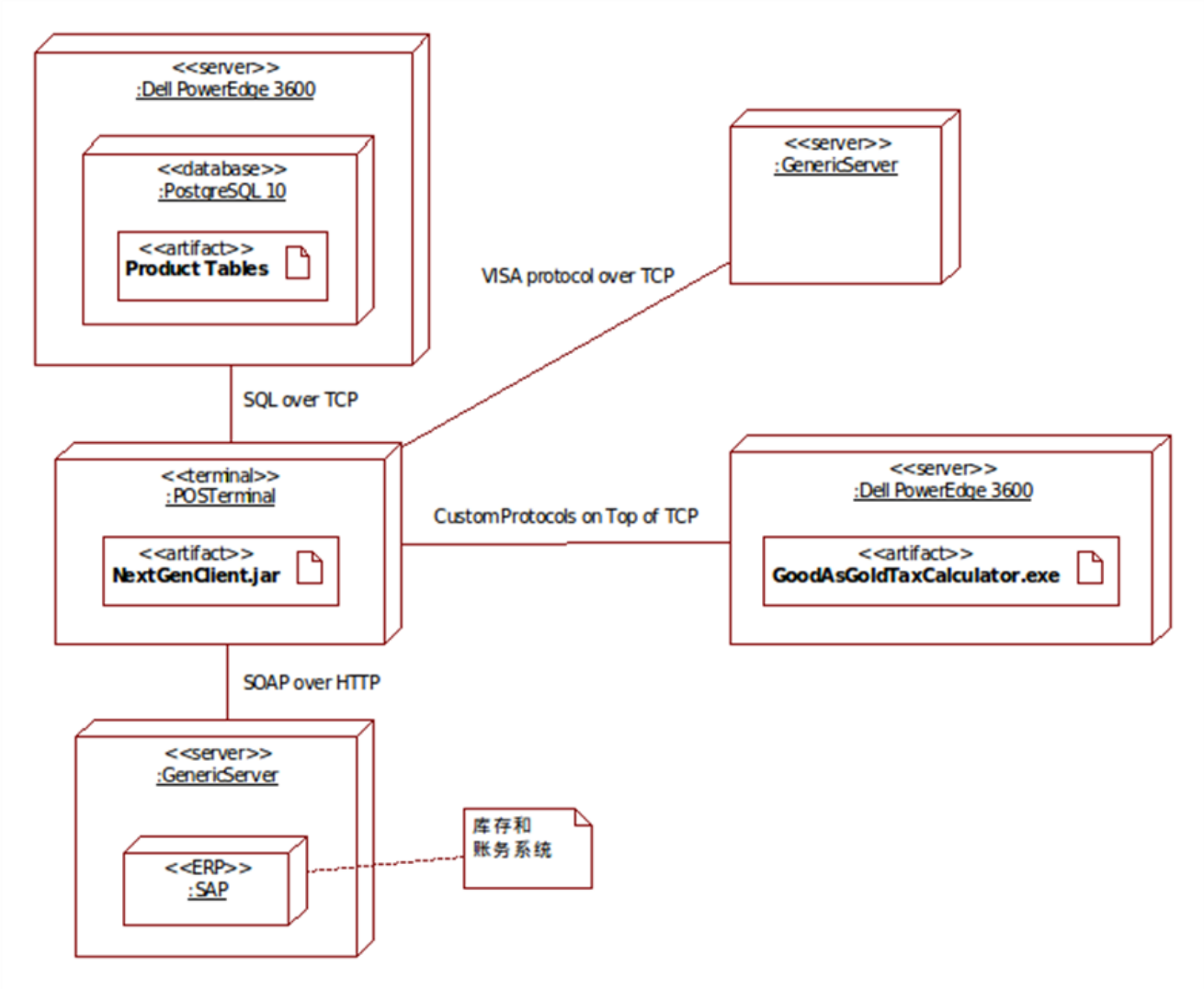
UML 4+1的架构描述


功能需求
开发期质量属性
●可扩展性
●可重用性
●可移植性
●易理解性
●易测试性
逻辑视图
开发视图
场景视图
处理视图
物理视图
运行期质量属性
●易用性
●性能
●可伸缩性
●持续可用性
●鲁棒性
●安全性
安装和部署需求

4+1之1：逻辑架构视图

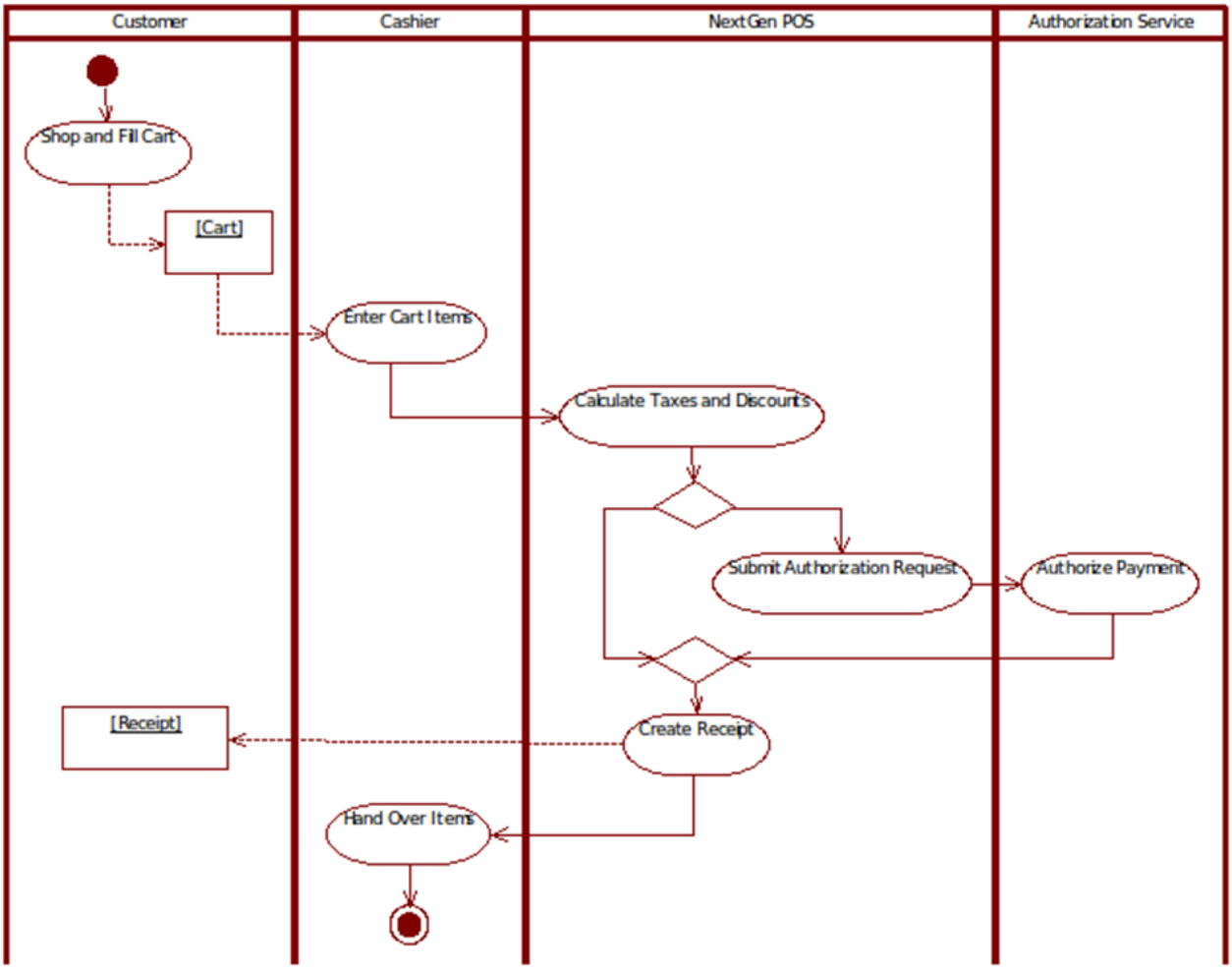

Records-sale-of
Ledger
ProductCatalog
Contains
ProductDescription
itemID
description
price
1
1..*
1
1
1
Records-accounts-for
Used-by
Describes
0..1
*
*
SalesLineItem
quantity
Store
name
address
Stocks
Item
1..*
1
1
*
1..*
Contained-in
Logs-completed
Houses
1
1..*
Sale
dateTime
total
*
Captured-on
Register
id
0..1
1
1
1
1
Paid-by
Is-For
Works-on
1
1
1
CashPayment
amountTendered
Customer
Cashier
id

4+1之2：开发架构视图

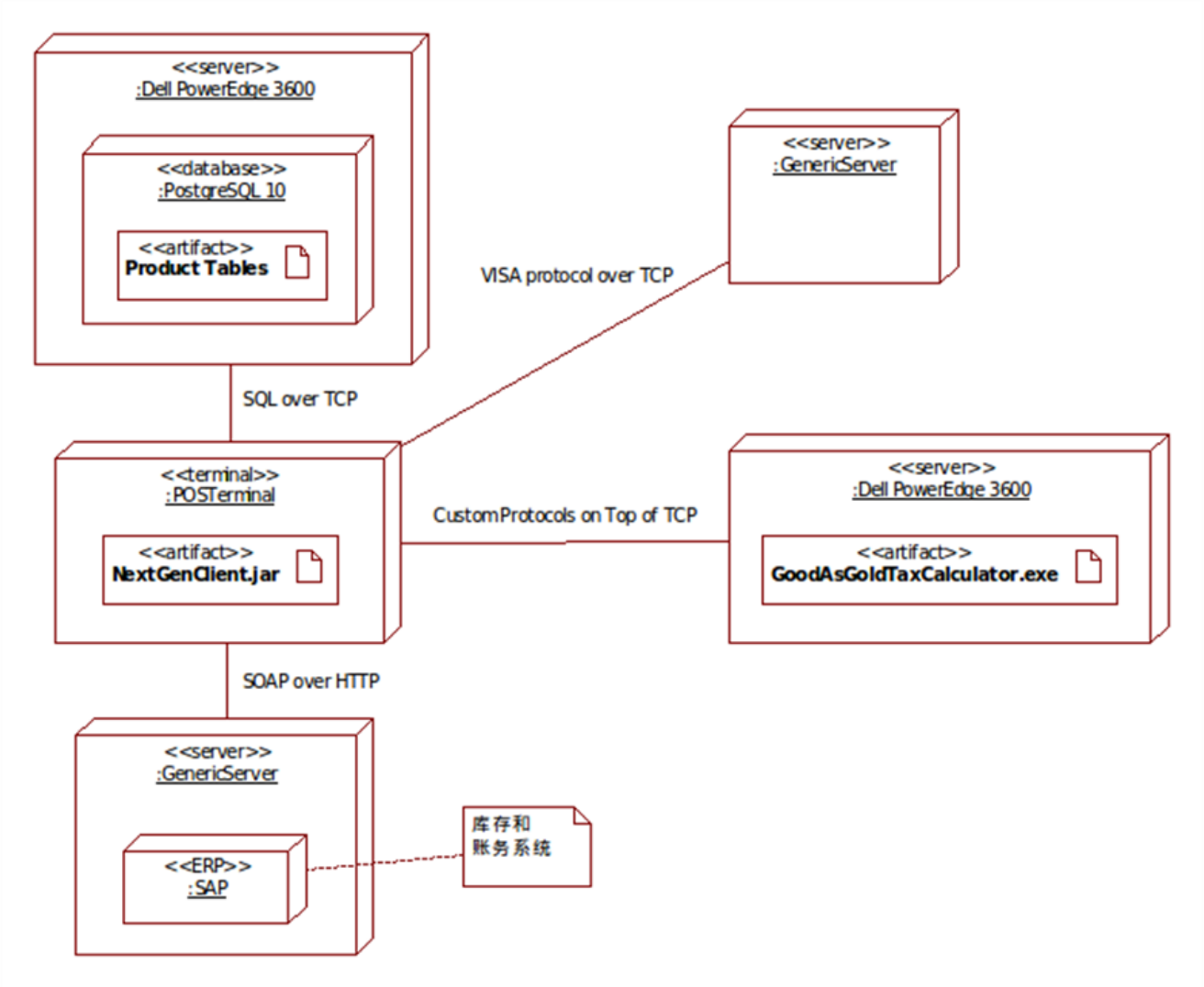


简单对象访问协议SOAP（Simple Object Access Protocol），是交换数据的一种协议规范，是一种轻量的、简单的、基于XML的协议，它被设计成在WEB上交换结构化的和固化的信息。

4+1之3：进程架构视图


Customer
Cashier
NextGen POS
Authorization Service
Shop and Fill Cart
[Cart]
Enter Cart Items
Calculate Taxes and Discounts
Submit Authorization Request
Authorize Payment
[Receipt]
Create Receipt
Hand Over Items

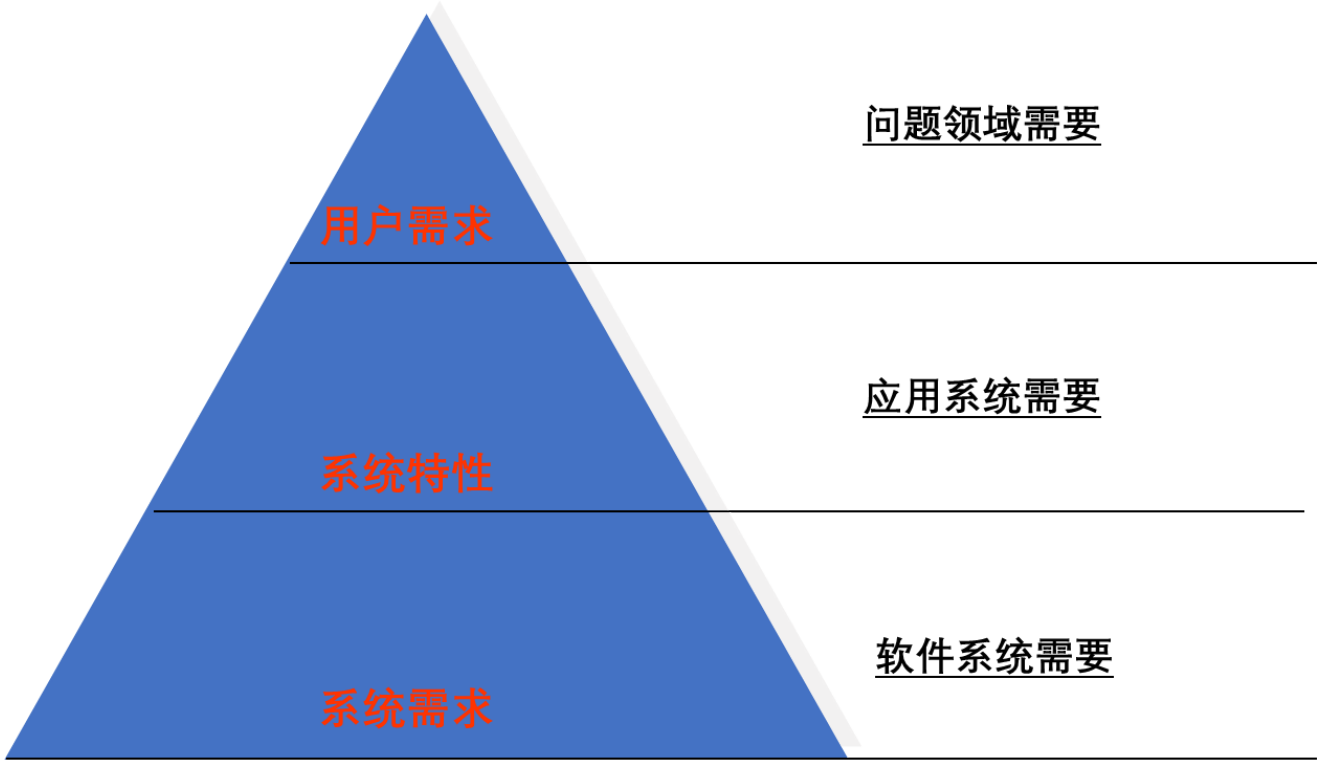
4+1之4：部署架构视图



从需求到架构

架构师的需求过程

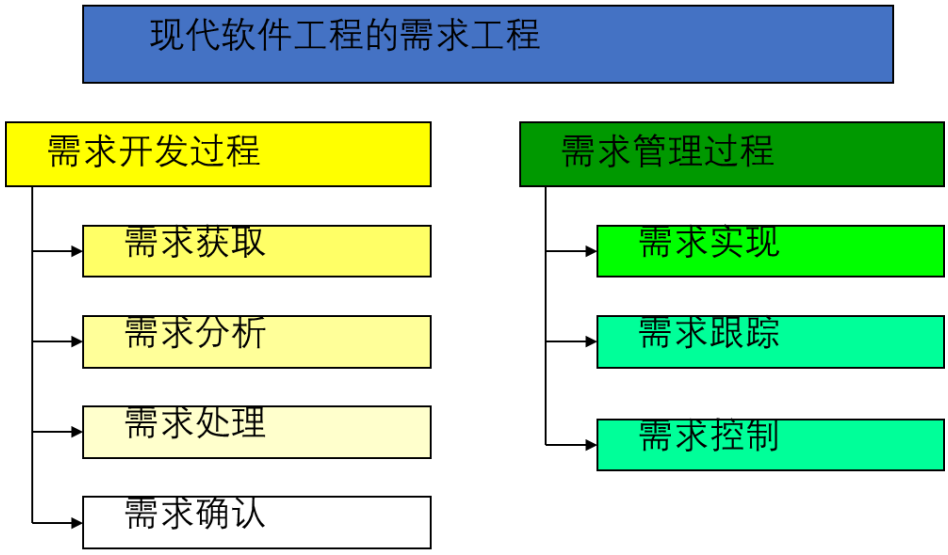
现代软件工程的需求过程



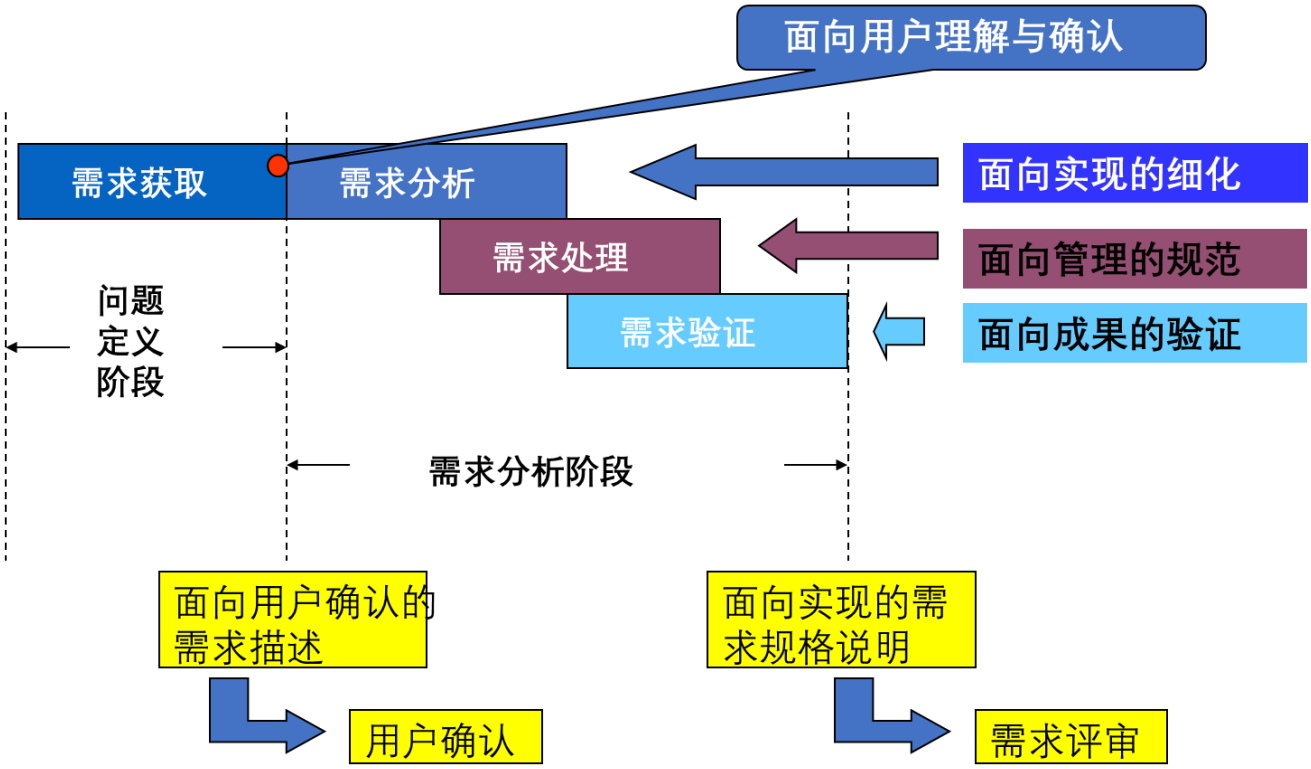
需求的分解

需求工程是提供一种适当的机制，以了解用户想要什么、分析需求、评估可行性、协商合理的解决方案、无歧义地规约解决方案、确认规约以及在开发过程中管理这些被确认的需求规约的过程。

需求工程的两大过程域



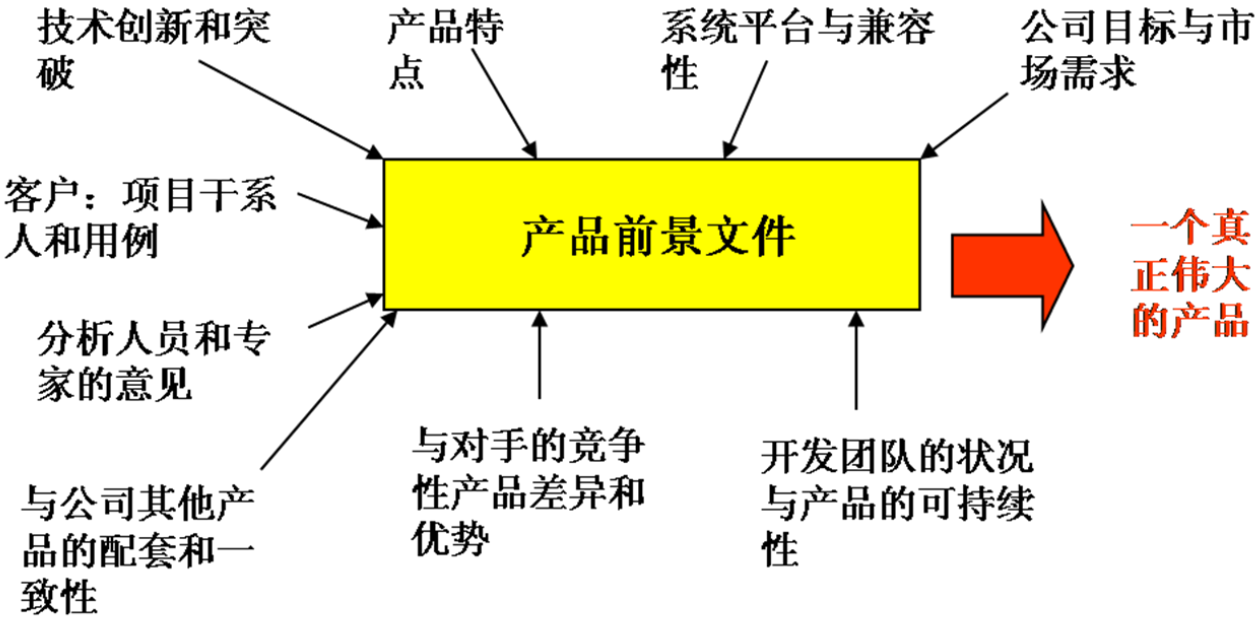
需求开发过程的阶段任务\需求开发过程的重要里程碑



软件需求获取与架构师的关注点

1、需求获取阶段的工作目标与关键交付物成果：
•参与需求诱导活动的客户、用户和其他干系人的名单；
•系统目标与核心关注；
•系统技术环境的描述；
•系统和产品范围的限定性假设；
•功能点列表及相应的假设和限制；
•需求和可行性的描述。
2、项目团队在需求获取阶段的关注点,在需求获取阶段，项目团队的最后目标是获得用户的需求确认。需求获取阶段的关注点是：
•在问题定义上与用户达成共识；
•理解问题背后的根本原因；
•确定用户和项目干系人；
•定义问题解空间的边界；
•确定问题解决方案的约束和假设；
•最终阶段完成标志：用户对系统目标的认可——签字。

3、需求获取阶段架构师的关注点：



需求获取阶段的产品管理考虑

软件需求分析与架构师的关注点

1、需求分析阶段的工作目标与关键交付物成果：

需求分析阶段的任务是面向系统实现（最主要的是面向架构设计，而不是代码）、严格对系统的需求，进行再分析。包括：

(1)通过仔细地分析系统的输入、输出、功能、属性以及系统的构成环境，决定系统构成的完整集合。
(2)需求分析主要是面向系统架构及其实现的，所讨论的是在技术上，系统应该如何构建并做什么。
(3)需求分析将引导架构设计（正向作用），也受架构设计方法的制约（反向作用）。
(4)所以，需求分析阶段是架构师介入项目需求开发的最主要阶段。需求分析的验收标志是组织的需求评审，其性质和内容，与需求获取阶段的用户确认，有本质的不同。

1、需求分析阶段的工作目标与关键交付物成果：

p因此，在需求开发过程中，需求分析工作的本质是细化系统定义。

p在需求获取阶段，已经通过建立业务模型、系统模型，与用户共同确定了系统的特性。这些基本特征包括：

p业务模型：可以映射出软件产品核心的需求，包括与业务功能对应的组织的特性（如：组织的业务流程是否做相应改变），与业务流程对应的内外部关系特性（如：相应的岗位职责是否做调整）等；

p系统用例模型：是在已经建立的业务模型的基础上，建立的系统模型，是高度抽象的系统描述；

p用例：业务模型和系统模型的最典型表示形式，是用户通过系统将获得的应用体验。

p需求分析是以架构设计为目标，或者是从已有架构平台出发，反向迭代、补充分析已经得到的用例—细化用例。

2、需求分析阶段架构师的关注点：

p软件产品本身除了用户功能需求以外，可能还存在与用户业务过程没有直接关系的非功能性需求，如：与硬件、软件环境相关的操作系统和软件平台要求、对软件运行的远端监控要求、异常处理（如通信连接中断等非业务异常）、响应时间和负载能力要求等等。

p另一方面，组织的或产品的设计约束和限制，也是系统需求必须要考虑的内容。通常这三部分需求，构成了软件需求的总集。

p后二个部分是架构师“新增加”的需求。

p架构师的责任是保证这三部分的需求，能够合适地“糅合”在一起。

“统一平台”需求与实现方案分析

1、网络级的重用（无软件重用）

p客户端（C/S-B/S）：运行各自的客户端软件

p服务器：按网络端口，运行不同的服务器软件

2、客户端/服务器非游戏部分可重用

p客户端（C/S-B/S）：除登录外，运行各自的客户端软件

p服务器：用户登录、用户信息管理等部分可重用，游戏对弈的部分，运行各自的代码。

- ✓五子棋二次开发的需求分析例子：
  - ✓需求分析：细化五子棋与象棋的用例（差异）
- ✓需求分析师的不同关注点：
  - ✓功能差异：棋盘、判胜负
  - ✓业务流程差异：没有（相对独立）
  - ✓业务实体差异：没有（相对独立）
- ✓架构师的不同关注点：
  - ✓源码的模块划分（功能独立性）——可重用性
  - ✓架构层次与功能的划分——可重用度
  - ✓扩展——重用度的追求
- ✓共同关注点——差异
  - ✓需求：需求的功能差异
  - ✓架构：源码的架构差异

✓五子棋二次开发的需求分析例子：

- ✓需求分析阶段不是做架构设计
- ✓只是把与架构有关的“关键需求”提出来
  - ✓分析
  - ✓平衡
  - ✓决策（取舍）
    - ✓与用户有关
    - ✓与技术限制和可行性有关
    - ✓与可采取的技术方案有关
    - ✓与时间、成本、产品/技术方向、风险有关

软件架构的概要设计与实现

架构结构

1.模块结构–体现了任务的划分，每个模块有其接口描述、代码和测试计划等，各模块通过父子关系联系起来，在开发和维护阶段用于分配任务和资源。
2.逻辑结构–系统功能需求的抽象，功能图。
3.物理结构–软件与硬件之间的映射关系，在分布式或并行系统中有重要意义。
4.进程结构–运行系统的动态特征，包括进程间的同步关系、缺少不能运行、存在不能运行、先后等关系，与模块结构、概念结构成垂直正交关系。
5.数据流–模块之间可能发送数据的关系，最适合用于系统需求的追踪
6.控制流–程序、模块或系统状态之间的“之后激活”的关系，适合于对系统功能行为和时序关系的验证。
7.类结构–对象之间的继承或实例关系。
8.使用结构–描述过程或模块之间的联系，这种联系是“假设正确存在”的关系，用于设计可轻松扩展的系统。
如果过程A的运行必须以过程B的正确运行为前提，则说过程A使用过程B。
9. 层次结构–是一种特殊的使用结构，层就是相关功能的一致集合，在严格的分层结构中，第n层仅能使用第n-1层提供的服务。
10. 调用结构--（子）过程之间调用和被调用的关系，可用来跟踪系统的执行过程。

传统系统设计的基本思路和方法

思路

软件构架设计经历了40年的发展演化过程：

1、早期的模块化程序设计和自顶向下逐步求精的结构化方法—面向结构的方法（功能模型）
2、由数据库技术的发展，带来的将数据流和数据结构，转化为设计定义的方法（E-R模型）—面向数据的方法
3、面向对象的方法—从对象及其关系导出系统的方法（对象模型）
4、现在的方法：软件设计的重点，已经移到软件构架和实现软件构架的设计模式

软件结构始终是软件研究的重要课题，而研究的层次和侧重点，从最基本和底层，向越来越抽象层次发展，但结构的基本思想仍然有指导意义。

抽象与求精的设计方法

1、抽象：

•抽象是常见的思考问题的方法，抽象的目的是为了专注主要方面，避开细节。在软件开发的不同阶段，可以有不同的抽象。例如：

（1）需求分析的抽象—用问题域的语言描述需求，而不是用计算机的语言表达，避开了因涉及过多计算机实现细节带来的干扰，抽象级别最高；

（2）系统设计的抽象—分别采用面向问题域和面向实现域的术语，来描述解决方案，面向实现的语言抽象级别降低；
（3）编码阶段的抽象—采用高级语言，实现系统设计。高级语言是对机器执行代码的抽象，抽象级别最低。

数据抽象：

是描述数据对象的一个命名的数据集合，与过程抽象一样，也有从高级到低级的层次。如：在讨论业务流、数据流时，只描述对特定文件、数据库表的操作等，如：读用户文件、判断用户权限。这里并不涉及具体操作数据域，抽象关注的是业务流程，而不是具体操作的数据内容。

过程抽象：

•指具有特定功能的一个被“命名的”指令序列。在高抽象级别中，它可能就是一个过程名，不需要具体实现细节。在低抽象层次中，它可以是一段抽象的“宏”描述。

抽象层次：

（1）体系结构在不同的阶段，依据层次和细节的不同，分为概略型、需求型和设计型
（2）概略型是上层宏观的描述，反映系统最上层的部件和连接关系
（3）需求型是对概略结构的深入表达，以满足用户功能和非功能需求的表达为主
（4）设计型从设计实现的角度，对需求型进行更深入的描述表达，需要从不同的侧面/视图，设计系统的各个层面的各个部件和连接结构。在这个层面上的描述，将直接为系统实现和性能分析服务.

案例：一个可以实现二维绘图的计算机辅助设计CAD软件

抽象级别1：需求描述

该软件应具有以下功能：

- ✓提供一个所见既所得的绘图用户界面；
- ✓提供一个数字化仪界面，用以代替绘图板和丁字尺；
- ✓能实现各种类型的直线、矩形、圆及曲线的绘制；
- ✓可进行几何计算、视图或剖面图处理；
- ✓设计结果以图形文件形式存储。

✓显然，这个抽象是问题域术语描述的。

抽象级别2：系统任务描述

CAD软件的任务包括：

- ✓系统用户交互；
- ✓二维图形创建与输入；
- ✓图形显示与处理；
- ✓图形文件管理。

这个抽象，已经比较接近计算机术语，而不是用户术语，但仍然还不是实现术语。

抽象级别3：过程描述（部分）

```
PROCEDURE 二维图形创建
  REPEAT 下列操作 UNTIL 图形创建完毕:
        DO WHILE 需要与数字化仪交互时
                数字化仪接口处理
           CASE 绘图请求 OF
             直线：直线绘图处理;
             矩形：矩形绘图处理;
             圆 ： 圆绘图处理;
              ……
           END CASE
        END DO
  END REPEAT
END PROCEDURE
```

```
DO WHILE 需要与键盘交互时
     键盘交互处理
        CASE 分析/计算 OF
          视  图：视图处理;
        剖面图：剖面图处理;
          计  算：计算处理;
                    ……
        END CASE
END DO
```

这个实现描述是“宏”描述，通常，在系统详细设计时，需要达到这种详细程度。

模块化和信息隐蔽

模块的概念

1.定义
同时具有以下四个要素的一组语句称为一个模块
四个要素是：
输入/输出
逻辑功能
运行程序
内部数据
前两个称为模块的外部要素，后两个为内部要素。
2.模块的属性
p模块具有四种属性：
p一个模块的输入/输出都是指向同一个调用者。
p模块的逻辑功能是指模块能够做什么事，表达了模块把输入转换成输出的功能。
p模块的运行程序指模块如何用程序实现其逻辑功能的过程。
p模块的数据指属于模块自己的数据。

在结构化系统设计中，人们主要关心的是模块的外部要素（输入/输出、逻辑功能），而内部要素（运行程序、内部数据），将在系统实现/编码时完成

模块分解的假设：

有两个函数：C(x)表示问题x的复杂程度；E(x)表示解决问题x所需要的工作量（时间）。

对于两个问题P1和P2，如果：C(P1)＞C(P2) 则：E(P1)＞E(P2)

如果问题复杂，则解决问题的工作量也大

另一个有趣的特性是：C（P1＋P2）＞C（P1）＋C（P2）

问题分解，有利于降低问题的总的复杂度

根据前面的结论，我们可以得出下面的不等式：

E(P1＋P2)＞E(P1)＋E(P2)

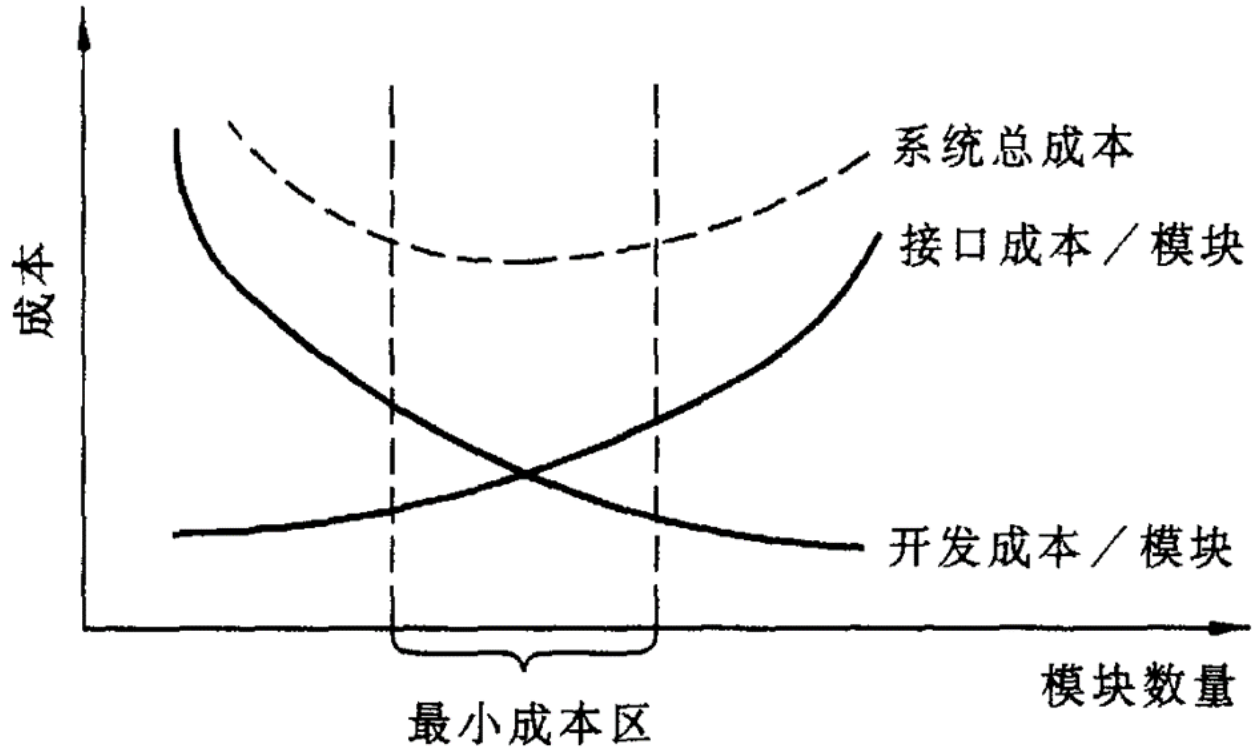
所以，问题分解既可以降低问题的复杂程度，也可以减少总的工作量。

这种“分而治之”的思想提供了模块化的根据：把复杂的问题分解成许多容易解决的小问题，原来的问题也就容易解决了。

但是，过分的分细模块，也会带来集成的复杂度，因此，需要选择一个适度平衡的方案。

模块数量与软件开发复杂性

不是把问题细分的越小越好，有一个最小成本区



模块数量与系统成本的关系

如何确保模块数量，落在“最小成本区”内?

p模块中所包含的信息（数据、过程）对不需要这些信息的其他模块，是透明的（不可见、不可访问）
p涉及这些信息的改变，都只局限在模块内部，不会影响到其他模块
p抽象帮助定义组成软件系统的过程实体、而隐蔽机制则通过对模块内部访问的约束，有助于模块的分离和实现
p信息隐蔽是系统设计的重要原则，也是系统实现（并行开发）、测试（问题分离）、后期维护（局部维护变更）的重要方法

p每个理想模块只解决一个问题。

p每个理想模块的功能都应该明确，使人（开发者、使用者、维护者）容易理解。

p理想模块之间的联结关系简单，具有独立性。

p由理想模块构成的系统，使人容易理解、编程、测试、修改和维护。对模块的使用者来说，其感兴趣是模块的功能，而不必去理解模块内部的结构和原理。（卖萝卜青菜的也可以卖电脑了）

对理想模块独立性设计的最经典描述：既插既用

模块偶合与内聚的追求

模块独立性的度量
度量模块独立性的尺度：偶合性与内聚度
偶合度：
模块设计的基本原则：降低系统中模块之间的联结程度（偶合性），提高每个模块的独立性
模块按不同的需要而偶合在一起，偶合有七种形式，偶合度是度量模块间紧密程度的尺度

模块的七种偶合形式
两个模块之间的偶合有七种形式，按照偶合紧密程度由低到高排列为：
非直接偶合
数据偶合
标记偶合
控制偶合
外部偶合
公共偶合
内容偶合

模块的七种偶合形式的比较

联结方式	对连锁反应的影响	可修改性	可读性	通用性
非直接偶合	最弱	好	好	好
数据偶合	弱	好	好	好
标记偶合	弱	中	中	中
控制偶合	中	不好	不好	不好
外部偶合	较强	不好	坏	坏
公共偶合	强	不好	最坏	最坏
内容偶合	最强	最坏	最坏	最坏

设计模块时，应以数据偶合为主，辅以标记偶合、外部偶合和控制偶合，消除公共偶合和内容偶合。
如果对模块间的偶合关系没有把握的话，就尽量把模块打小，以增加系统一定的控制复杂度的方式，降低模块的偶合度，可能是值得的。

模块的内聚度1

•模块的内聚度是指：一个模块内部的各个组成部分的紧密程度，其处理动作的组合强度等，它是衡量模块独立性的标准。
G.myers定义的内聚的七种形式：按照内聚度从高到低：

1.功能内聚

p如果一个模块内部的各组成部分的处理动作全都为执行同一个功能或实现单一的目标而存在，并且只执行一个功能，则称为功能聚合。

p判断一个模块是不是功能聚合，只要看这个模块是“做什么”，是完成一个具体的任务，还是完成多任务。

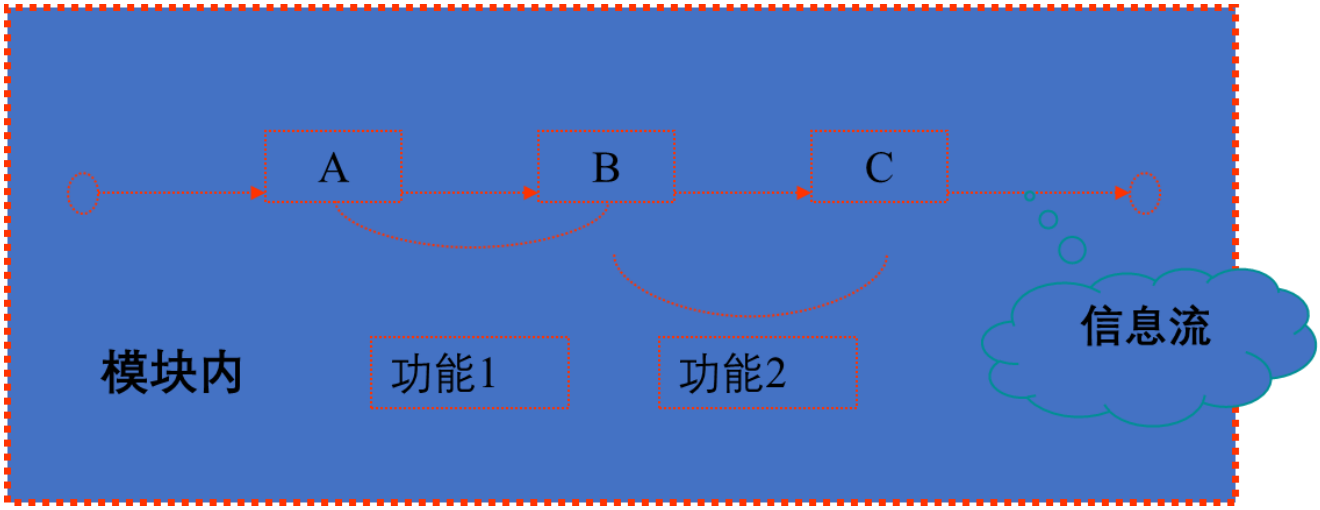
p功能内聚的模块易于复用、易于维护

2.信息内聚

p如果一个模块要完成多个功能，而每个功能都有各自的入/出口和独立的代码，但功能实现依赖相同的数据结构，则称为信息内聚。

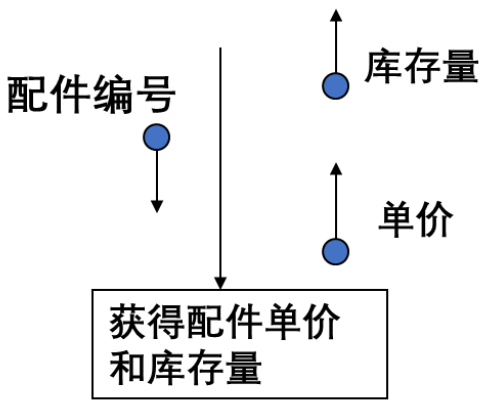
p通常，信息内聚的各个功能处理，还具有顺序处理的特点，既：前一个处理动作所产生的输出数据是后一个处理动作的输入数据，因此，也称为顺序内聚。

p信息（顺序）内聚比功能内聚复杂，修改数据结构涉及到多个功能实现，修改模块中的一个功能，可能会影响到同一个模块中的下一个功能。



3.通信内聚

一个模块要完成多个功能，每个功能的完成使用的是相同的模块的（而不是模块内上一个功能的）输入数据或相同的输出数据，称为通信内聚。



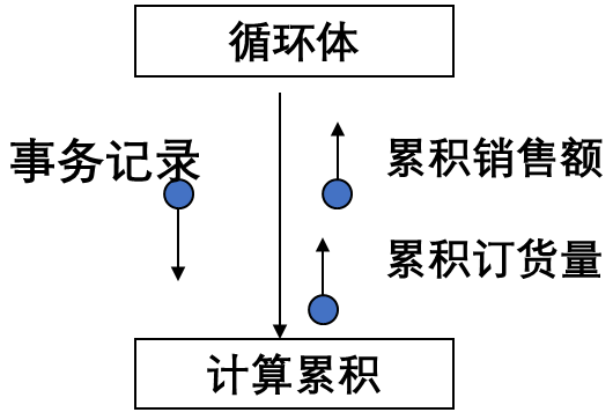
两个工作：
1.按配件编号查询“数据存储”，获得单价。
2.按配件编号查询“数据存储”，获得库存量。

这两个处理动作都使用相同的输入数据

4.过程内聚

如果一个模块内部的各个组成部分的处理动作各不相同，彼此也没有联系，但他们都受同一个控制

流支配，决定他们的执行次序，称为过程内聚。



通过循环体，计算两种累积数。

5.时间内聚

p如果一个模块内的各组成部分的处理动作和时间有关，则称为时间内聚。

p时间内聚模块的处理动作必须在特定的时间内完成。

p例如：程序设计中的初始化模块。

6.逻辑内聚

p如果一个模块内部的各组成部分的处理动作在逻辑上相似，但功能都彼此不同或无关，则称为逻辑内聚。

p一个逻辑内聚模块往往包括若干个逻辑相似的动作，使用时可以选用一个或几个功能。

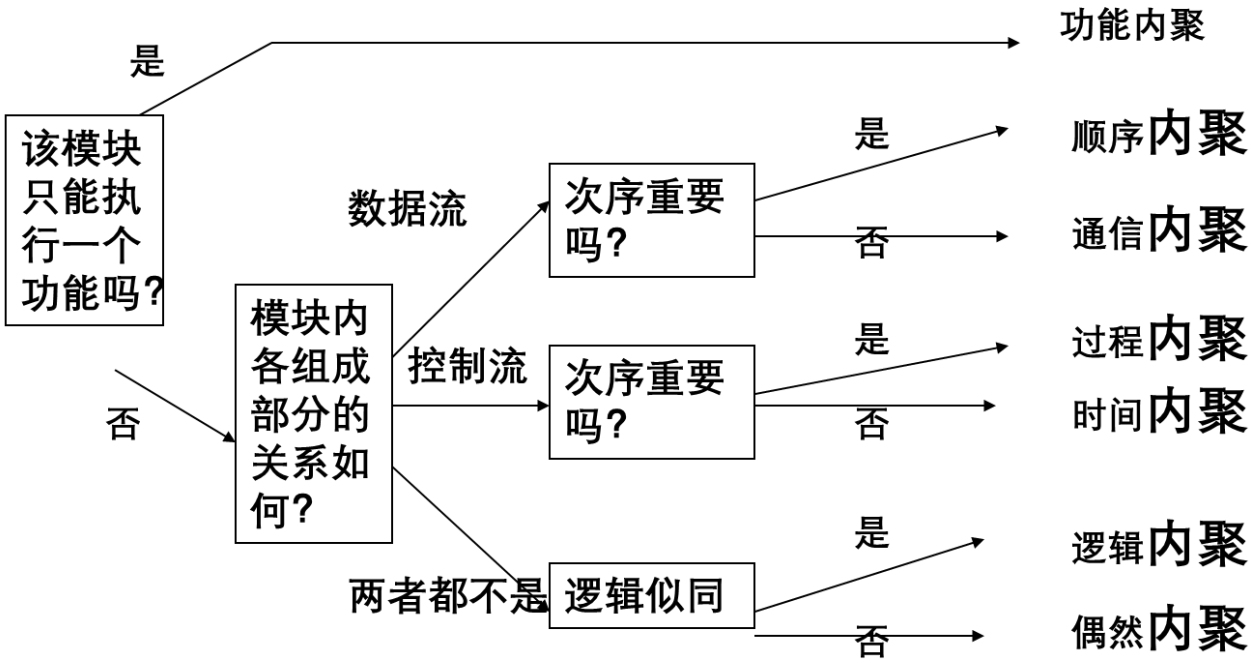
p例如：处理软盘、硬盘等外部设备的模块，在使用者看来，在逻辑上是相似的，但在实现上，却彼此无关，因此，把对这类设备的数据操作功能，放在一个模块中。

7.偶然内聚

p如果一个模块的内部各组成部分的处理动作彼此没有任何联系，则称为偶然内聚。

p例如：程序员发现，有些语句在很多地方出现，于是，把这些语句单独编成一个子程序，组成一个模块，这个模块就是偶然内聚

模块内聚度的判断



七种内聚模块的性能比较

聚合形式	聚合形式	可修改性	可读性	通用性	黑箱程度	聚合性
功能内聚	好	好	好	好	黑箱	10
顺序内聚	好	好	好	中	不完全黑	9
通信内聚	中	中	中	不好	不完全黑	7
过程内聚	中	中	中	不好	半透明	5
时间内聚	不好	不好	中	最坏	半透明	3
逻辑内聚	最坏	最坏	不好	最坏	透明	1
偶然内聚	最坏	最坏	最坏	最坏	透明	0

软件架构设计的模式与风格

建筑模式

- 每个模式是一个由三部分组成的规则，表达了特定环境、问题和解决方式(solution)之间的关系。
- 作为现实世界的一个成分，每个模式表达了下列三者之间的一种关系：特定环境，在该环境中反复出现的因素forces)的系统，以及协调这些因素的某种方案。
- 作为语言的一个成分，每个模式是一条指令，展示了这种方案如何被一再重复使用，目的是协调同特定环境相关的因素的系统。
- 简单地说，模式既是存在于现实世界中的事物，又是告诉我们如何以及何时创造该事物的规则。模式既是过程，又是事物；既是活生生的事物的描述，又是创造该事物的过程的描述。

软件构架风格概念

- 软件构架风格(Architectural style)又称为软件构架模式(Architectural pattern)
- 软件构架风格描述某一特定应用领域中系统组织方式的惯用模式，以结构组织模式定义了一个系统家族
    - 关于构件和连接件类型的术语词汇表
    - 一组约束它们组合方式的规定
    - 一个或多个语义模型，规定了如何从各成分的特性决定系统整体特性
- 概括地说，一种软件构架风格刻划一个具有共享结构和语义的系统家族

典型的构架风格

分类	典型风格
数据流风格 (Data flow)	批处理序列 (Batch sequential),管道和过滤器(Pipes and filters)
调用/返回风格 Call/return	主程序/子程序(Main program and subroutine),面向对象 (Object-oriented),层次结构(Layered)
独立构件风格(Independent components)	进程通信(Communicating processes),事件系统(Event systems)
虚拟机风格 (Virtual Machine)	解释器(Interpreter),基于规则系统(rule-based system)
仓库风格 (Repository)	数据库系统(Database system), 黑板系统(Blackboard system)

其他典型构架风格

分类	典型风格
通讯类 (Communication)	**Service-Oriented Architecture(SOA)**, Message Bus,
部署类 (Deployment)	**Client/Server**,3-Tier,N-tier,Peer-to-peer(P2P),Grid Computing, Cloud Computing
领域类 (Domain-Specific)	Search-oriented architecture,Web2.0
交互类 (Interaction)	Separated Presentation, **Model-Viwer-Controller(MVC)**
结构类 (Structure)	**Plugin**, Shared nothing architecture

两种典型软件系统的架构模式分析

ISO/OSI 模型的最大贡献，是对以下三个概念，作出了明确的区分和定义：

p服务：每一层都为它的上一层提供服务（原语操作），服务定义了这一层应该做什么，而不管上面的层如何访问它以及该层如何实现预定的服务工作
p接口：接口（原语）告诉自己的上层应该如何访问自己，定义了访问的参数和预期的结果，这也和该层如何实现无关
p协议：协议是端到端的对等协议（帧格式、信息定义），是该层的内部事物。因此，采用什么协议或协议的改变，是端与端之间的事，与上层（服务）无关

比较ISO/OSI与TCP/IP模型的体系结构

- ISO/OSI的优缺点：
  - 优点：完全符合结构化模式的设计要求
  - 缺点：由于模型产生在应用（协议发明）之前，很多功能不知道应该放在哪层更合适
  - 没有流行的四个糟糕：
    - 时机：研究-标准-投资的双象曲线，TCP/IP更符合双象曲线
    - 技术：以IBM的SNA七层为模型，有些部分过于复杂，有些则没有考虑到
    - 实现：由于过于复杂，实现的效果很差，丢失了市场
    - 策略：ISO/OSI走的是政府（标准）的路线，TCP/IP走的是UNIX的（一个部分）大众、免费路线

- TCP/IP的优缺点：
  - 优点：有大量应用
  - 缺点：由于没有很好地区分和定义服务、接口和协议的概念，因此，导致：
    - TCP/IP模型不是通用的网络模型，只适合自己，不适合描述其他网络，如：IBM SNA
    - 在网络层之下（如：ISO/OSI的链路和物理层），没有一个清晰定义的层，只在网络层和数据链路层之间，有IP一个接口
    - 由于没有数据链路层和物理层，所以，这二层在网络设计和实现中，做不到标准统一和规范一致。这是ISO/OSI做的最好的二层
    - TCP/IP协议中，只有TCP和IP协议定义的比较好，其他都非常简单粗糙

比较ISO/OSI与TCP/IP模型的体系结构

- 良好的体系结构，可以提供：
  - 广泛的灵活性：OSI的灵活性与TCP/IP的局限性
  - 清晰的结构定义：OSI功能层定义清楚、服务功能明确（是否合理是问题），TCP/IP功能与协议混在一起，层次概念模糊
  - 较低的系统复杂性：单独一层的复杂度降低
  - 较好的扩展性：OSI扩展容易，TCP/IP扩展困难
- 结构是发展的前提和基础
- 但是，市场的作用

体系结构的层次模型

p软件系统的体系结构包含的内容是：
p构成单元
p作用和关系
p继承方法和约束

p在讨论体系结构的时候，发现了一个重要的思想方法：分层

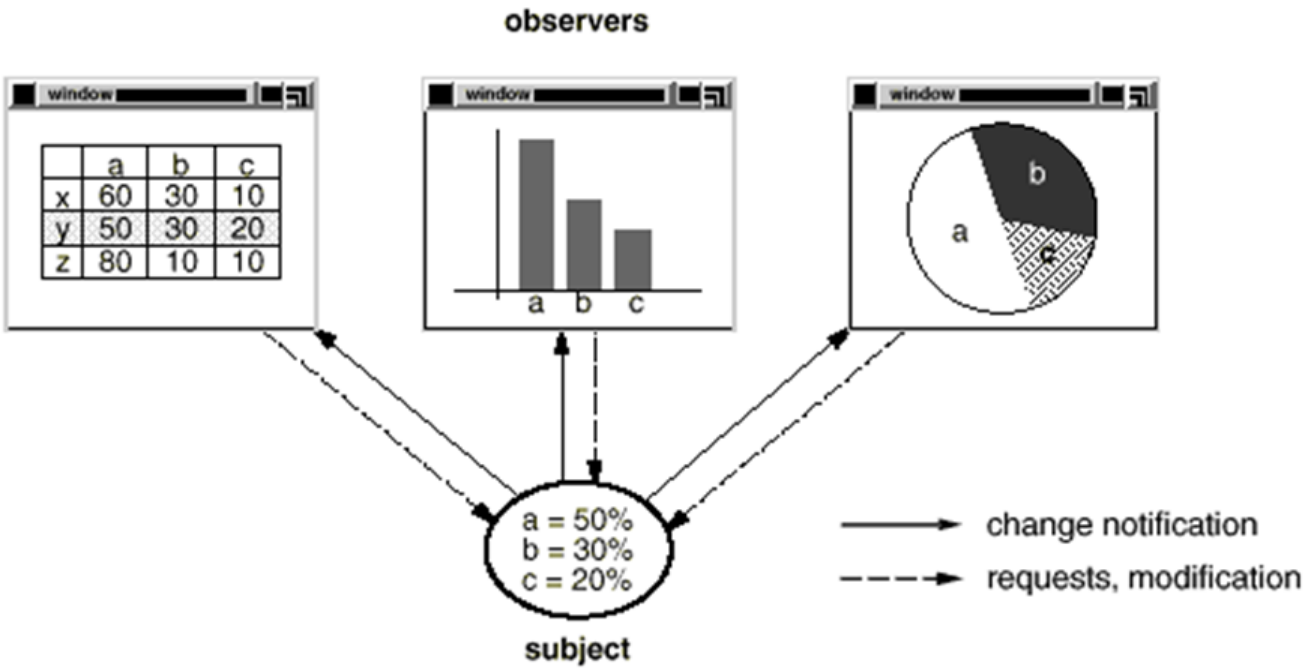
p层次结构——具有某种特定“趋向”性的结构
p按功能：从低到高、从物理到逻辑
p按抽象：从具体到一般、从稳定到多变
p从范围：从小到大、约束从紧变松

基于MVC设计模式的架构设计与实现

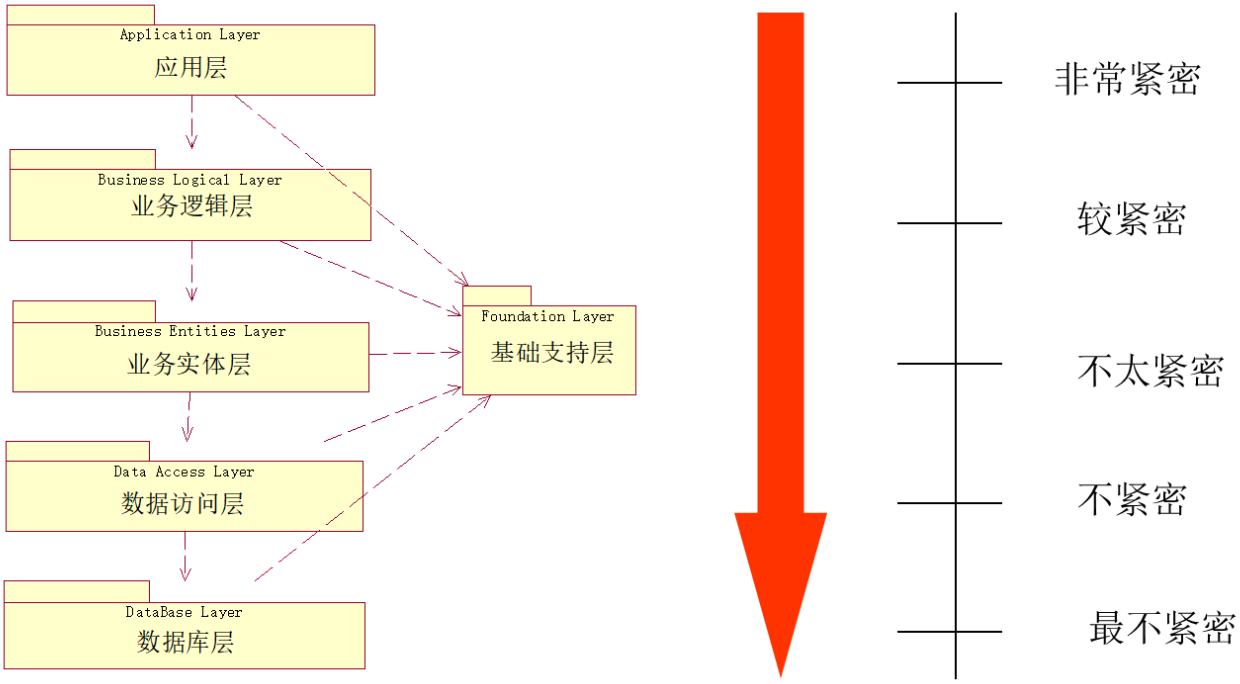
面向对象的MVC设计模式

图形化与交互式应用的可变需求

p基于图形显示的应用，是使用最广泛的应用。
p这个应用模型典型地反映了数据存储与表示的分离。
p用户界面是用户与系统交互操作的窗口，是用户对系统的直接感受和接触点
p在应用系统中，与应用逻辑相比较，用户界面的变化是系统需求中最容易发生改变的地方
p系统设计的一个重要任务，就是在不改变系统功能的前提下，如何更好地支持和适应用户不断变化的变更需求



系统架构与需求变化的关系



软件架构与实际业务模型联系的密切程度

界面设计的可变需求

问题（1）界面变化的需求与功能无关

系统功能是相对稳定的，但是：

ü同一个软件版本的不同类型用户，需要不同的用户界面
ü同一个系统的不同应用（例如：因权限不同、可打开的模块不同、可看见的数据不同），也可能需要构成不同的用户界面。
ü不同时期、不同用户的不同表现喜好，需要不同的界面

因此，如果把用户界面与实现功能紧密结合，则随着用户界面的不同或更改，系统的灵活性和需求变更的工作量将无比巨大。

问题（2）界面变化的需求与功能有关

ü同一种信息可以在不同的地方有不同的表示
ü应用程序的显示和动作必须立即反映出数据的变化
ü用户的接口易于改变，甚至在运行时刻也可以改变
ü支持不同的窗口系统，或者用户界面的基础软件

n对系统界面部分的要求：
具有灵活的人-机界面的交互方式
ü可以灵活选择不同的信息表示方式
ü可以灵活选择用户的操作方式
ü当然，这里灵活必须保证是在一定限度内，不可能是无限的

n困难：
必须满足界面构成与系统计算模型的独立

n模型视图控制器MVC（Model-View-Controller）就是一种交互界面的架构组织模式
模型视图控制器是在２０世纪８０年代，随着SmallTalk-80而流行起来的设计模式。它是一个十分有效的设计模式，广泛用于图形用户界面的设计中。

MVC强调把用户输入、数据模型和图像显示以模块的方式分开设计。

模型-视图-控制器（MVC）模式

•MVC模式将一个交互式应用程序分成3个部件
•模型（model）：软件所处理的核心逻辑，包含核心功能和数据
•视图（View）：向用户显示信息，对相同的信息可以有不同的显示
•控制器（Controller）：处理用户的输入（如：鼠标、键盘等），转化成用户对模型或视图的服务请求，并把信息的变化，传递给视图。用户仅通过控制器与系统交互

•一组视图和控制器组成了一个用户界面

•一个模型可以有多个视图界面，如果用户通过某个视图的控制器，改变了模型的数据，控制器会将这个变化，通知所有视图，导致显示的更新。

•这是典型的观察者（Observer）或称为：发布-订阅（publish-subscribe）、变更-传播模式。这种机制保证了模型和用户界面之间的一致性。

行为型模式7：观察者（Observer）

•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变化时，所有依赖于他的对象都得到通知并被自动更新。

•动机：
•把一个系统分解成一些相互协作的类或者对象，有一个问题，就是如何维护这些相关的类/对象的一致性?
•一致性是协作的需要，但太紧密，又导致紧偶合
•观察者模式的典型应用是MVC结构

•这个模式，也称为发布-订阅（publish-subscribe）模式。目标是通知的发布者，它发出通知时，并不知道谁是订阅者、有多少订阅者、是什么样的订阅者。

n发布-订阅（publish-subscribe）模式：
n发布：广而告之
n订阅：只针对订阅者
n发布主体：模型（Model）
n消息受众：
l广播读者（推）：可能没有任何请求、被动接收
l订阅读者（拉）：点击、主动请求
n实现：变更-传播机制

MVC的结构

•将应用程序分成三个部分：

> •模型组件：封装了内核数据和功能。模型独立于特定输出表示或者输入方式。
> •视图组件向用户显示信息。视图从模型获得数据。一个模型可能有多个视图。
> •每个视图有一个相关的控制器组件。控制器组件接受输入，通常将鼠标移动、CLICK等用户输入翻译成为对视图或者模型的服务器请求。用户仅仅通过控制器与系统交互。

类：模型
数据结构和关系 视图和控制器的注册关系
内部数据和逻辑计算 向视图和控制器通知数据变化

类：视图
显示形式 显示模式控制
从模型获得数据 视图更新操作

类：控制器
状态
事件处理 控制视图更新

MVC结构1——模型

•模型部件包含了应用程序的功能内核。
•封装了相应的数据
•封装了完成问题处理的过程，控制器代表用户调用这些过程。
•模型也提供访问它封装的数据的函数，视图为了获得显示数据，通过函数对这些数据进行访问操作

•变更-传播机制
•维护了一个注册表，各部件通过注册表建立相互依赖关系
•所有视图还有一些控制器在这个表中登记了对变更通知的需求。
•模型状态的改变将触发变更-传播机制。每个在表中登记的视图和控制器都会收到 变更通知。

MVC结构2——视图

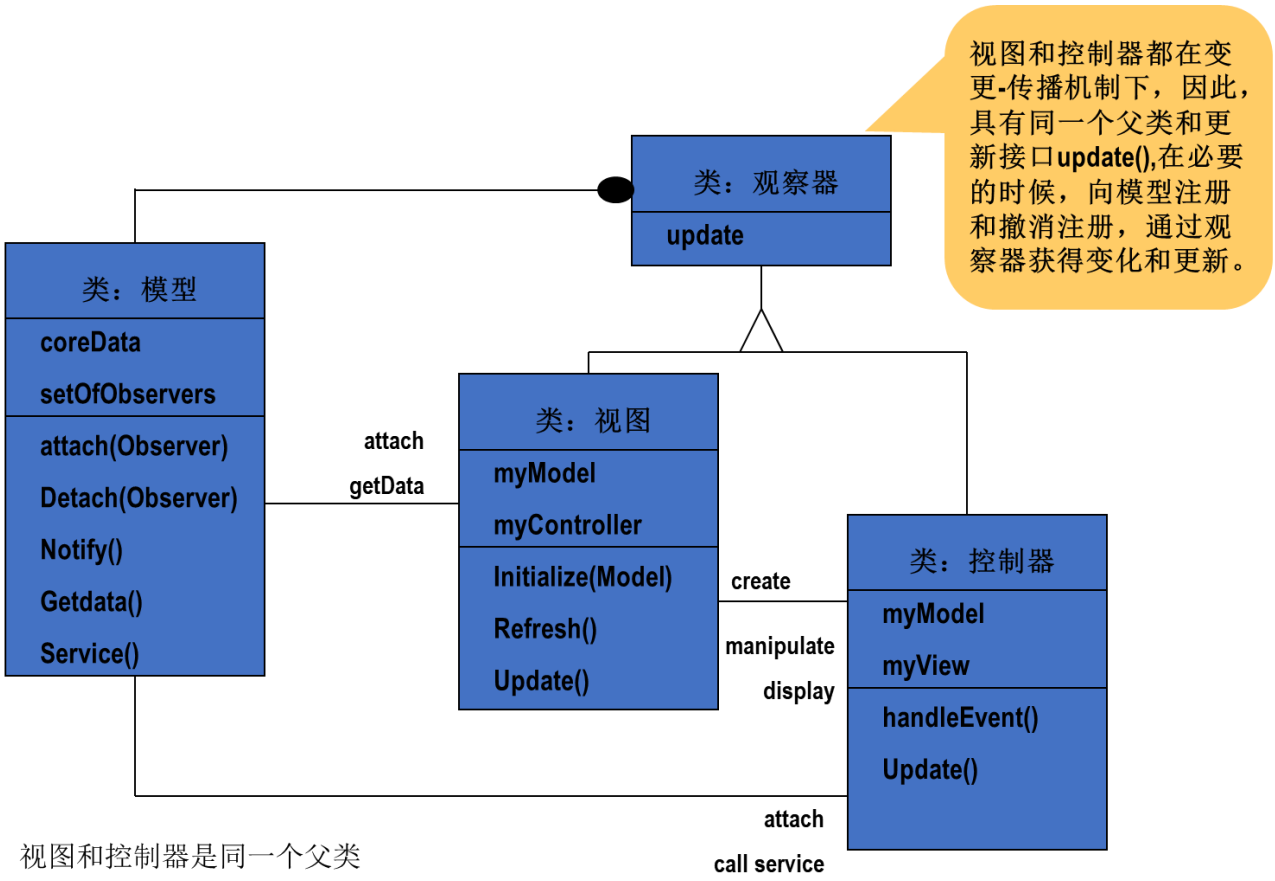
•视图部件向用户呈现信息
•不同的视图用不同的方法呈现信息。
•每个视图部件都有一个更新操作。这个操作被模型变更通知激活（视图被动地更新）。
•这个操作被激活（此时模型已经改变）后，视图得到新的模型数据，视图和模型重新一致。
•在初始化阶段，通过建立注册关系（订阅），视图向模型登记请求变更通知（视图主动地更新）。
•每个视图创建一个相应的控制器，视图给控制器可以处理显示的操作，因此，控制器具有可以主动激发更新界面的能力（控制权交给控制器）

MVC结构3——控制器

•控制器与视图
•控制器获得事件输入，如何获得事件与用户界面平台有关，例如：是命令行、还是鼠标点击等
•事件被翻译成为对模型或者视图的请求（视图对模型的操作激发）或模型对视图的更新（双向功能）

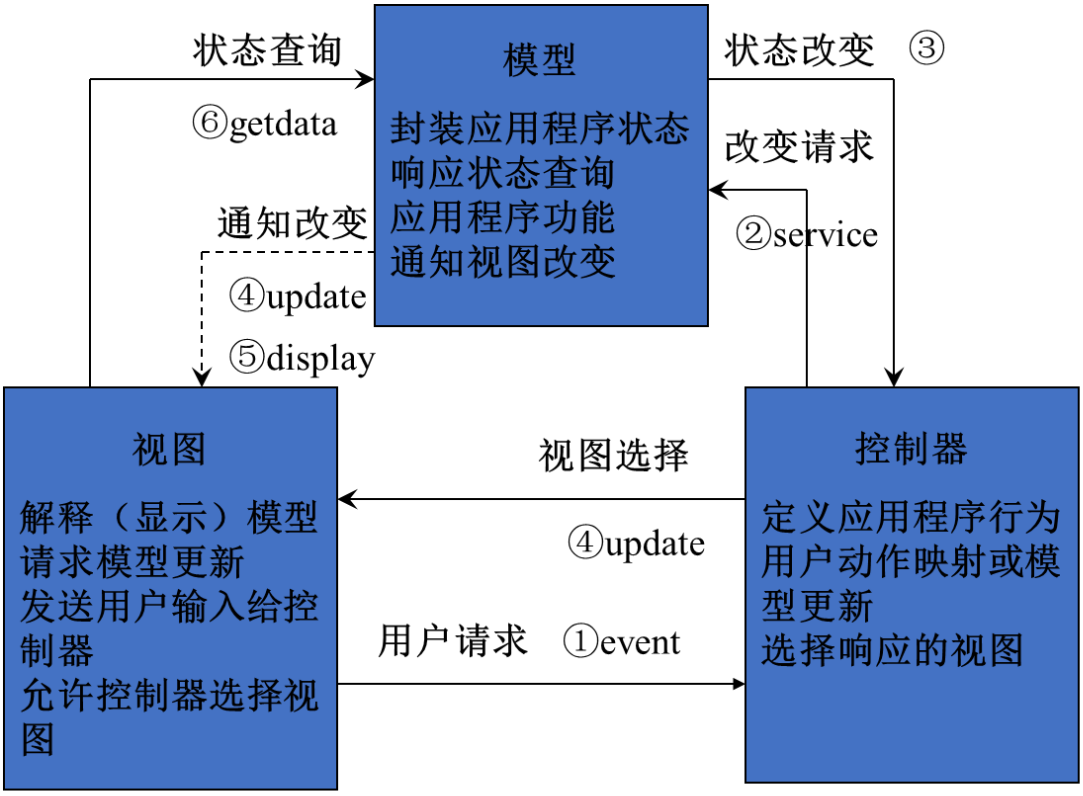
•控制器本身也是一个独立的功能部件
•如果控制器的行为依赖于模型的状态（例如：由模型的状态，决定控制器允许或不允许进行某类操作），那么控制器也需要向变更-传播机制注册，模型为控制器建立一个操作。
•当模型状态变化时，变更-传播机制像处理视图一样，为控制器提供一个更新操作，使控制器改变操作行为（例如：关闭或打开某种操作）

MVC的结构关系（面向对象的实现）



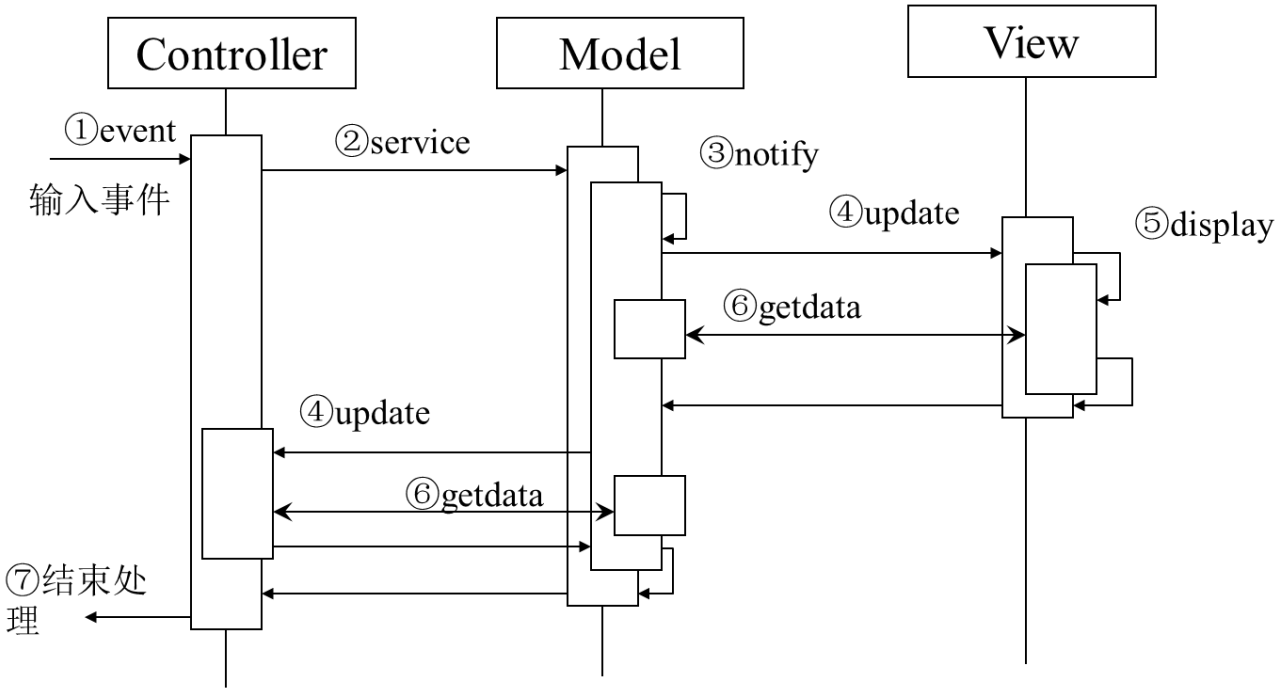
- 视图和控制器是同一个父类

MVC的动态行为关系



- 控制器只传“信令”，不“二传”数据

MVC的动态行为关系



用户输入导致模型变化，并触发变更-传递机制的过程

该情境为：用户输入导致模型变化，并触发变更-传递机制的过程。具体过程如下：
①控制器接收到事件，解释事件并且启动模型的服务过程。
②模型执行相应的过程，并导致内部状态的变化。
③模型调用其更新过程，向所有登记请求了变更-传播机制的视图和控制器发出通知。
④每个视图从模型中读取新数据并且重新显示。
⑤每个控制器从模型中读取新数据，修改自己的行为，比如禁用某个功能。
⑥最初的控制器恢复控制并从事件处理过程返回。

软件测试

软件测试过程

测试计划

- 确定测试范围、测试策略
- 规划测试任务日程表
- 安排测试资源
- 评估测试风险
- 制定风险控制策略

测试设计

制定测试方案
设计测试用例
开发自动化测试脚本

测试执行

搭建测试环境
配置测试数据
执行测试用例并记录测试结果
报告缺陷
分析、处理、跟踪缺陷`

```
测试总结
        分析测试结果
        编制测试报告
        提交测试文档
```

软件测试的特点

（1）软件测试的成本很大。
（2）不可进行“穷举”测试。
（3）测试具有“破坏性”
（4）软件测试是整个开发过程的一个独立阶段，并贯穿到开发各阶段。

软件测试基本原则

```
1.测试可以证明缺陷存在，但不能证明缺陷不存在
2.穷尽测试是不可能的
3.测试活动应尽早开始
4.缺陷集群性
5.杀虫剂悖论
6.测试与环境相关
7.没有失效/缺陷不代表系统是可用的

1.并非所有软件缺陷都要修复
2.什么时候才叫缺陷难以说清
3.程序员应当避免测试自己的程序
4.软件测试员在产品小组中不受欢迎（检查和批评、挑毛病、公布问题）
```

软件测试策略

软件测试策略的特征

软件测试策略是指软件测试的思路模式，也是采用特定测试用例技术和方法的重要依据。如遵循从单元测试到最终的功能性测试和系统性测试等。
软件测试策略，具体包含5个特征：
（1）测试从模块层开始，然后扩大延伸到整个系统。
（2）不同的测试技术适用于不同的时间点。
（3）对于大型系统测试，由软件的开发人员和独立的测试组进行管理。
（4）测试和调试是不同的活动，但是调试必须能够适应任何的测试策略。
（5）充分考虑以下特性，有利于测试策略更科学合理、优质高效。

软件测试策略的内容

软件测试的重点主要考虑软件在测试中，模块、功能、性能、接口、版本、配置和工具等方面及其各个因素的影响。因此，测试策略的主要内容包括：测试目的、测试用例、测试方法、测试通过标准和特殊考虑。

软件测试方法

1．黑盒测试

黑盒测试也称为功能测试或黑箱测试，其盒是指被测试的软件，“黑盒”则指测试人员只知道被测软件的界面和接口外部情况，不必考虑程序内部逻辑结构和特性，只根据程序的需求分析规格说明，检查其功能是否符合。
以黑盒测试发现程序中的错误，应在所有可能的输入条件和输出条件中确定测试数据，检查程序是否都能产生正确输出。
黑盒测试主要检测的错误/问题包括：功能不正确/被遗漏、界面错误、数据结构/外部数据库访问错误、性能错误、初始化/终止错误。
测试模块之间的接口，适合采用黑盒测试，适当辅以白盒测试，以便能对主要的控制路径进行测试。

常用的几种黑盒测试技术方法为：

- 等价类划分（ECP）法

  所谓等价类，是输入条件的一个子集合，该输入集合中的数据对于揭示程序中的错误是等价的。从每一个子集中选取少数具有代表性的数据，从而生成测试用例。
- 边界值分析法
- 错误推测法
- 因果图法
- 判定表驱动法
- 正交试验设计法
- 功能图法
- 场景图法

2．白盒测试

白盒测试主要是对程序内部结构执行路径的测试，也称透明盒测试、开放盒测试、结构化测试、基于代码测试和逻辑驱动测试等。测试人员将测试对象看作一个打开的盒子，搞清软件内部逻辑结构和执行路径后，利用其结构及有关信息设计测试用例，对程序所有逻辑路径进行测试，以检测不同点检查程序的实际状态与预期状态一致性。

1）白盒测试的原则
(1) 模块中每一个独立的路径至少执行一次。
(2) 所有判断的每一个分支至少执行一次。
(3) 每个循环都在边界条件和一般条件下至少执行一次.
(4) 所有内部数据结构的有效性。

2. 白盒测试技术
   (1) 逻辑覆盖测试。
   (2) 循环测试。
   (3) 基本路径测试。
3. 白盒测试的步骤及优缺点
   （1）白盒测试的步骤为：根据详细设计/源程序代码导出程序流图、计算环路复杂性、确定线性独立的基本路径集、设计测试用例。

（2）白盒测试的优点是：迫使测试人员去仔细思考软件的实现；可以检测代码中的每条分支和路径；揭示隐藏代码中的错误；对代码的测试较彻底。

（3）其缺点是：无法检测代码中遗漏的路径和数据敏感性错误，而且难以验证具体规格的正确性。

表04-2 黑白盒测试法优缺点及应用范围比较

项目	黑盒测试法	白盒测试法
规划方面	功能测试	结构测试
优点方面	能确保从用户的角度出发进行测试	能对程序内部的特定部位进行覆盖测试
缺点方面	无法测试程序内部特定部位；当规格说明有误，则不能发现问题	无法检查程序的外部特性； 无法对未实现规格说明的程序内部欠缺部分进行测试
应用范围	边界分析法 等价类划分法 决策表测试	语句覆盖，判定覆盖，条件覆盖，判定/条件覆盖，路径覆盖，循环覆盖，模块接口测试

3．灰盒测试

白盒和黑盒测试方法各有所侧重及特点不可替代。灰盒测试则是介于白盒测试和黑盒测试之间的测试。

4．易用性测试

易用性测试目的明确，标准不易确定。涉及的范围较广，如安装易用性、功能易用性、界面易用性，特别可以含有听力、视觉、运动及认知有缺陷的客户体现的易用性。

5．负载/压力测试

对于软件运行的最低配置或最低资源需求，可通过减少软件需要的资源（内存、存储空间、网络资源等）进行测试，而且，可正常提供软件需求的资源，并不断加载软件处理的任务，来测试软件在正常配置下的能力指标。

6．兼容性测试

兼容性测试主要检测不同软件之间或软件与硬件/数据之间的兼容性。如应用软件与操作系统、数据库、中间件、浏览器和其他支撑软件的兼容性，同一软件不同版本之间或对不同数据格式的兼容性等。

7．回归测试

回归测试是指软件修改之后，为保证其修改的正确性，重新使用原有测试用例执行的测试方法。

8．边界值测试

一些专门针对软件需要从外界（客户、接口程序）获取数据的地方，提供数据的边界值，验证程序是否对边界值进行正确或合理的处理。

9．α测试和β测试

> α测试由用户在开发者的场所进行，而且在开发者对用户的“指导”下进行测试。
> β测试由软件的最终用户在客户场所（如网络下载试用）进行，开发者通常不在测试现场。

10．基于Web的系统测试方法

- 1）功能测试
  - （1）链接测试。
  - （2）数据库测试。
  - （2）表单测试。
  - （3）设计语言测试。
- 2）性能测试
  - （1）连接速度测试。
  - （2）负载测试。
  - （3）压力测试。
- 3）可用性测试
  - （1）导航测试。
  - （2）图文测试。
  - （3）内容测试。
  - （4）整体界面测试。
- 4）客户端兼容性测试
  - （1）平台测试。
  - （2）浏览器测试。

5）安全性测试

Web应用系统的**安全性测试**，主要包括：

（1）对先注册后登陆方式，检测用户名和密码的有效性、使用次数及大小写的限制等。

（2）用户填写或提交信息时的超时限制。

（3）测试系统日志文件对相关信息存储和可追踪性。

（4）使用安全套接字时，检测加密正确性和信息完整性。

（5）测试无授权时，服务器端脚本放置和编辑问题，以防安全漏洞。

软件测试理念

明确的目标

人们通常是目标驱动的，人们通常根据管理者或利益相关方确定的目标制定计划，比如发现缺陷或证明软件系统能正常工作。因此，一定要明确测试的目标。

独立的测试

开发人员很难发现自己程序中的问题，最好采用独立测试。独立的程度可有以下几种：
同事互测
专门的测试团队
专门的测试机构（第三方测试）

软件质量与质量管理概述

软件质量

ISO软件质量定义：对用户再功能和性能方面需求的满足、对规定标准和规范的遵循以及正规软件某些公认的应该具有的本质。

ANSI/IEEE定义：与软件产品满足规定的和隐含的需求能力的特征或特性的全体。

质量管理

确定质量方针、目标和职责并在质量体系中通过质量计划、质量控制、质量保证和质量改进使其实施的全部管理职能的所有活动。
质量管理涉及到一切质量因素，也涉及到质量管理的手段与方针，是有计划的系统活动。实际上，质量管理主要就是监控项目的可交付产品和项目执行过程，以确保它们符合相关的要求和标准，同时确保不合格项能够按照正确方法或者预先规定的方式处理。对于软件项目，良好的项目管理过程是取得令人满意的项目成果、项目产品或者服务的保证。

规划质量——定规则

软件质量管理主要过程：

（1）质量计划

软件质量计划过程是确定项目应该达到的质量标准，以及决定如何满足质量标准的计划安排和方法。

（2）质量保证

为了提供信用，证明项目将会达到有关质量标准而开展的有计划、有组织的工作活动。

是贯穿整个项目生命周期的系统性活动，经常性的对整个项目质量计划的执行情况进行评估、检查与改进工作，向管理者、顾客或者其他相关人员提供信任，确保项目质量与计划保持一致。

（3）质量控制

质量控制是确定项目结果与质量标准是否相符，同时确定消除不符的原因和方法，控制产品质量，及时纠正缺陷的过程。质量控制是对阶段性成果进行检测、验证，为质量保证提供参考依据。

质量管理是组织的生命线
QA：Quality Assurance, 质量保证
重点关注过程改进
QC：Quality Control , 质量控制
重点关注对产品和成果的检验和测试
软件质量管理不能只关注软件质量控制（软件测试）；国内很多公司，对软件质量管理的定义就是软件测试，软件测试只是软件质量管理的一部分，更重要的是控制质量-查结果。通过控制质量-查结果，使质量管理真正与软件设计开发融为一体。

管理质量——管过程

软件质量管理体系：CMM

过程能力等级	特点	关键过程领域
I级（初始级）	软件开发过程是特定的，只有很少的工作过程是经过严格定义的，软件过程经常被改变，软件质量不稳定，进度、费用等难以预测。	
II级（可重复级）	建立了基本的项目管理过程，可进行软件开发以及跟踪成本、进度和性能等方面所必须的过程管理。能够提供可重复以前成功项目管理的经验和环境，软件需求、软件开发过程及其相应的技术状态是受控的。	需求管理 软件项目 软件项目跟踪和监督 软件分包合同管理 控制质量-查结果 软件配置管理
III级（已确定级）	软件开发活动的过程在管理活动、技术活动和支持活动等方面都已经文档化、规范化。所有项目或产品的开发和维护都在这规范化的体系基础上进行定制。软件项目的成本、进度、质量以及过程是受控的，软件质量具有可追溯性。	组织过程焦点 组织过程定义 培训大纲 综合软件管理 软件产品工程 组织协调 同行专家评审
IV级（已管理级）	运用度量方法和数据，可以对软件产品和开发过程实施定量的分解和控制	定量的过程管理 软件质量管理
V级（优化级）	通过建立开发过程的定量反馈机制，不断产生新的思想、采用新的技术来不断地改进和优化软件开发过程。	缺陷预防 技术改变管理 过程改变管理

风险管理概要

基本概念

主观：风险是损失的不确定性
客观：是给定情况下一定时期可能发生的各种结果间的差异

基本特征：不确定性、损失

规划风险应对

次生风险

实施风险应对措施而直接导致的风险。
规划风险应对时，需要识别次生风险。往往需要为风险分配时间或成本应急储备，并可能需要说明动用应急储备的条件。

威胁应对策略——规避

- 定义：指项目团队采取行动来消除威胁，或保护项目免受威胁的影响（将发生概率降低到零）
- 举例：延长进度、改变策略、缩小范围、澄清需求、获取信息、改善沟通、取得专有技能
- 注意点：适用于发生概率较高，且具有严重负面影响的高优先级威胁

威胁应对策略——转移

•定义：将应对威胁的责任转移给第三方，让第三方管理风险并承担威胁发生的影响。
•举例：保险、使用履约保函、担保、保证书、外包
•注意点：通常需要支付风险转移费用。转移风险是把风险管理责任简单地推给另一方，而并非消除风险。

威胁应对策略——减轻

- 定义：采取措施降低威胁发生的概率和（或）影响
- 举例：采用较简单的流程、进行更多测试、选用更可靠的卖方、原型开发、加入冗余部件
- 注意点：它意味着把不利风险的概率和/或影响降低到可接受的临界值范围内

威胁应对策略——接受

- 定义：承认威胁的存在，但不主动采取措施
- 举例：主动接受 - 建立应急储备；被动接受 - 记录策略，无需任何其他行动，需要定期复查。
- 注意点：适用于低优先级危险，或无法以任何其他方式加以经济有效地应对的威胁。