**C:\Users\sse\Desktop\logo_01.png**

**软件体系结构**

**知识点总结**

|  |  |
| --- | --- |
| **姓名：** | **陈俊哲** |
| **学号：** | **19301141** |

目录

[软件体系结构知识点总结 3](#_Toc106088334)

[1. 软件体系结构简介 3](#_Toc106088335)

[1.1. 软件开发的历史 3](#_Toc106088336)

[1.2. 软件架构的定义 3](#_Toc106088337)

[1.3. 软件结构的相关概念 3](#_Toc106088338)

[1.4. 软件架构的影响因素 4](#_Toc106088339)

[1.5. 软件结构的好处 4](#_Toc106088340)

[1.6. 软件架构的当前研究和实践现状 5](#_Toc106088341)

[2. 软件体系结构建模 5](#_Toc106088342)

[2.1. 软件体系结构视图模型 5](#_Toc106088343)

[2.2. 4+1 视图模型 5](#_Toc106088344)

[2.3. 统一建模语言 UML 6](#_Toc106088345)

[2.4. Rational’s 4+1模型 8](#_Toc106088346)

[3. 软件体系结构质量属性 8](#_Toc106088347)

[3.1. 质量属性介绍 8](#_Toc106088348)

[3.2. 质量属性脚本 8](#_Toc106088349)

[3.3. 常见质量属性 9](#_Toc106088350)

[4. 设计原则 10](#_Toc106088351)

[4.1. 失败的设计特征 10](#_Toc106088352)

[4.2. 单一责任原则Single-Responsibility Principle (SRP) 10](#_Toc106088353)

[4.3. 开闭原则Open-Closed Principle (OCP) 11](#_Toc106088354)

[4.4. 里式替换原则 Liskov Substitution Principle (LSP) 11](#_Toc106088355)

[4.5. 依赖反转原则 Dependency-Inversion Principle (DIP) 12](#_Toc106088356)

[4.6. 接口分离原则 Interface-Segregation Principle (ISP) 12](#_Toc106088357)

[4.7. 打包原则 13](#_Toc106088358)

[5. 体系结构风格 15](#_Toc106088359)

[5.1. 体系结构风格介绍 15](#_Toc106088360)

[5.2. 管道过滤器风格 Pipe-Filter Style 15](#_Toc106088361)

[5.3. 存储器风格 Repository Model Style 16](#_Toc106088362)

[5.4. 客户端服务器风格Client-Server Style 16](#_Toc106088363)

[5.5. 模型视图控制器风格Model-View-Controller(MVC) Style 17](#_Toc106088364)

[5.6. 层次系统风格Layered System Style 17](#_Toc106088365)

[5.7. P2P风格Peer-to-Peer Style 18](#_Toc106088366)

[5.8. 事件驱动风格Event-Driven Style 18](#_Toc106088367)

[5.9. SOA风格Service-Oriented Architecture Style 19](#_Toc106088368)

[6. 设计模式 19](#_Toc106088369)

[6.1. 设计模式介绍 19](#_Toc106088370)

[6.2. 创建型模式Creational Patterns 19](#_Toc106088371)

[6.3. 结构型模式Structural Patterns 22](#_Toc106088372)

[6.4. 行为型模式Behavioral Patterns 25](#_Toc106088373)

[7. 软件产品线 29](#_Toc106088374)

[7.1. 软件产品线介绍 29](#_Toc106088375)

[7.2. 软件产品线体系结构 29](#_Toc106088376)

# 软件体系结构知识点总结

## 软件体系结构简介

### 软件开发的历史

总的来说，软件的发展历史大致可以分为四个阶段：

1、在20世纪七十年代之前，通常以汇编语言为主，方案规模较小。

2、接下来的十年中，不断地出现各种高级语言，如C语言等，提出以结构为导向的理论，以及数据流控制流设计方法。

3、在八十年代到1995年前，各种应用开发框架涌现，如J2EE等，组件技术也应运而生，同时也提出了对象模型和设计的标准化。

4、95年后，出现了多种应用开发框架，底层功能已经搭建完成，复杂的软件也可以轻松开发。

### 软件架构的定义

一个程序或计算机系统的软件架构是系统的结构，包括软件元素、这些元素的外部可见属性以及它们之间的关系。

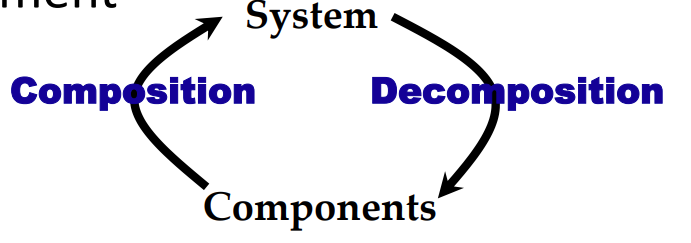
软件体系结构的组成：

* 元素elements：函数、接口、程序、类模块、层、子系统、客户端/服务端。
* 属性properties：提供服务、性能特征、纠错处理、共享资源的使用。
* 关系relationship：元素的组合机制和风格样式。

### 软件结构的相关概念

1、组件：在系统内逻辑上和功能上相互结合来实现某些需求，在不同的场景下有不同的目标，可以被继续划分为更小模块的单元称为组件。简单来说就是在编写程序时需要我们填充的内容，如：包、函数、类等。

2、组件之间的关系：是一个系统内组件之间的关系，如包和类的包含关系、类与类之间的调用关系。

3、组件的属性：主要体现在组件的非功能性需求，如内存利用率、可靠性、安全性、兼容性、可扩展性等。

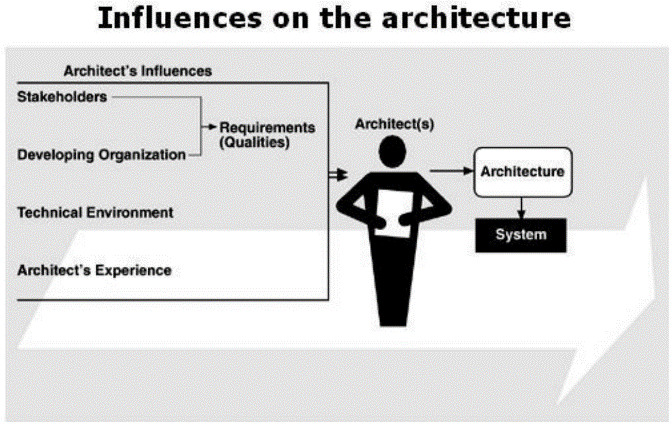
### 软件架构的影响因素

1、利益相关者：包括最终用户、客户、项目经理、系统工程师、开发人员、架构师、维护人员、其他开发人员等。不同的利益相关者有不同的视图、不同的蓝图。

2、开发公司：也可以说是开发公司里的职员，每个职员也是利益相关者。所以可以说开发公司也是特殊的利益相关者。

3、设计师的背景和经验：好的架构设计对于软件来说有至关重要的作用。架构师也属于特殊的利益相关者。

4、技术环境：不同的软件设计适合不同的语言，选择适合的语言非常重要。

影响因素会影响结构，反过来结构也会作用于影响因素。

### 软件结构的好处

1、技术方面的意义：满足系统需求和目标；指定实现和测试的约束；使得系统更加灵活的进行模块划分；降低维护和升级成本；提升系统的重用性，减少软件后期维护和后期软件改进的开支。

2、组织方面的意义：为软件利益相关者提供相应信息，例如成本与风险评估，工作分配与项目进度等；便于利益相关者之间的沟通。

### 软件架构的当前研究和实践现状

形式化研究：如何用具体的规则来描述软件架构。

验证和评估的研究。

最新的研究来自于ICSE：软件工程会议。

## 软件体系结构建模

### 软件体系结构视图模型

什么是模型：（1）模型是一种描述或类比，有助于将无法直接观察到的事物形象化。（2）模型是对现实的简化。（3）模型提供了系统的蓝图。（4）模型可以是结构化的，强调系统的组织，也可以是行为的，强调系统的动态性。模型的含义很广泛，在已有模型的基础上简化的模型也是模型。

模型的作用：（1）模型有助于我们将一个系统的现状或我们想要的样子可视化。（2）模型为我们提供了一个模板，指导我们构建一个系统。（3）模型允许我们指定一个系统的结构或行为。（3）模型可以把我们的决定记录下来。

体系结构视图模型：软件设计的模型，使用工程制图的形式。

### 4+1 视图模型

4+1视图模型包括4个属于软件设计模型的逻辑视图、开发视图、过程视图、物理视图，以及1个属于需求分析模型的场景（脚本）视图。

逻辑视图：最重要的视图，受到用户关注，主要描述系统的模块与模块之间的关系。其来源是功能性需求。

开发视图：受到程序员和软件管理人员关注，主要描述软件开发时的任务划分，系统模块的分组。其来源是可扩展性、可维护性等非功能性需求。

过程视图：受到集成工程师关注，主要描述系统模块之间的通信同步等。其来源是性能、并发度等非功能性需求。

物理视图：受到系统工程师关注，主要描述模块应该怎样部署在硬件之上。其来源是非功能性需求。

场景视图：描述软件系统功能。其来源是功能性需求。

### 统一建模语言 UML

一、UML简介

统一建模语言(Unified Modeling Language，UML)是一种为面向对象系统的产品进行说明、可视化和编制文档的一种标准语言，是非专利的第三代建模和规约语言。UML使用面向对象设计的建模工具，但独立于任何具体程序设计语言。

二、组成

UML由模型元素(Model Element)、图(Diagram)、视图(View)和通用机制(General Mechanism)等几个部分组成。

1、模型元素：代表面向对象中的类、对象、消息和关系等概念，是构成图的最基本的常面向对象系统分析与设计方法用概念。

2、图：是模型元素集的图形表示，通常是由弧（关系）和顶点（其他模型元素）相互连接构成的。

3、视图：是表达系统的某一方面的特征的UMI，建模元素的子集，由多个图构成，是在某一个抽象层上，对系统的抽象表示。

4、通用机制：用于表示其他信息，比如注释、模型元素的语义等。另外，UMI。还提供扩展机制，使UML语言能够适应一个特殊的方法（或过程），或扩充至一个组织或用户。

三、UML各种图

经典的13种UML图，只讲9种：静态图5个，动态图4个，静态图：用例图、类图、对象图、组件图；动态图：时序图、阶段图、活动图

1、用例图（user case diagram）

描述角色以及角色与用例之间的连接关系。说明是谁要使用系统，以及他们使用该系统可以做些什么。一个用例图包含了多个模型元素，如系统、参与者和用例，并且显示了这些元素之间的各种关系，如泛化、关联和依赖。

2、类图（class diagram）

类图是描述系统中的类，以及各个类之间的关系的静态视图。能够让我们在正确编写代码以前对系统有一个全面的认识。类图是一种模型类型，确切地说，是一种静态模型类型。类图表示类、接口和它们之间的协作关系。

3、对象图（object diagram）

与类图极为相似，它是类图的实例，对象图显示类的多个对象实例，而不是实际的类。它描述的不是类之间的关系，而是对象之间的关系。

4、组件图(component diagram)

描述代码构件的物理结构以及各种构建之间的依赖关系。

5、部署图(deployment diagram)

用来描述软件和硬件如何交互，软件布设到多少个服务器上。

6、时序图(sequence diagram)

序列图是用来显示参与者如何以一系列顺序的步骤与系统的对象交互的模型。最上面的方框：对象，而不是类。矩形的长条：对象的生命周期。中间的连线：对象之间发送的消息，通信。

7、协作图（collaboration diagram）

和序列图相似，显示对象间的动态合作关系。可以看成是类图和顺序图的交集，协作图建模对象或者角色，以及它们彼此之间是如何通信的。

8、状态图（statechart diagram）

描述类的对象所有可能的状态，以及事件发生时状态的转移条件，可以捕获对象、子系统和系统的生命周期。

9、活动图（activity diagram）

系统的输入、中间的活动到输出，整个流程的图。

四、UML图与“4+1”模型的关系

UML的各种图有一些可以作为“4+1”模型的某个模型，另一些可以作为“4+1”模型的另一个模型。一个“4+1”模型可以用多种图实现，但是并非多种图都要使用，具有一定的选择性。

用例图用于4+1的脚本模型；类图用于4+1的逻辑模型；4+1中的物理模型一般就使用部署图；4+1中的过程模型一般使用时序图；包图一般用于开发模型。

### Rational’s 4+1模型

实际上就是换了个名字：设计模型-逻辑模型；实现模型-开发模型；交互模型-过程模型；部署模型-物理模型；用例视图-脚本模型。

## 软件体系结构质量属性

### 质量属性介绍

质量属性（非功能性需求）主要包括：Performance 性能；Security 安全性；Availability 有效性；Scalability 延展性（可扩展性）；Usability 可使用性；Reliability 可依赖性；Portability 可移植性；Modifiability 可修改性；Maintainability 维护性。

质量属性是影响软件架构的一个重要因素，需求中的非功能性需求主要指的就是质量属性。所以在系统的设计、实施和部署过程中必须考虑实现质量属性。

### 质量属性脚本

质量属性场景就是对非功能性需求的描述，它描述了在某一场景下软件应该做的事情。质量属性场景由以下六个部分组成：

刺激源：生成该刺激的实体（人、计算机系统或其他激励器）；

刺激：刺激到达系统时可能产生的影响（即需要考虑和关注的情况）；

环境：该刺激在某条件内发生。如系统可能正处于过载情况；

制品：系统中受刺激的部分（某个制品被刺激）；

响应：刺激到达后所采取的行动；

响应度量：当响应发生时，应能够以某种方式对应其度量，用于对是否满足需求的测试。

需要将一般的质量属性场景（一般场景）与具体的质量属性场景（具体场景）区别开来，前者是指独立于具体系统、适合于任何系统的一般性场景；而后者是指适合于正在考虑的某个特定系统的场景，具体场景通常是指从一般场景中抽取特定的、面向具体系统的内容。

### 常见质量属性

1、性能performance：

吞吐量throughput：单位时间处理的任务量

响应时间：处理时等待的时间

死线deadline：必须在规定的时间内完成

2、可扩展性（延展性）scalability：

请求的负载量request load：系统可以支持多少个请求

系统连接数connections：有多少个用户可以同时连接，和socket相关

连接量≠请求量，连接上以后可能不请求，因此请求量＜=连接量

数据大小data size：处理的数据大小

部署deployment：系统可以同时部署在几台机器上使用

3、可修改性modifiability：

度量更改应用程序以满足新的(非)功能需求的容易程度。

4.、安全性security

受到黑客、病毒攻击后，系统是否能很好的应对。

策略：authentication验证：登录的用户名密码；authorization授权：不同用户有不同的角色等级；encryption加密：系统信息传输（内部或者交互）的过程中加密。

6.、有效性availability

包含两个方面。可依赖性reliability：保证运行一段规定时间，不出问题；可恢复性recoverability：如果出了问题，需要多长时间可以恢复。恢复的时间越短，有效性越好。

7、可用性usability：

关注的用户完成所需任务的难易程度以及提供的用户支持类型。

## 设计原则

### 失败的设计特征

组件之间的依赖特别强：耦合高；组件特别大，包含功能特别多：聚合低。

耦合coupling：组件两两之间的依赖关系。

聚合cohesion：组件本身功能数量就是聚合度，数量多，越发散，就是聚合低。功能少，关系密切，越接近，就是聚合高。

1.僵硬性（Rigidity）：难以更新维护

评判：更新一个功能时，所需要修改的模块越多，僵硬性越高。

导致原因：高耦合和低内聚（high coupling and low cohesion）

2.脆弱性（Fragility）：变动时导致崩溃

评判：当变更后导致与该变更处在逻辑上无关的模块无法使用，那么就是脆弱的。

导致原因：高耦合和低内聚（high coupling and low cohesion）

3.不可移动性（Immobility）：难以被拆分、复用

导致原因：没有做复用设计，高耦合和低内聚（high coupling and low cohesion）

4.粘性（Viscosity）：随着系统的更新，僵硬性、脆弱性、不可移动性逐渐增加

设计粘性、环境粘性

5.不必要的复杂性（Needless Complexity）：包含大量非必要或不会带来收益的模块

导致原因：往往是为了应对频繁变化的需求而提前设计的模块

6.不必要的重复（Needless Repetition）：包含重复的结构，但却没有进行统一抽象复用

7.不透明性（Opacity）：设计自己能看懂，别人看不懂，比如组件命名有问题。

### 单一责任原则Single-Responsibility Principle (SRP)

1、单一责任原则：一个组件的功能越少越好，每个组件只有一个责任。修改的影响范围变小，修改起来更加容易一些。每个类都只有一个函数，会使系统非常庞大而臃肿，会造成不必要的复杂性。要设计符合到什么程度，还要具体项目分析。

2、为什么要遵循单一责任原则：

• 提高类的可维护性和可读性

• 提高系统的可维护性

• 降低变更带来的风险

3、怎样遵循单一责任原则：

合理的职责分解。即把相同的职责放到一起，不同的职责和实现则放到不同的接口和类中。

### 开闭原则Open-Closed Principle (OCP)

1、开闭原则：对扩展开放，对修改关闭。

2、好处：

•开闭原则有利于进行单元测试。

•开闭原则可以提高组件复用性。

•开闭原则可以提高组件维护性。

•开闭原则是面向对象开发的要求。

3、保证开闭原则的方法：

使用继承和多态：需要调用父类和子类的类要直接关联父类，而不是关联子类。被调用的是父类而不是子类。

使用接口：直接在两个类之间增加接口。（其实是符合依赖反转原则。而依赖反转原则其实就是开闭原则的子原则。）

### 里式替换原则 Liskov Substitution Principle (LSP)

1、里式替换原则：父类的对象应该能被子类的对象替代而不产生问题（看父类的对象其实是子类对象）。

2、优点： 子类拥有父类的所有方法和属性，从而可以减少创建类的工作量。提高了代码的重用性。提高了代码的扩展性，子类不但拥有了父类的所有功能，还可以添加自己的功能。

3、如何遵守Liskov替换原则：

尽量从抽象类继承，而不从具体类继承。

如果有两个具体类A和B有继承关系，那么一个最简单的修改方案应当是建立一个抽象类C，让类A和B成为抽象类C的子类。

### 依赖反转原则 Dependency-Inversion Principle (DIP)

1、依赖反转原则：一个类A依赖于另一个类B的时候，可以在A和B之间加入一个B的接口，让A依赖于这个接口，B实现这个接口。这样可以降低A和B的耦合性，所以依赖反转原则可以看作是特殊的开闭原则。

2、依赖反转原则的主要作用如下：

依赖倒置原则可以降低类间的耦合性。

依赖倒置原则可以提高系统的稳定性。

依赖倒置原则可以减少并行开发引起的风险。

依赖倒置原则可以提高代码的可读性和可维护性。

3、实现方法：

每个类尽量提供接口或抽象类，或者两者都具备。

变量的声明类型尽量是接口或者是抽象类。

任何类都不应该从具体类派生。

使用继承时尽量遵循里氏替换原则。

### 接口分离原则 Interface-Segregation Principle (ISP)

1、接口分离原则：单一责任原则的子原则。如果一个接口对应好几个类，那么应该把这个接口分割为多个，每一个对应一个类。该原则可以保证接口的聚合性，所以可以看作是特殊的单一责任原则。

2、优点：

（1）将臃肿庞大的接口分解为多个粒度小的接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。（2）接口隔离提高了系统的内聚性，减少了对外交互，降低了系统的耦合性。（3）如果接口的粒度大小定义合理，能够保证系统的稳定性；但是，如果定义过小，则会造成接口数量过多，使设计复杂化；如果定义太大，灵活性降低，无法提供定制服务，给整体项目带来无法预料的风险。（4）使用多个专门的接口还能够体现对象的层次，因为可以通过接口的继承，实现对总接口的定义。（5）能减少项目工程中的代码冗余。过大的大接口里面通常放置许多不用的方法，当实现这个接口的时候，被迫设计冗余的代码。

3、实现方法：

（1）接口一定要适度。接口过小，则会造成接口数量过多，使设计复杂化。（2）为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。（3）提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。

### 打包原则

包聚合原则：关注的是类应该放在哪个包里。

#### 包聚合原则

##### 复用-发布等价原则 Reuse-release equivalence principle

可复用的类：功能有一般性，可以被复用。

不可复用的类：与业务相关，难以复用。

复用-发布等价原则：把可复用的类放到一组包中，把不可复用的类放到另一组包中。

##### 共同复用原则 Common-reuse principle

共同复用原则：把一起复用的类放到同一个包中。

##### 共同闭包原则 Common-closure principle

共同闭包原则：把将来会一起更新的类放到同一个包中。

一种情况下会被同时修改的一些类，就先把他们放在同一个包。另一种情况下被同时修改的类放到另一个包里。

前两条原则都是从复用角度进行分组的，第三条原则从维护角度出发，但实际上三者是相通的，即将依赖关系强的类分到一个包里，以此提高包的内聚降低包的耦合，便于后期的复用和维护等管理工作。

如果需要，就取交集。

#### 包耦合原则

包耦合原则：根据包之间的关系，对所打包的类进行调整。

##### 无环依赖原则 Acyclic-dependencies principle

无环依赖原则：这样构建一个图，让每一个包对应一个图的节点，一个包依赖于另一个包就给相应的节点之间连一条有向边。因此，如果这个图有环，那么修改环上的任意一个包，都会影响到其它包，说明包之间的依赖关系比较高，需要想办法去掉图中的环。

两个解决方法：

包级别的依赖反转原则，反转环的一条边。在环上增加一个包，让与这个包相关的两个包都依赖于这个包。在打包的时候减少包的耦合度，保证包的耦合度低，以减少早晚综合征。

##### 稳定依赖原则 Stable-dependencies principle

稳定依赖原则：如果一个包被多个包依赖，并且它自身依赖于很少的包，那么它就是稳定的，不容易受到其他包影响；反之，它就是不稳定的，容易受到其他包影响。因此，如果一个包依赖于另一个包，那么前者应该相对不稳定，后者应该相对稳定，这样它们之间才不容易相互影响，也就是依赖关系才比较低。

依赖应该是从不稳定的包到稳定的包。被多个包依赖（不经常修改），且不依赖于其他的包（不受其他包修改的影响），就是稳定的。

##### 稳定抽象原则 Stable-abstractions principle

稳定抽象原则：如果一个包包含多个抽象类，它的抽象度就高；反之，它的抽象度就不高。如果一个包依赖于另一个包，并且依赖关系源自包里面类的继承关系，那么前者应该抽象度低，后者应该抽象度高，而根据稳定依赖原则，前者应该相对不稳定，后者应该相对稳定。因此，抽象度低的包应该是不稳定的，抽象度高的包应该是稳定的。

抽象包：跟包里抽象类的数量有关。如果数量多，这个包就是抽象的，如果数量少，这个包就是不抽象的。

## 体系结构风格

### 体系结构风格介绍

软件体系结构风格：是描述某一特定应用领域中系统组织方式的惯用模式。它反映了领域中众多系统所共有的结构和语义特性，并指导如何将各个模块和子系统有效地组织成一个完整的系统。也就是说，软件体系结构风格定义了用于描述系统的术语表和一组指导构建系统的规则。

优点：（1）可重用性（核心）：设计重用、代码重用、文档重用。（2）提高开发效率和生产率（3）为其它新的设计理念提供起点（4）有助于权衡和预先评估设计（5）降低设计风险（6）促进设计师之间的沟通。

### 管道过滤器风格 Pipe-Filter Style

组件就是过滤器，组件的连接通过管道。数据输入到过滤器，过滤器进行处理，通过管道输入到下一个过滤器…最后通过管道输出。管道过滤器风格的组件是过滤器，即处理数据的模块，组件的关系是管道，即传输数据的模块，该风格把数据经过滤器A处理后，通过管道A传输到过滤器B，然后再由过滤器B处理，以次类推。

优点：

1、高内聚，过滤器是包含处理服务并执行特定功能的组件，因此它的内聚程度高。

2、低耦合，过滤器仅通过管道通信，因此耦合程度较低。

3、简单性，过滤器可以很容易地用于并发或顺序系统的设计中。

4、可重用性，可以重用现有的过滤器，而不需要考虑其他过滤器。

5、灵活性，现有的过滤器可以很容易地重定义或修改。

6、可扩展性，可以很容易地添加新的过滤器，而不需要考虑其他过滤器。

缺点：

1、需要在管道中传输通用格式的数据。

2、难以将多种输入分派到相应的过滤器(在基于事件的交互中)。

### 存储器风格 Repository Model Style

存储器风格的组件是一个共享数据模块，以及多个客户端模块，多个客户端模块与共享数据模块相连，每一个客户端对共享数据的修改都对其他客户端可见。

有两个变种：

黑板风格Blackboard style：数据变化时，管理数据的组件会主动通知使用数据的组件变化。

存储风格Repository style：数据变化时，不会主动通知，只当使用的时候才会知道有数据变化。

优点：独立子系统内部是内聚的，耦合仅限于共享数据。单一的数据存储使数据共享更加高效。

缺点：共享数据中的任何更改都需要协议，并且可能需要所有或部分子系统中的更改。如果数据存储出现故障，所有子系统都会受到影响，可能必须停止。

### 客户端服务器风格Client-Server Style

组件是客户端模块和服务器模块，服务器模块负责数据存储，以及部分数据处理，客户端模块负责另一部分数据处理。

前面所述的存储器风格相当于特殊的客户端服务器风格，服务器模块只负责数据存储，而不负责数据处理。

有两种分类方法：

根据服务器模块的数量，分为两个变种：

两层客户端服务器Two Tier，优点：可以有效地利用网络资源；容易添加新的client/server或升级现有client/server；多个客户端之间可以共享数据。

三层客户端服务器Three Tier，优点：更好的性能与可伸缩性；安全措施可以集中分配；不同层可以并行开发。

根据客户端负责数据处理量的多少，分为两个变种：

胖客户端Thick Client：除了展示逻辑，还有一部分服务器的功能放在了客户端

瘦客户端Thin Client：应用比较少，只有展示逻辑

### 模型视图控制器风格Model-View-Controller(MVC) Style

M——model，封装了业务逻辑和数据库访问逻辑

V——view，用于显示用户界面

C——controller，封装控制逻辑

对应到客户端服务器风格，则model表示服务器，view和controller表示客户端。此外，模型视图控制器风格也可以被描述为分层的架构。

优点：组件之间耦合度低，可重用性好，可以满足用户对显示界面需求的不断变化

缺点：严重依赖开发和生产环境和工具

### 层次系统风格Layered System Style

组件是层级模块，高层的模块依赖于低层的模块。有两个分类：

严格的层次系统风格Strict layered system style：任何层只使用它正下方的层。

宽松的层次系统风格Relaxed layer system style：一个层可以使用它下面的任何层。

优点：

•每个层都被选择为一组相关的服务，因此它在层内提供了高度的内聚性

•层可能只使用较低的层，因此它限制了耦合的数量

•每一层都是内聚的，并且只与较低的层耦合，因此更容易添加、修改和/或重用一层

缺点：

•严格的分层样式可能会导致性能问题，这取决于层的数量

•设计干净的图层并不总是很容易

### P2P风格Peer-to-Peer Style

组件是Peer即同伴模块，同伴模块之间是平等的，任何一个同伴模块既依赖于其他同伴模块，使用其他同伴模块提供的服务，又被其他同伴模块所依赖，为其他同伴模块提供服务。

优点：去中心化；高容错；可扩展性强；最终一致性。

缺点：消息延迟；消息冗余。

### 事件驱动风格Event-Driven Style

组件是事件即消息的分发模块，以及消息的处理模块，消息到来之后，分发模块把消息发送给处理模块，处理模块进行处理。

有两个分类：

广播风格Broadcast：事件被分配给所有处理器。

中断驱动风格Interrupt-Driven：一个中断被发送到一个事件分配器(中断处理程序)，并传递给相应的处理器。

好处：事件调度器和事件处理器是分开的，保证了低解耦；任何处理器的故障都不会影响其他处理器。  
缺点：调度程序是性能瓶颈；调度员的故障会使整个系统瘫痪。

### SOA风格Service-Oriented Architecture Style

组件除了客户端模块、服务器模块，还有目录模块，目录模块记录网络上各个服务器模块的信息，为客户端模块的不同请求提供相应的服务器模块的地址，供客户端模块快速访问相应的服务器模块。

优点：测试容易、可伸缩性强、可靠性强、跨语言程度会更加灵活、团队协作容易、系统迭代容易。

缺点：运维成本过高，部署数量较多、接口兼容多版本、分布式系统的复杂性、分布式事务。

## 设计模式

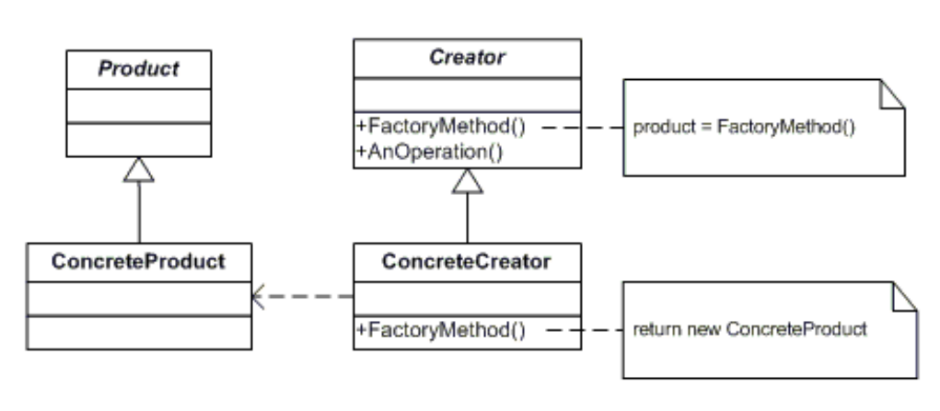
### 设计模式介绍

### 创建型模式Creational Patterns

#### 工厂方法模式Factory Method Pattern和抽象工厂模式Abstract Factory Pattern

工厂方法模式:

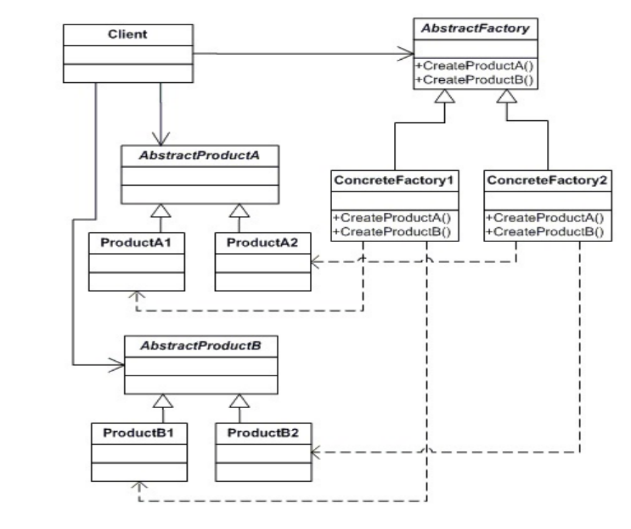
工厂方法模式是一种创建型设计模式， 其在父类中提供一个创建对象的方法，允许子类决定实例化对象的类型。

工厂方法模式建议使用特殊的工厂方法代替对于对象构造函数的直接调用 （即使用 new运算符）。 但对象仍将通过 new运算符创建， 只是该运算符改在工厂方法中调用。 工厂方法返回的对象通常被称作 “产品”，于是可以在子类中重写工厂方法， 从而改变其创建产品的类型

优点：避免创建者和具体产品之间的紧密耦合。单一职责原则：将产品创建代码放在程序的单一位置，从而使得代码更容易维护。

开闭原则：无需更改现有客户端代码，可以在程序中引入新的产品类型

抽象工厂模式：

抽象工厂模式是一种创建型设计模式，它能创建一系列相关的对象，而无需指定其具体类。首先，抽象工厂模式建议为系列中的每件产品明确声明接口。然后，确保所有产品变体都继承这些接口。然后需要声明抽象工厂，包含系列中所有产品构造方法的接口。这些方法必须返回抽象产品类型。

优点：可以确保同一工厂生成的产品相互匹配。可以避免客户端和具体产品代码的耦合。

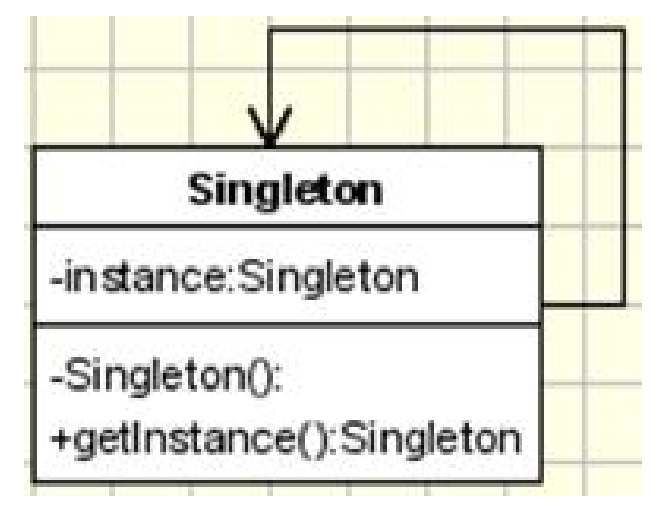
单一职责原则：可以将产品生成代码抽取到同一位置，使得代码易于维护。

开闭原则：向应用程序中引入新产品变体时，无需修改客户端代码。

#### 单例模式Singleton Pattern

组建是单例类，该类只能创建一个对象，如果再次调用则不能创建对象，这可以满足软件某个类只有一个对象的要求。

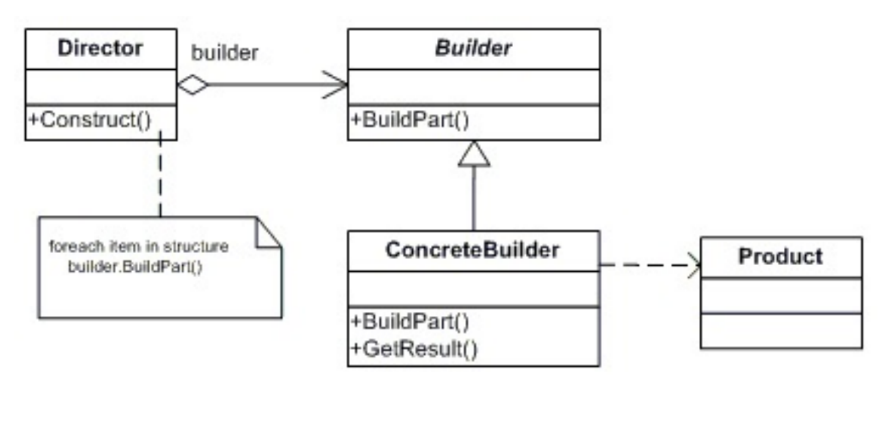
两种方法：构造器中判断；构造器私有，使用另外一个方法构造。

优点：1、由于单例模式在内存中只有一个实例，因此可以节约系统资源。2、单例模式可以提供对唯一实例的受控访问。3、单例模式可以在系统设置全局访问点，优化和共享资源访问。

#### 建造者模式Builder Pattern

将一个复杂对象的构造与它的表示分离，使同样的构建过程可以创建不同的表示，这样的设计模式被称为建造者模式。它是将一个复杂的对象分解为多个简单的对象，然后一步一步构建而成。它将变与不变相分离，即产品的组成部分是不变的，但每一部分是可以灵活选择的。

优点：封装性好，构建和表示分离；扩展性好，各个具体的建造者相互独立，有利于系统的解耦。

客户端不必知道产品内部组成的细节，建造者可以对创建过程逐步细化，而不对其它模块产生任何影响，便于控制细节风险。

简化类图：Director是使用建造者类的类，没有必要；Product是被创建对象的类，也没有必要；中间也没有必要继承。其实建造者模式关键就是有一个类来负责建造。建造者类负责把已经创建好的对象整合到一起。

### 结构型模式Structural Patterns

#### 适配器模式Adapter Pattern

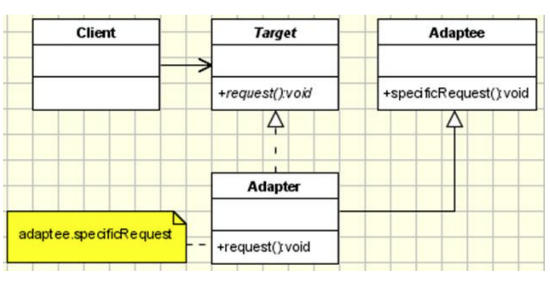
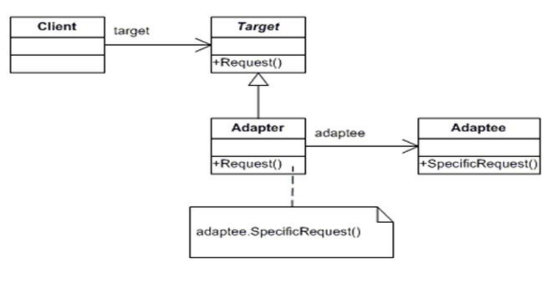
组件是适配器类，如果已有一个已经实现好的类，但是它的函数与定义好的接口即所需要使用的函数略有区别。

那么根据开闭原则，不应该修改这个已经实现好的类，而是要在类和接口之间加一个适配器类进行转换。

转换的方式分为两种：一是让适配器类继承已经实现好的类，再让适配器类实现接口，二是让适配器类包含已经实现好类的对象，再让适配器类实现接口。

1、增加一个新的类

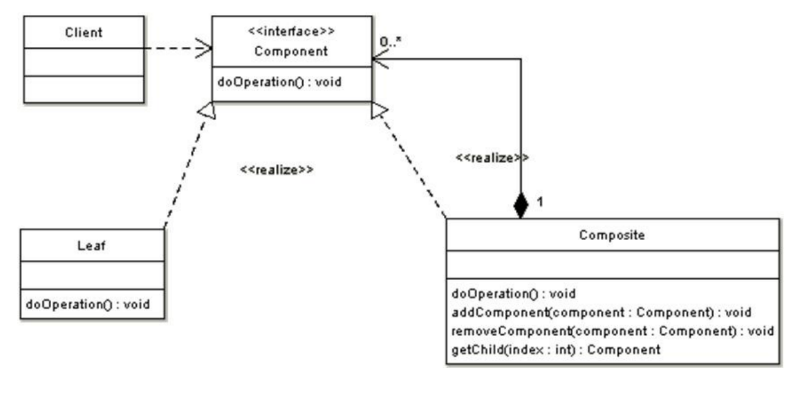
2、新的类实现接口

3、新的类调用旧的类的函数（继承）

#### 组合模式Composite Pattern

组件是组合类，该类包含一组函数，用于把该类的多个对象连接起来，构建对象树。

例子：composite类的addComponent()函数就是把对象连接起来的函数。而leaf叶子结点没有这个功能。

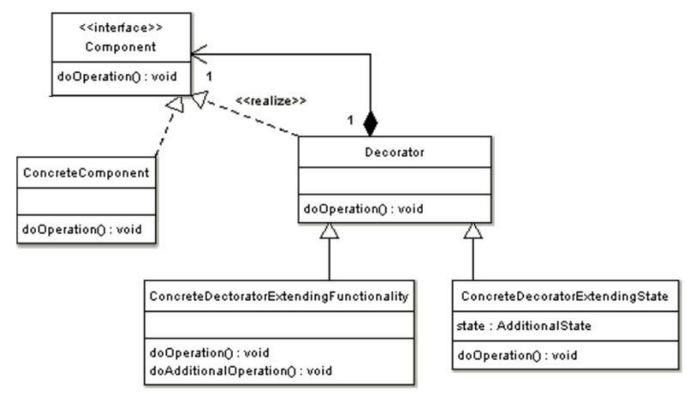
画类图的技巧：如果一个类和它实现的接口类/父类的所有的实现类/子类都有关联，那么这些关联关系可以都简化为这个类和它实现的接口类/父类的关联。（另一种情况：一个类和一个接口类/父类的所有实现类/子类相关联，那么这些关联关系都可以简化为这个类和这个接口类/父类的关联。）

优点：

1、组合模式使得客户端代码可以一致地处理单个对象和组合对象，无须关心自己处理的是单个对象，还是组合对象，这简化了客户端代码；

2、更容易在组合体内加入新的对象，客户端不会因为加入了新的对象而更改源代码，满足“开闭原则”。

#### 装饰器模式Decorator Pattern

当为一个原有类增加新的方法和功能时，可以通过装饰器类和装饰器子类，以增加新的方法。

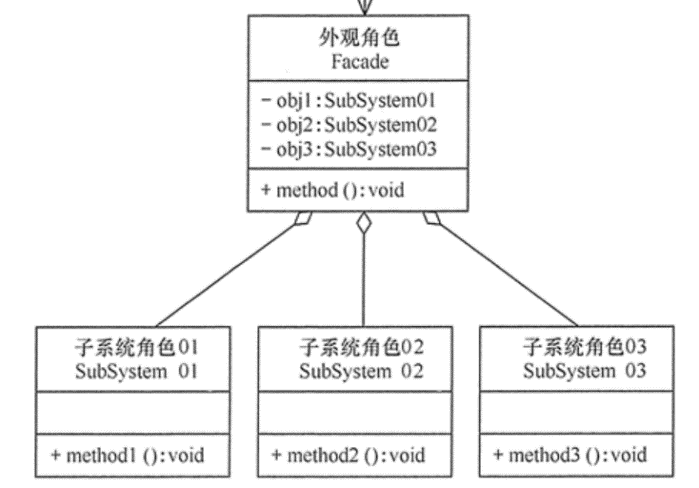
优点：

1、将类的核心方法和装饰性方法分开，符合单一责任原则，有助于提高聚合；

2、装饰类和被装饰类可以独立发展，不会相互耦合；

3、装饰器模式完全遵守开闭原则。

#### 外观模式Facade Pattern

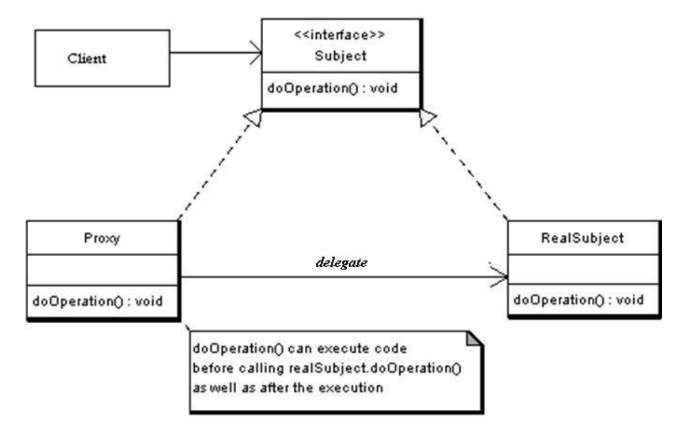
组件是外观类，某个模块里面的类和外面的类交互比较复杂的时候，为了简单起见，可以给模块增加一个外观类，让里面的类和外面的类通过这个外观类进行交互。

优点：

1、实现了子系统与客户端之间的松耦合关系；

2、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。

#### 代理模式Proxy Pattern

由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。

优点：

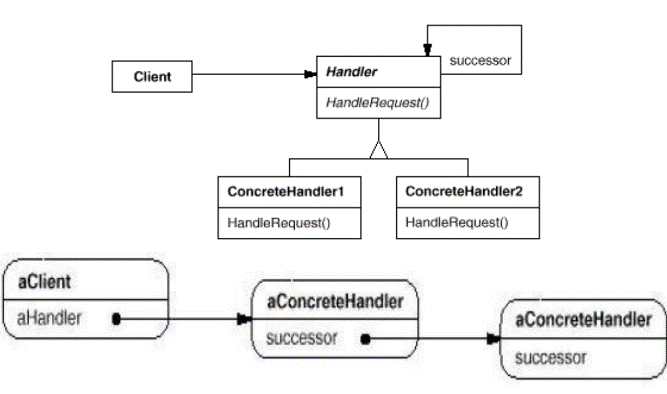
1、代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；

2、代理对象可以扩展目标对象的功能；

3、代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度，增加了程序的可扩展性。

### 行为型模式Behavioral Patterns

#### 责任链模式Chain of Responsibility Pattern

在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推，直到请求被一个接收者处理，那么将不用接受了。

优点：

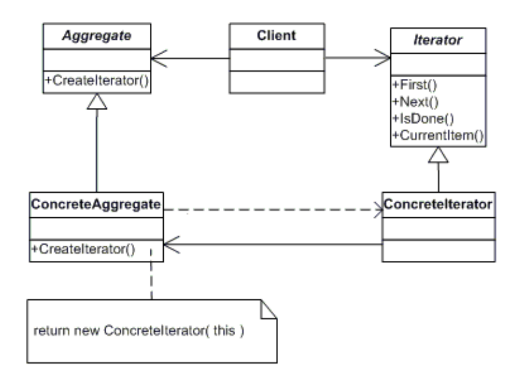
1、降低耦合度。它将请求的发送者和接收者解耦。

2、简化了对象。使得对象不需要知道链的结构。

3、增强给对象指派职责的灵活性。通过改变链内的成员或者调动它们的次序，允许动态地新增或者删除责任。

4、增加新的请求处理类很方便。

#### 迭代器模式Iterator Pattern

提供一个对象来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。

优点：

1、访问一个聚合对象的内容而无须暴露它的内部表示。

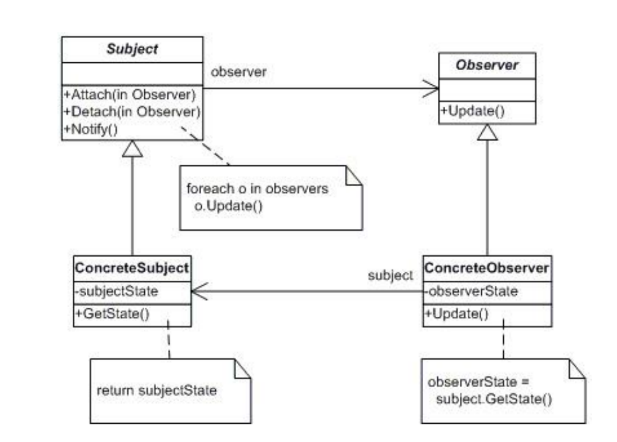
2、遍历任务交由迭代器完成，这简化了聚合类。

3、它支持以不同方式遍历一个聚合，甚至可以自定义迭代器的子类以支持新的遍历。

4、增加新的聚合类和迭代器类都很方便，无须修改原有代码。

5、封装性良好，为遍历不同的聚合结构提供一个统一的接口。

#### 观察者模式Observer Pattern

组件是观察者类和被观察类，如果需要被观察类的对象变化能够引起观察者类的对象变化，那么就需要在观察者类中引用被观察类的对象，以获得变化，也需要在被观察类中引用观察者类的对象，以通知发生了变化。画类图时，要把里面的函数写出来，才能体现观察和被观察的关系。

关键的函数：

Subject被观察者的getState()得到被观察者的状态。Notify()是通知观察者。

Observer观察者的函数update()。

优点：

1、观察者和被观察者是抽象耦合的。

2、建立一套触发机制。

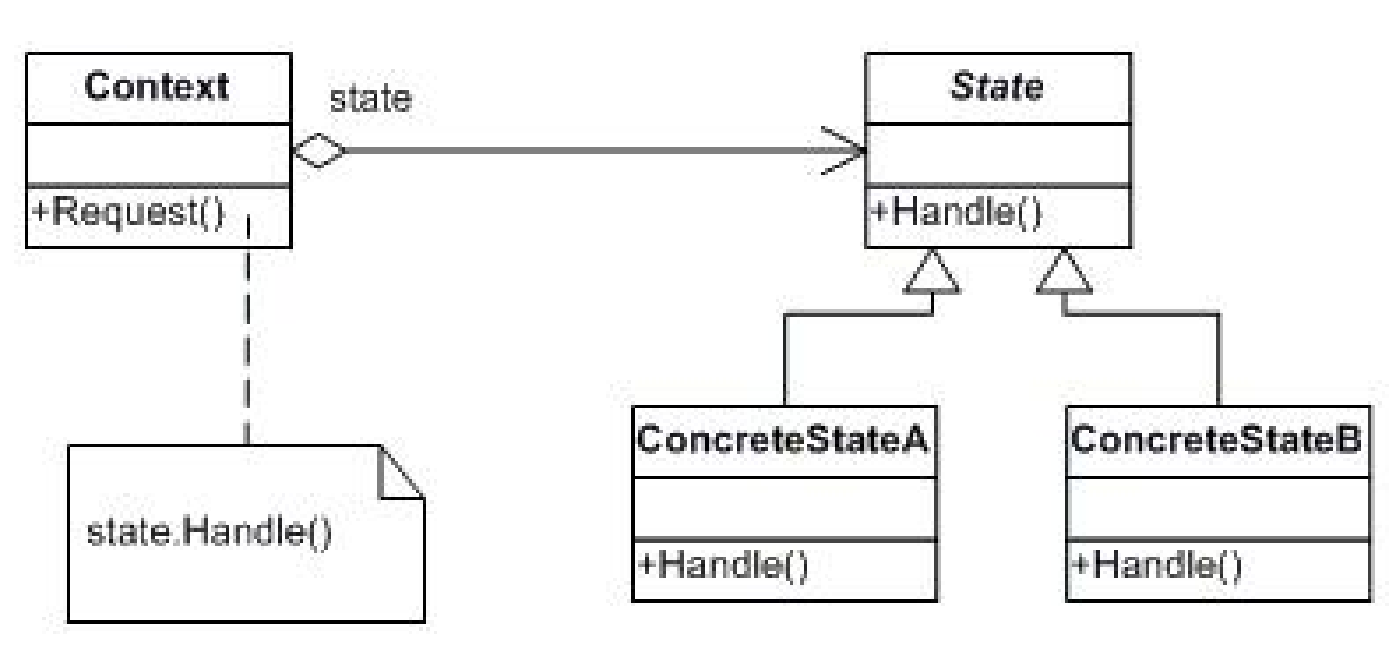
#### 状态模式State Pattern

组件是状态类，当某个类在不同状态下，需要有不同的数据处理方式时，可以把每一种处理方式放到一个新的类中，这样便于增加或更新状态和相应的处理方式。为每个状态增加一个状态类，直接调用某个状态对应状态类的对象的操作。

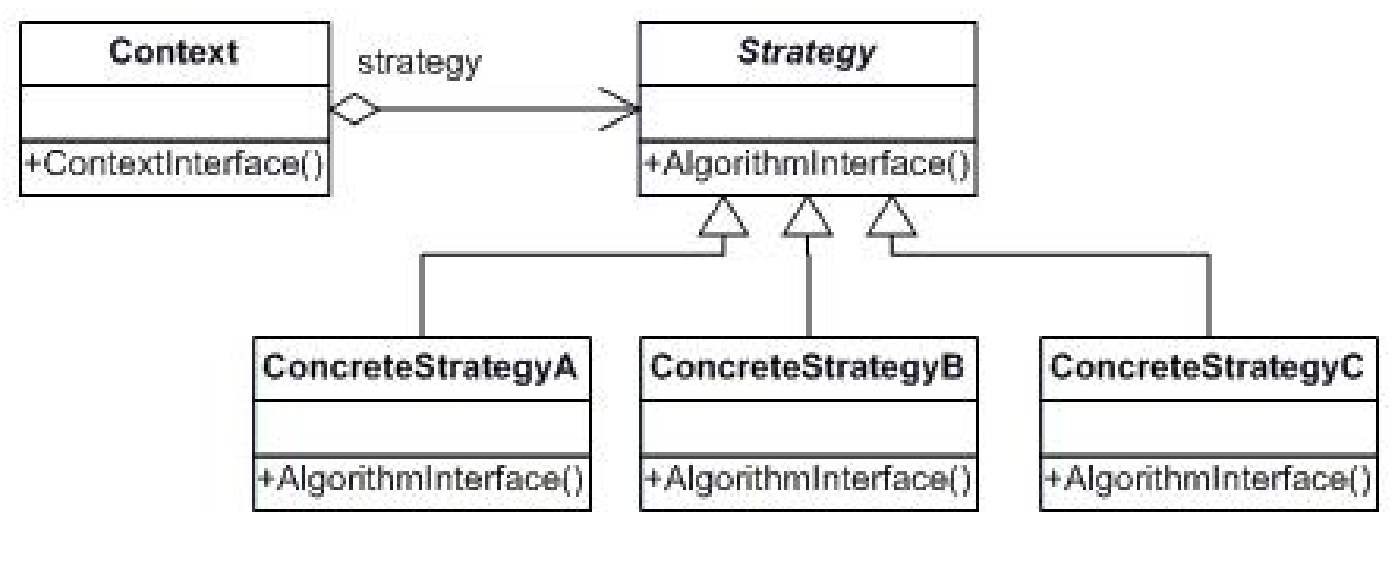
优点：

1、结构清晰，状态模式将与特定状态相关的行为局部化到一个状态中，并且将不同状态的行为分割开来，满足“单一职责原则”。

2、将状态转换显示化，减少对象间的相互依赖。将不同的状态引入独立的对象中会使得状态转换变得更加明确，且减少对象间的相互依赖。

3、状态类职责明确，有利于程序的扩展。通过定义新的子类很容易地增加新的状态和转换。

#### 策略模式Strategy Pattern

组件是策略类，当某个类在不同输入数据下，需要有不同但相似的数据处理方式时，可以把每一种处理方式放到一个新的类中，这样便于增加或更新数据和处理方式，然后让这些类继承自一个统一的父类以供调用。该模式与状态模式相似，都为不同的数据处理方式增加了新的类，但是状态模式是基于状态的；该类与责任链模式也相似，把每一种数据处理方式放到一个类里面，但是责任链模式是数据处理方式差别很大时所使用的。判断的不是状态，而是输入数据的类型。针对数据类型，构建策略类，和状态模式类似。

优点：

1、多重条件语句不易维护，而使用策略模式可以避免使用多重条件语句，如 if...else 语句、switch...case 语句。

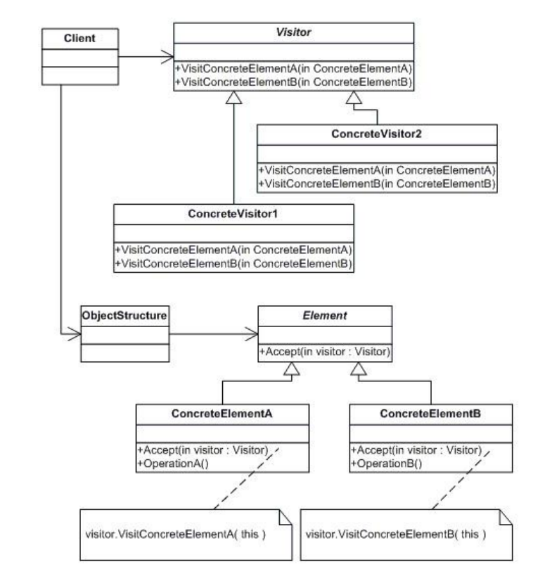
2、策略模式提供了一系列的可供重用的算法族，恰当使用继承可以把算法族的公共代码转移到父类里面，从而避免重复的代码。

3、策略模式可以提供相同行为的不同实现，客户可以根据不同时间或空间要求选择不同的。

4、策略模式提供了对开闭原则的完美支持，可以在不修改原代码的情况下，灵活增加新算法。

5、策略模式把算法的使用放到环境类中，而算法的实现移到具体策略类中，实现了二者的分离。

#### 访问者模式Visitor Pattern

将作用于某种数据结构中的各元素的操作分离出来封装成独立的类，使其在不改变数据结构的前提下可以添加作用于这些元素的新的操作，为数据结构中的每个元素提供多种访问方式。

优点：

1、扩展性好。能够在不修改对象结构中的元素的情况下，为对象结构中的元素添加新的功能。

2、复用性好。可以通过访问者来定义整个对象结构通用的功能，从而提高系统的复用程度。

3、灵活性好。访问者模式将数据结构与作用于结构上的操作解耦，使得操作集合可相对自由地演化而不影响系统的数据结构。

4、符合单一职责原则。访问者模式把相关的行为封装在一起，构成一个访问者，使每一个访问者的功能都比较单一。

## 软件产品线

### 软件产品线介绍

软件产品线是指具有一组可管理的公共特性的软件密集性系统的合集，这些系统满足特定的市场需求或任务需求，并且按预定义的方式从一个公共的核心资产集开发得到。一组类似的多个软件产品，比如QQ、微信、腾讯会议，都有相同的组件，但是又有不同的组件，这就叫做软件产品线。

优点：提高生产力；提高质量；降低成本；缩短上市时间。

三大开发组：核心组件（共有的部分）、每个产品的特有开发组、管理组。

### 软件产品线体系结构

软件产品线体系结构是对上述一系列软件核心部分的设计，既然是对核心部分的设计，那么它是没有包含对核心部分之外的其他部分的设计的，所以需要针对具体的软件来具体补充。

正常的软件体系结构是一个整体，像一块完整的蛋糕，而软件产品线的核心部分的体系结构是不完整的，像一块有洞洞的奶酪。只有共有的部分，留下了需要具体问题具体分析的地方。