设计模式报告

疫情地图模块

Epidemic Map Module

作者

3160104062 刘涵宇

3170105138 杨钰茹

3160101315 张童童

3160102473 肖韩

浙江大学计算机科学与技术学院战疫软工开发组

Date 2020-5-8

目录

1. 设计模式调查

1.1 概述

在软件设计中，合适的设计模式可以帮助软件设计者更好的进行软件设计，使得设计变得规范。

设计模式包含很多方面比如在一类问题中的常见错误，针对这类问题的解决方案等，并且它可以使得设计者在设计时能够避免一些将会引起问题的紧耦合，增强软件设计的适应变化的能力。

在《设计模式：可复用面向对象软件的基础》一书中，提到了多种设计模式，并把它们分为创建型、结构型与行为型三种模式大类。下本也将详细写到，针对其中几种设计模式进行调查后进行的分析与总结。

		Characterization		
		Creational	**Structural**	**Behavioral**
Jurisdiction	**Class**	Factory Method	Adapter (class) Bridge (class)	Template Method
	Object	Abstract Factory Prototype Solitaire	Adapter (object) Bridge (object) Flyweight Glue Proxy	Chain of Responsibility Command Iterator (object) Mediator Memento Observer State Strategy
	Compound	Builder	Composite Wrapper	Interpreter Iterator (compound) Walker

1.2　设计模式调查与分析

设计模式主要分为创建型模式、结构型模式、行为型模式和 J2EE 模式，我们就这几种模式的样例进行分析

1.2.1 创建型模式

创建型设计模式我们调查了 3 种设计模式：单例模式、建造者模式和工厂模式。

（1）单例模式

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

单例模式包含以下几个要素：

◆私有的构造方法;

◆指向自己实例的私有静态引用;

◆以自己实例为返回值的静态的公有的方法。

分析：

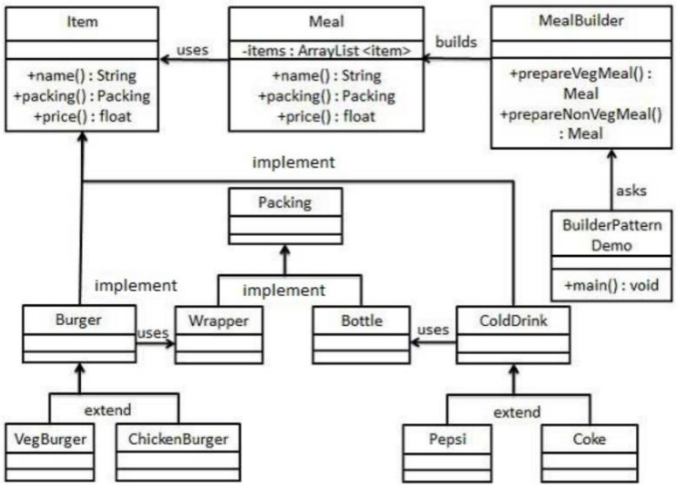
单例模式在内存中只有一个对象，节省内存空间，并且有指向实例的静态引用，可以避免频繁的创建销毁对象，提高性能，减少资源重复占用。

因此，在需要频繁实例化之后销毁的对象创建场景中我们可以应用单例模式，在频繁访问数据库或文件的对象创建时也适合单例模式的应用。总体来说单例模式比较简单，在软件设计中的应用也比较多。

(2) 建造者模式

建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

一个 Builder 类会一步一步构造最终的对象。该 Builder 类是独立于其他对象的。



建造者模式角色：

建造者：创建和提供实例

导演：管理建造出来的实例的依赖关系。

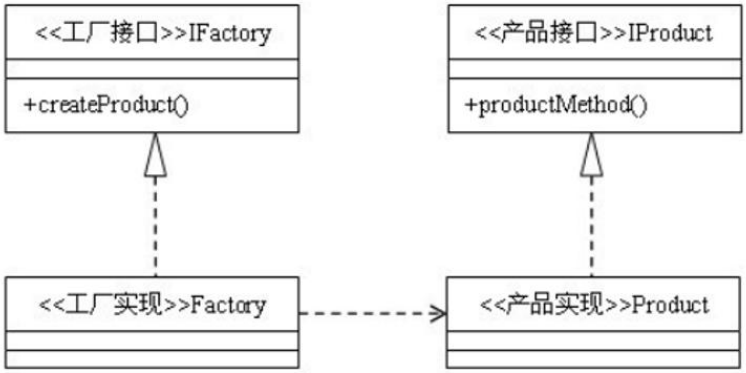
分析:

用建造者模式可以使客户端不必知道产品内部组成的细节。具体的建造者类之间相互独立，有利于系统的扩展。具体的建造者相互独立，可以对建造的过程逐步细化，而不会对其他模块产生任何影响。但建造者模式所创建的产品一般具有较多的共同点，其组成部分相似；如果产品之间的差异性很大，则不适合使用建造者模式，因此其使用范围受到一定的限制。如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大。

(3) 工厂模式

工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。



工厂方法模式中包含的角色及其相应的职责如下:

◆抽象工厂(Creator)角色：是工厂方法模式的核心，与应用程序无关。任何在 模式中创建的对象的工厂类必须实现这个接口。

◆具体工厂(Concrete Creator)角色：这是实现抽象工厂接口的具体工厂类， 包含与应用程序密切相关的逻辑，并且受到应用程序调用以创建产品对象。

◆抽象产品(Product)角色：工厂方法模式所创建的对象的超类型，也就是产 品对象的共同父类或共同拥有的接口。

◆具体产品(Concrete Product)角色：这个角色实现了抽象产品角色所定义 的接口。某具体产品有专门的具体工厂创建，它们之间往往一一对应。

分析:

工厂方法模式和所有的工厂模式一样，将产品的实例化封装起来，使得调用者根本无需关心产品的实例化过程，只需依赖工厂即可得到自己想要的产品。但是 区别于简单工厂模式，该模式中工厂不再负责产品的创建，由接口针对不同条件返 回具体的类实例，而由具体类实例去实现。

因此工厂方法模式增加了系统的可扩展性，但是在工厂方法模式中一个抽象工厂对应一个抽象产品，如果多个产品同时需要修改，那么要针对工厂类做出较大的改变，不易于维护。

1.2.2 结构型设计模式

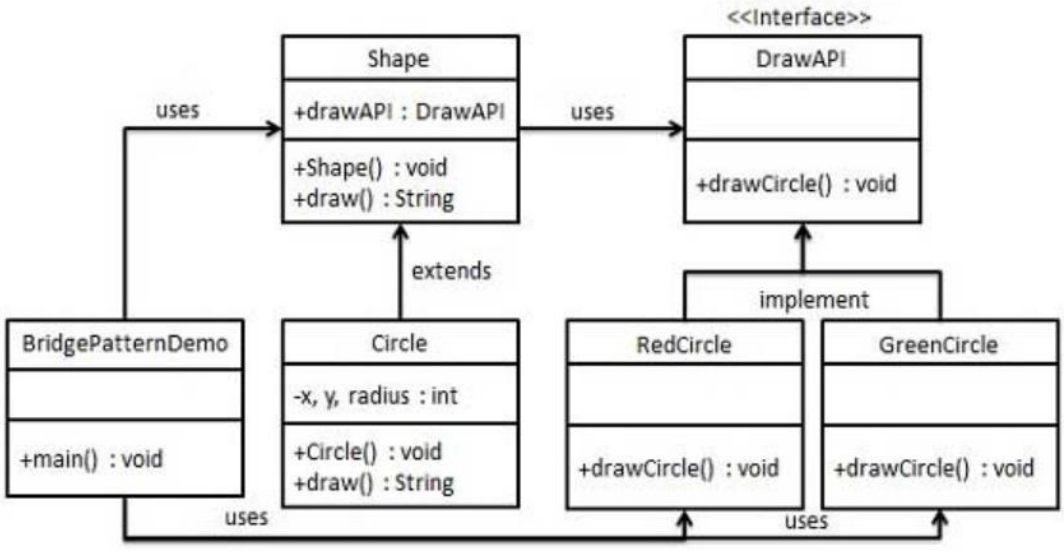
创建型模式非常容易理解，就是将对象集合的创建方法抽象出来“定义”了如 何去创建一类对象的基本方法。而结构型模式就是如何把已有的对象组装起来实现 新的功能。

桥接模式（Bridge Patten）：用于把抽象化与实现化解耦，使得二者可以独立变化。这种类型的设计模式属于结构型模式，它通过提供抽象化和实现化之间的桥接结构，来实现二者的解耦。

这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响。

桥接模式包含如下角色：

- **Abstraction**：抽象类
- **RefinedAbstraction**：扩充抽象类
- **Implementor**：实现类接口
- **ConcreteImplementor**：具体实现类



理解桥接模式，重点需要理解如何将抽象化(Abstraction)与实现化(Implementation)脱耦，使得二者可以独立地变化。

- 抽象化：抽象化就是忽略一些信息，把不同的实体当作同样的实体对待。在面向对象中，将对象的共同性质抽取出来形成类的过程即为抽象化的过程。
- 实现化：针对抽象化给出的具体实现，就是实现化，抽象化与实现化是一对互逆的概念，实现化产生的对象比抽象化更具体，是对抽象化事物的进一步具体化的产物。
- 脱耦：脱耦就是将抽象化和实现化之间的耦合解脱开，或者说是将它们之间的强关联改换成弱关联，将两个角色之间的继承关系改为关联关系。桥接模式中的所谓脱耦，就是指在一个软件系统的抽象化和实现化之间使用关联关系（组合或者聚合关系）而不是继承关系，从而使两者可以相对独立地变化，这就是桥接模式的用意。

分析：

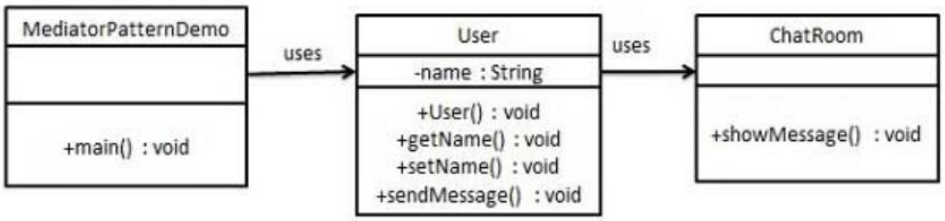
桥接模式实现了抽象和实现的分离，同时具备优秀的扩展能力，在实现细节对

客户透明。但桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。

1.2.3 行为型设计模式

（1）中介者模式

中介者模式（Mediator Pattern）是用来降低多个对象和类之间的通信复杂性。这种模式提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护。



中介者模式中包含的角色及其相应的职责如下：

抽象中介者（mediator）：定义一个接口用于和对象通信（SmartDevice）

具体中介者（concretemediator）：协调各同事对象实现协作，了解维护各个同事

抽象同事角色（colleague）：规定了同事的基本类型

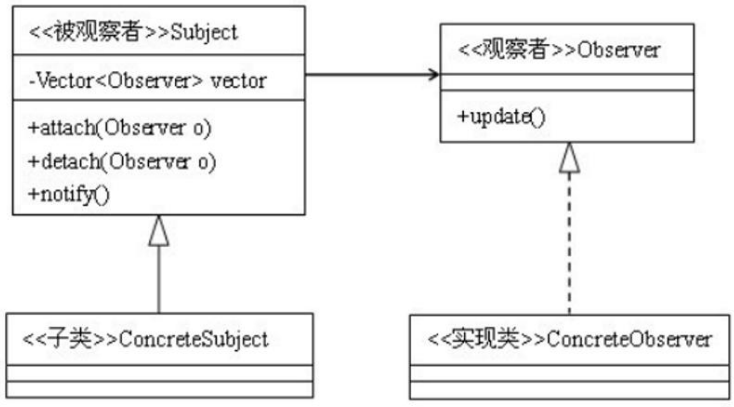
具体同事角色（concreteColleague）：每个同事都知道中介者对象，要与同事通信则把通信告诉中介者

分析：

中介者模式通过让对象彼此解耦，增加对象的复用性，同时将控制逻辑集中，便于化系统维护。通过中介者使一对所有变成了一对一，便于理解。但如果涉及不好，引入中介者会使程序变的复杂，此外中介者承担过多责任，维护不好会出大事

（2）观察者模式

观察者模式(Observer Pattern):该模式定义了一种一对多的依赖关系，让多个 观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知 所有观察者对象，使它们能够自动更新自己。



观察者模式中包含的角色及其相应的职责如下:

◆被观察者:从类图中可以看到,类中有一个用来存放观察者对象的 Vector 容器(之所以使用 Vector 而不使用 List,是因为多线程操作时,Vector 在是安全的,而 List 则是不安全的),这个 Vector 容器是被观察者类的核心,另外还有三个方法:attach 方法是向这个容器中添加观察者对象;detach 方法是从容器中移除观察者对象;notify 方法是依次调用观察者对象的对应方法。这个角色可以是接口,也可以是抽象类或者具体的类,因为很多情况下会与其他的模式混用,所以使用抽象类的情况比较多。

◆观察者:观察者角色一般是一个接口,它只有一个 update 方法,在被观察者状态发生变化时,这个方法就会被触发调用。

◆具体的被观察者:使用这个角色是为了便于扩展,可以在此角色中定义具体的业务逻辑。

◆具体的观察者:观察者接口的具体实现,在这个角色中,将定义被观察者对象状态发生变化时所要处理的逻辑。

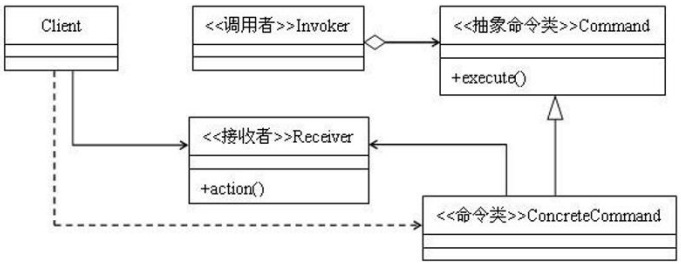
分析:

由观察者模式中的角色与职责可以看出,观察者模式中观察者和被观察者是轻度耦合的,观察者只是监听被观察者者的一个主题对象,有变化时自己做出相应的改变,在逻辑上的关联度并不是很高,因此系统的可扩展性较好。

观察者模型的应用范围也非常明显了,对于需要实现一条触发链的系统,这是一种比较适合的设计模式。但是在触发链的设计上,需要注意循环触发这一严重问题。

(3) 命令模式

命令模式(Command Pattern):该模式将一个请求封装成一个对象,从而让你使用不同的请求把客户端参数化,同时对请求排队或者记录请求日志,并提供命令的撤销和恢复功能。



命令模式中包含的角色及其相应的职责如下:

◆抽象命令角色(Command):抽象命令,包含命令执行的抽象方法

◆命令接收者(Receiver):命令接收者角色,它包含所有命令的具体行为实现方法。

◆具体命令角色(ConcreteCommand):它包含一个命令接收者对象,并调 用接收者的对象相应实现方法。

◆命令调用者角色(Invoker):提供给客户端调用,接收客户端所传递的具体 命令对象。

分析:

在命令模式中,我们不仅仅将命令直接封装起来提供调用,而是加入了调用者和接受者两个角色,使得一条命令将分步完成,降低耦合度,提高灵活性。并且将命令进行多层逻辑封装,可以重用底层封装,且确保了一定的扩展性,对于客户端来说不需要知道复杂的逻辑也很便捷。

2. 疫情地图模块设计模式分析

2.1 简单工厂模式（Simple Factory Pattern）：

简单工厂模式（Simple Factory Pattern）属于类的创新型模式，又叫静态工厂方法模式（Static Factory Method Pattern）,是通过专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类

简单工厂模式中包含的角色及其相应的职责如下：

工厂角色（Creator）：这是简单工厂模式的核心，由它负责创建所有的类的内部逻辑。当然工厂类必须能够被外界调用，创建所需要的产品对象。

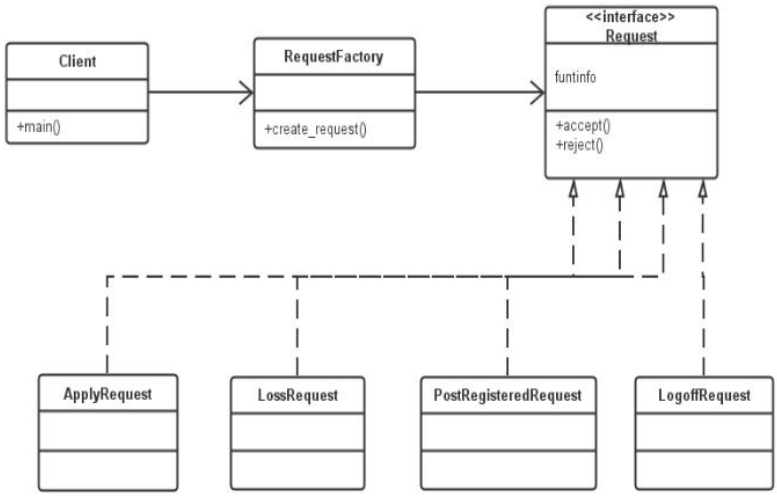
抽象（Product）产品角色：简单工厂模式所创建的所有对象的父类，注意，这里的父类可以是接口也可以是抽象类，它负责描述所有实例所共有的公共接口。

具体产品（Concrete Product）角色：简单工厂所创建的具体实例对象，这些具体的产品往往都拥有共同的父类。

应用：

用户在查看疫情地图模块时，通常会查看相关的疫情数据，选择相关的表格进行查看，在这一过程中，我们可以根据用户提供的详细字段请求该课程的对象

我们可为组这些请求提供一个接口，并且创建一个简单工厂模式：



总结：

因为请求创建模块较简单，简单工厂模式足以，不需要应用更复杂的工厂方法模式和抽象工厂模式。

请求工厂类是整个模式的关键，包含了必要的逻辑判断，根据外界给定的信息，决定究竟应该创建哪个具体请求类的对象。通过使用工厂类，外界可以从直接创建具体产品对象的尴尬局面摆脱出来，仅仅需要负责“消费”对象就可以了。而不必管这些对象究竟如何创建及如何组织的。明确了各自的职责和权利，有利于整个软件体系结构的优化。

简单工厂模式使得我们的系统在请求创建模块上具有更好的复用性。

2.2 责任链模式（Chain of Responsibility）：

在软件构建构成中，一个请求可能被多个对象处理，但是每个请求在运行时只能有一个接收者，如果显示指定，将必不可少地带来请求发送者与接收者的紧密耦合。COR(Chain of Responsibility)设计模式可以使请求的发送者不需要指定具体的接收者，让请求的接收者自己在运行时决定来处理请求，从而使两者解耦。

COR 将这些对象连成一条链，并沿着这条链传递请求，直到有一个对象对应它为止。

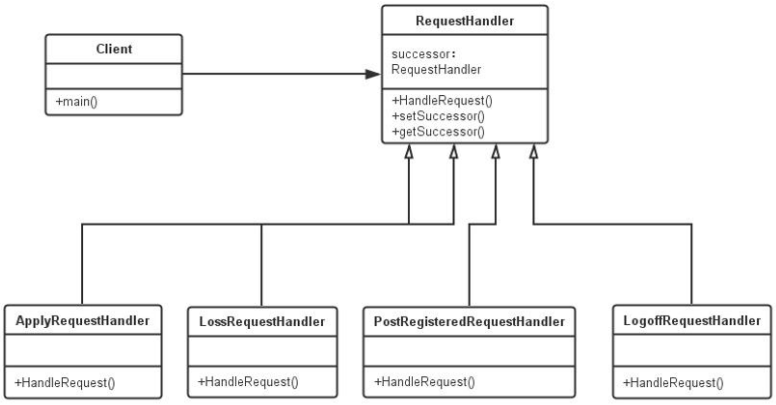
抽象处理者 (Handler) 角色 ： 定义出一个处理请求的接口。如果需要，接口可以定义出一个方法，以设定和返回对下家的引用。这个角色通常由一个抽象类或接口实现。

具体处理者 (Concrete Handler) 角色 ： 具体处理者接到请求后，可以选择将请求处理掉，或者将请求传给下家。由于具体处理者持有对下家的引用，因此，如果需要，具体处理者可以访问下家。

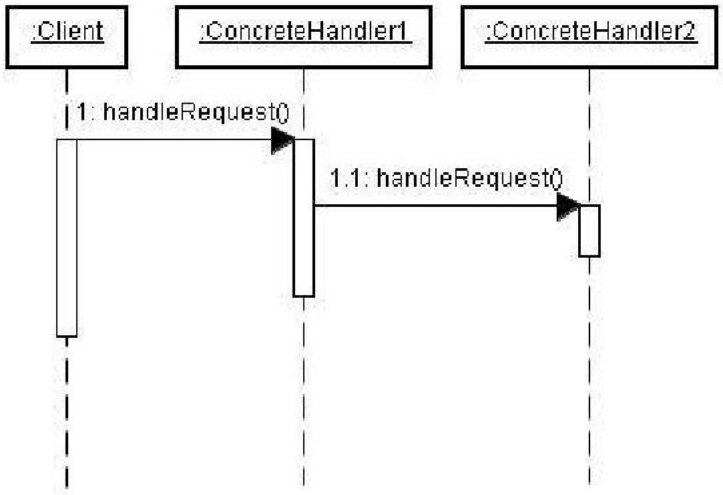
应用:

用户在查看疫情地图模块时，通常会查看相关的疫情数据，选择相关的表格进行查看，在这一过程中，我们可以根据用户提供的详细字段请求该课程的对象

我们可为组这些请求提供一个接口，并且创建一个简单工厂模式。并将请求传入，经过责任链选择适当的处理方法。



从物件执行请求的时间来看，其运作是很简单的职责传递而已，如下:



总结:

责任链模式降低了耦合度，分离了请求处理的发送者和接收者，并增加了处理的灵活性。然后在我们的选课系统中，通常一个课程搜索的目标比较明确，寻找到的课程往往就是用户的目标课程，所以课程搜索不一定非要使用责任链模式，但是总的来说，使用责任链模式提高了编码的灵活性和复用程度。

3. 疫情地图模块体系结构分析

3.1 系统关键质量

依据本子系统的软件需求分析说明书（SRS），本子系统的关键质量属性（QA）为 性能（performance）、安全性（security）、使用性（usability）和服务获得性 （availability）。其中在性能方面，负载容量和操作响应时间有明确的下界需求:

◆负载容量方面必须保证可允许 500 名用户同时访问。

◆响应时间方面单个操作响应时间不超过 1s，图片加载时间不超过 2s

3.2 在关键质量属性中选择的策略

3.2.1 性能（performance）

（1）Tactic 1

Introduce Concurrency（并发），毫无疑问，引入并发是提高性能必不可少的策略。

（2）Tactic 2

Prioritize Events（优先级），引入优先级有利用在系统负载过大时优先处理优先级较高的事物。

（3）Tactic 3 Reduce overhead（减少开销），减少开销包括减少计算开销和减少通信开销两方面。

应用：用户在进行注册时，一些格式匹配的工作可以在浏览器端直接进行验证，而无须在

服务器端进行，以减少网络通信所带来的开销。

3.2.2 安全性（security）

（1）Tactic 1

Authenticate actors（用户认证），用户认证保证进入系统的用户是合法用户。

应用：用户登录需要验证 ID 和密码的匹配性。

（2）Tactic 2

Authorize Actors（用户授权），用户授权保证合法用户进行的操作满足权限。

应用：管理员和管理员具有不同的权限，管理员能够查看所有用户的信息，但是无法修改用户的信息，而普通用户智能查看和修改自己的信息，无法查看其它用户的信息。

（3）Tactic 3

Limit Access（限制访问），通过 DMZ（防火墙非军事化区）来限制一些来自危险 IP 的访问和修改操作。

应用：用户无法直接访问数据库，必须通过应用服务器来访问，同时用户无法绕过应用服务器查看一些敏感信息，如系统日志等。

（4）Tactic 4

Encrypt Data（数据加密），在网络上进行传输的数据必须进行加密，存储在数据库中的用户数据也必须进行加密。

应用：用户的密码在数据库中必须进行加密，可以采用当前主流的加盐哈希算法（数据冗余，有效防止字典攻击）进行加密，同时在对资金账户中资金进行操作时必须进行数字证书的认证。需要提醒的一点是加密算法必须使用当前主流的普遍使用的加密算法进行加密，如 RAS 加密算法、AES 加密算法和 MD5 加密算法，在任何情况下不能使用自己开发的加密算法。

（5）Tactic 5

Change Default Settings（改变缺省设置），用户在首次进入系统后改变公共的缺省设置，以防止通过这些缺省的设置所进行的攻击。

应用：在本子系统中应该尽量减少缺省设置。

（6）Tactic 6

Recover from Attacks（攻击后恢复），通过进行数据备份，结合系统日志和检查点（checkpoint）来进行事物的 redo 和 undo。

3.2.3 服务获得性（availability）

3.2.3.1 概述

衡量指标：

a = MTBF /（MTBF + MTTR）

MTBF：Mean time between failure，即平均故障时间。

MTTR：Mean time to recover，即平均恢复时间。

3.2.3.2 策略

（1）Tactic 1

Exception Detection（异常检测），异常检测包括操作系统异常，常数异常等情况，这里要讨论的主要是超时异常（timeout）。

应用：当用户进行操作，出现网络中断，应该设置一个超时时间，这样在超时异常发生的情况下能够进行回退。

（2）Tactic 2

Rollback（回滚），事物的回滚一般与 Redundancy（冗余）策略一起使用。

应用：回滚策略能够在发生 failure 的情况下通过系统日志和检查点回滚到之前一个状态，在本系统中涉及到账户资金和信息安全的各种操作都应该能够进行回滚。

（3）Tactic 3

Passive Redundancy（被动备份），被动备份能够进行 failure 的恢复和灾难恢复。

（4）Ttactic 4

State Resynchronization（状态同步），在被动冗余的情况下，状态的同步操作一般由主服务器定期更新备份服务器的数据来完成。

3.2.4 使用性（usability）

（1）Tactic 1

Cancel（取消），取消策略几乎应用在了所有的体系结构设计中。

（2）Tactic 2

Undo（撤销），撤销操作同样在几乎所有的体系结构设计中有所体现。

（3）Tactic 3

Maintain User Model（维护用户模型），记录用户的设置。

3.2.5 可变更性（modifiability）

3.2.5.1 概述

在本子系统的软件需求分析说明书（SRS）中，特别提到了本子系统中在开发中需要实现易于维护，修改以达到迭代开发的要求。

3.2.5.2 策略

（1）Tactic 1

Coupling（耦合），耦合性描述了系统中两个模块之间的相互影响性。即当其中一个模块被修改后另外一个模块也需要修改的程度。

应用：在本子系统的软件需求说明书中明确要求要达到低耦合性的要求。

（2）Tactic 2

Cohesion（内聚），内聚性描述了一个模块的功能紧密结合的程度。

应用：在本系统的软件需求说明书中明确要求需要达到高耦合的要求。因此在进行模块划分时需要进行精心设计。

（3）Tactic 3

Split Module（模块分解），当一个模块被修改时，如果该模块非常的复杂，那么修改所带来的时间和人力开销将会大大增加。

3.2.5.3 总结

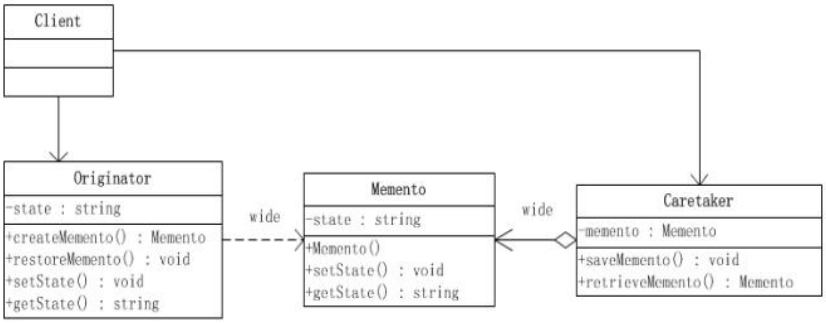
良好的可变更性最终需要达到的目标就是“高内聚，低耦合”，这需要软件架构人员需要有良好的设计经验和对业务逻辑深入的掌握。

4 其他可用设计模式分析

4.1 备忘录模式（Memento Pattern）

备忘录模式又叫做快照模式(Snapshot Pattern)或 Token 模式，是对象的行为模式。

备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。备忘录模式的用意是在不破坏封装的条件下，将一个对象的状态捕捉(Capture)住，并外部化存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。



备忘录模式所涉及的角色有三个：备忘录(Memento)角色、发起人 (Originator)角色、负责人(Caretaker)角色。

（1）备忘录(Memento)角色

◆将发起人（Originator）对象的内战状态存储起来。备忘录可以根据发起人 对象的判断来决定存储多少发起人（Originator）对象的内部状态。

◆备忘录可以保护其内容不被发起人（Originator）对象之外的任何对象所读 取。

（2）发起人（Originator）角色

◆创建一个含有当前的内部状态的备忘录对象。

◆使用备忘录对象存储其内部状态。

（3）负责人（Caretaker）角色

◆负责保存备忘录对象。

◆不检查备忘录对象的内容。

分析:

如果我们想提高系统的可靠性，则可以使用备忘录模式记录管理员的操作，以防管理员出错或系统崩溃等意外状况。

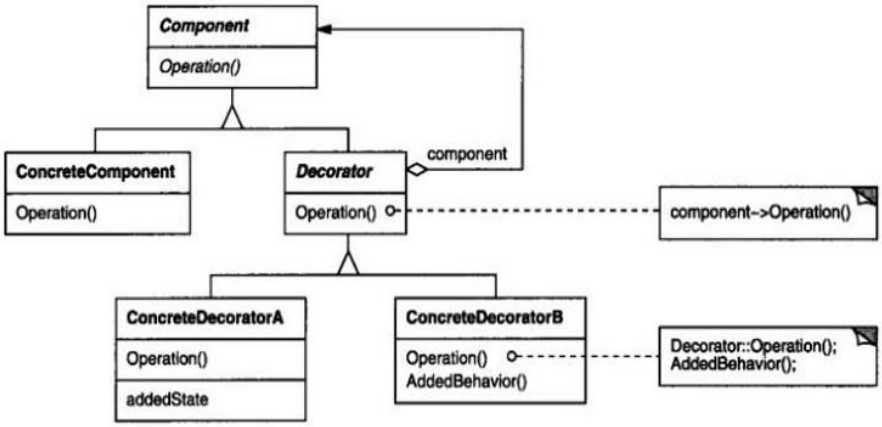
使用备忘录模式，把复杂的发起人内部信息对其他的对象屏蔽起来，从而可以、恰当地保持封装的边界。同时简化了管理，可自行进行存储和恢复状态。

4.2 装饰器模式（Decorator）

使用装饰器模式允许动态地给一个对象添加一些额外的职责或者行为。相对于通过继承来增加特性，装饰器模式允许动态添加对象的功能，能避免代码重复和类数量增加。

使用装饰器模式允许动态地给一个对象添加一些额外的职责或者行为。相对于通过继承来增加特性，装饰器模式允许动态添加对象的功能，能避免代码重复和类

数量增加。



装饰器模式中包含的角色及其相应的职责如下:

◆抽象组件角色(Component):定义一个对象接口以规范准备接受附加责任 的对象。

◆具体组件角色(ConcreteComponent) :被装饰者,定义一个将要被装饰增 加功能的类,可以给这个类的对象添加一些职责。

◆抽象装饰器(Decorator):维持一个指向构件 Component 对象的实例,并 定义一个与抽象组件角色 Component 接口一致的接口。

◆具体装饰器角色 (ConcreteDecorator):向组件添加职责。

分析:

在编程中用子类对类的功能进行扩展时应该谨慎,如果预计模块在之后被修改的可能性大则尽量使用装饰器模式进行扩展,增加软件的可修改性。同时装饰器模式不能过度使用,否则许多小类会使程序变得复杂,难以理解。

5 参考

[1] Roger S. Pressman, 软件工程：实践者的研究方法[M],机械工业出版社, 2011

[2] Freeman E. HeadFirst 设计模式[M]. 中国电力出版社, 2007

[3] Gamma E, Helm R, Johnson R, John V. 设计模式可复用面向对象的软件基础[M]. 机械工业出版社, 2007.