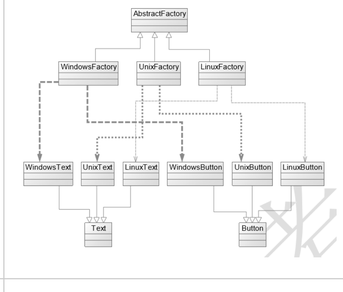
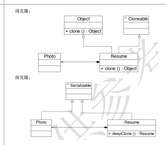
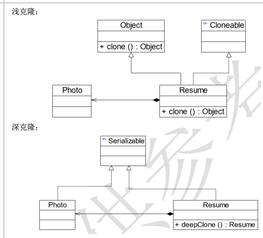
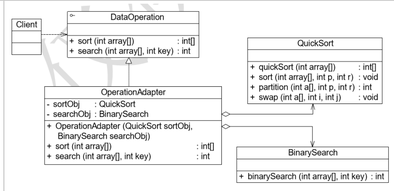
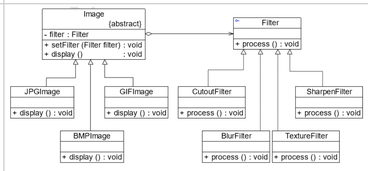
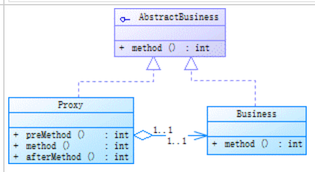
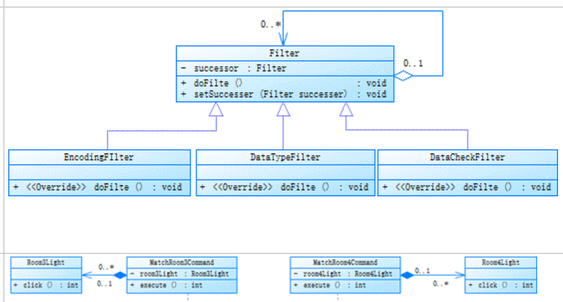
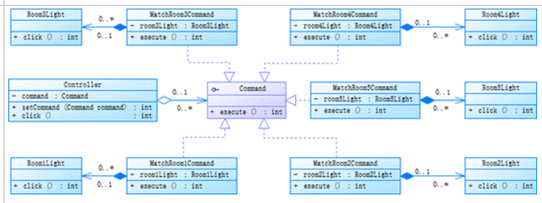
软件体系结构-设计模式

2020年1月3日 17:49

苏翔捷同学呕心沥血之作(*ᵔᵕᵔ)

名称	定义	角色	优点	缺点	适用场景	类图	备注
简单工厂模式 **Simple Factory Pattern** **静态工厂方法** **Static Factory Method**	定义一个工厂类，它可以根据参数的不同返回不同类的实例，被创建的实例通常都有共同的父类。 由于创建实例的方法通常是静态（static）方法，因此又被称为静态工厂方法（Static Factory Method）模式	Factory 工厂角色 Product 抽象产品角色 ConcreteProduct 具体产品角色	实现了对象创建和使用的分离； 客户端无需知道所创建的具体产品类的类名； 可以在不修改任何客户端代码的情况下，更换和新增新的具体产品类。	工厂类职责过重； 增加系统中类的个数； 系统扩展困难，一旦新增产品不得不修改工厂逻辑； 由于使用了静态工厂方法，造成工厂角色无法形成基于继承的等级结构（静态方法不支持多态 --> 子类无法覆盖父类）。	工厂类负责创建的对象比较少； 客户端只知道传入工厂类的参数，对于如何创建对象并不关心。		
工厂方法模式 **Factory Method Pattern** **虚拟构造器模式** **Virtual Constructor Pattern** **多态工厂模式** **Polymorphic Factory Pattern**	定义一个用于创建对象的接口，但是让子类决定将哪一个类实例化。工厂方法模式让一个类的实例化延迟到其子类。 Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.	Product 抽象产品 Concrete Product 具体产品 Factory 抽象工厂 Concrete Factory 具体工厂	向客户隐藏了哪种具体产品类将被实例化这一细节； 让工厂自主确定创建何种产品对象； 完全符合开闭原则；	系统中类的个数成对增加； 增加了系统的抽象性和理解难度。	客户端不知道它所需要的对象类； 抽象工厂类通过其子类来指定创建哪个对象。		
抽象工厂模式 **Abstract Factory Pattern**	提供一个船舰一系列相关或相互依赖对象的接口，而无须指定它们具体的类。 Provide an interface for creating families of related or depent objects without specifying their concrete classes.	Abstract Factory 抽象工厂 Concrete Factory 具体工厂 Abstract Product 抽象产品 Concrete Product 具体产品	隔离了具体类的生成； 能够保证客户端始终只适用同一个产品族中的对象； 增加新的产品族很方便，符合开闭原则。	增加新的产品等级结构麻烦，违背开闭原则。	一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节； 每次只适用某一产品族； 属于同一产品族的产品将在一起适用。 产品等级结构稳定。		
建造者模式 **Builder Pattern**	将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。 Separate the construction of a complex object from its representation so that the same construction process can create different representations.	Builder 抽象建造者 Concrete Builder 具体建造者 Product 产品 Director 指挥者	客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象； 可以很方便地替换具体建造者，或增加新的建造者，扩展方便，符合开闭原则； 可以更加精细地控制产品的创建过程。	如果产品之间的差异性很大，不适合适用建造者模式，因此其适用范围收到一定的限制； 产品的内部变化复杂，可能会需要定义很多具体建造者类。	需要生成的产品对象有复杂的内部结构； 需要生成的产品对象的属性相互依赖； 对象的创建过程独立于创建该对象的类； 隔离复杂对象的创建和适用。		
原型模式 **Prototype Pattern**	适用原型实例指定待创建对象的类型，并且通过赋值这个原型来创建新的对象。 Specify the kinds of objects to create using a prototypical instance, and create new objects by copyingg this prototype.	Prototype 抽象原型类 Concrete Prototype 具体原型类 Client 客户类	简化对象的创建过程，提高新实例的创建效率； 提供了简化的创建结构； 可以适用深克隆的方式，保存对象的状态。	需要为每一个类配备一个克隆方法，且克隆方法位于每一个类的内部，当对已有的类进行改造时，将违背开闭原则； 实现深克隆时需要编写较为复杂的代码。	创建新对象成本较大； 系统要保存对象的状态，而对象的状态变化很小。		浅克隆 Shallow Clone：只复制它本身和其中包含的值类型的成员变量，而引用类型的成员变量并没有复制。 深克隆 Deep Clone：除了对象本身被复制外，对象所包含的所有成员变量也将被复制。

单例模式
Singleton Pattern
	确保一个类只有一个实例，并提供一个全局访问点来访问这个唯一实例。
	Ensure a class has only one instance, and provide a global point of access to it.
		Singleton 单例	提供了对唯一实例的受控访问；
			可以节约系统资源，提高系统的性能；
			允许可变数目的实例（多例类）。
				扩展困难（缺少抽象层）；
				单例类的职责过重；
				由于自动垃圾回收机制，可能会导致共享的单例对象的状态丢失。
					系统只需要一个实例对象，或值允许创建一个对象；
					客户调用类的单个实例只允许适用一个公共访问点。
							饿汉式单例类 Eager Singleton：加载类时就创建对象；
							懒汉式单例类 Lazy Singleton：调用时才创建对象。
							双重检查锁定（Java）：
							`if (gameManager == null) {`
							`  synchronized (GameManager.class) {`
							`  if (gameManager == null) {`
							`  gameManager = new GameManager();`
							`  }`
							`  }`
							`}`
							`return gameManager;`

适配器模式 **Adapter Pattern**	将一个类的接口转换成客户希望的另一个接口。适配器模式让那些接口不兼容的类可以一起工作。 Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. 对象结构型模式 / 类结构型模式	Target 目标抽象类 Adapter 适配器类 Adaptee 适配者类	将目标类和适配者类解耦； 增加了类的透明性和复用性； 灵活性和扩展性非常好 类适配器模式：置换一些适配者的方法很方便（可以调用或覆盖适配者的方法）； 对象适配器模式：可以把多个不同的适配者适配到同一个目标，还可以适配一个适配者的子类。	类适配器模式： 一次最多只能适配一个适配者类； 适配者类不能为最终 final 类； 目标抽象类只能为接口，不能为类。 对象适配器模式： 在适配器中置换适配者类的某些方法比较麻烦（编写适配者子类覆盖该方法并注入适配器类）	系统需要适用一些现有的类； 创建一个可以重复适用的类，用于和一些彼此之间没有太大关联的类一起工作。		缺省适配器模式 Default Adapter Pattern： 不需要实现一个接口提供的所有方法时，设计一个抽象类实现该接口，其中每个方法提供一个默认实现（空方法），这一抽象类的子类无需实现所有方法。又称：但接口适配器模式。 双向适配器：……。
桥接模式 **Bridge Pattern**	将抽象部分与它的实现部分解耦，使得两者都能够独立变化。 Decouple an abstraction from its implementation so that the two can vary independently. 对象结构型模式	Abstraction 抽象类 Refined Abstraction 扩充抽象类 Implementor 实现类接口 Concrete Implementor 具体实现类	分离抽象接口及其实现部分； 极大地减少了子类的个数； 提高了系统的可扩展性。	增加系统的理解与设计难度； 正确识别出系统中两个独立变化的维度，并不是以见容易的事情。	避免在两个层次之间建立静态的继承关系； 抽象部分和实现部分可以以继承的方式独立扩展而互不影响； 一个类存在两个或多个独立变化的维度； 不希望适用继承，或因为多层基层导致系统类的个数急剧增加的系统。		
组合模式 **Composite Pattern**	组合多个对象形成树形结构以表示具有部分-整体关系的层次结构。组合模式让客户端可以统一对待单个对象和组合对象。 Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. 对象结构型模式	Component 抽象构建 Leaf 叶子构建 Composite 容器构建	定义分层次的复杂对象，让客户端忽略了层次的差异； 一致地适用一个组合结构，或其中单个对象，简化客户端代码； 增加新的容器构件和叶子构件都很方便； 树形结构的面向对象实现。	很难对容器中的构件类型进行限制。	具有整体和部分的层次结构，客户端可以一致地对待它们； 面向对象语言开发的系统中需要处理一个树形结构； 能够分离出叶子对象和容器对象，而且它们的类型不固定，需要增加一些新的类型。		
外观模式 **Façade Pattern**	为子系统中的一组接口提供一个统一的入口。外观模式定义了一个高层接口，这个接口使得这一子系统更加容易适用。 Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use. 对象结构型模式	Façade 外观角色 Subsystem 子系统角色	对客户端屏蔽了子系统组件，减少了哭护短所需处理的对象数目，并使得子系统使用起来更加容易； 实现了子系统与客户端之间的松耦合关系； 一个子系统的修改对其他子系统没有任何影响，子系统的内部变化也 不会影响到外观对象。	不能很好地限制客户端直接适用子系统类； 如果设计不当，增加新的子系统可能需要修改外观类的源代码，违背开闭原则。	要为一系列复杂的子系统提供一个简单入口； 客户端程序与多个子系统之间存在很大的依赖性； 可以适用外观模式的定义系统中每一层的入口，降低层之间的耦合度。		抽象外观类：若增加、删除或更换与外观类交互的子系统，必须修改外观类或客户端源代码，违背开闭原则。故引入抽象外观类。
代理模式 **Proxy Pattern**	给某一个对象提供一个代理或占位符，并由代理对象来控制对原对象的访问。 Provide a surrogate or placeholder for another object to control access to it. 对象结构型模式	Subject 抽象主题角色 Proxy 代理主题角色 Real Subject 真实主题角色	协调调用者和被调用者，降低了系统的耦合度； 增加和更换代理类无需修改源代码，符合开闭原则。 远程代理：提高系统的整体运行效率； 虚拟代理：一定程度上节省系统开销； 缓冲代理：优化系统性能，缩短执行时间； 保护代理：可以控制对一个对象的访问权限。	有些类型的代理模式可能会造成请求的粗粒速度慢（如保护代理）； 有些代理模式的实现过程较为复杂（如远程代理）。	（备注）		远程代理 Remote Proxy：本地对象调用远程对象； 虚拟代理 Virtual Proxy：资源消耗小的调用大的； 保护代理 Protect Proxy：控制权限； 缓冲代理 Cache Proxy：常驻内存，用内存对象代替硬盘对象； 智能引用代理 Smart Reference Proxy：当一个对象被引用时，提供一些额外的操作。
职责链模式 **Chain of Responsibility Pattern**	避免将一个请求的发送者与接收者耦合在一起，让多个对象都有机会处理请求。将接受请求的对象连接成一条链，并且沿着这条链传递请求，直到有一个对象能够处理它为止。 Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. 对象行为模式	Handler 抽象处理者 Concrete Handler 具体处理者	降低了系统的耦合度； 可简化对象之间的相互连接； 职责的分配带来更多的灵活性； 增加一个虚拟的具体请求处理者时，无需修改原有系统的代码。	不能保证请求一定会被处理； 系统性能将收到一定影响，在进行代码调试时不太方便； 如果建链不当，可能会造成循环调用，将导致系统陷入死循环。	有多个对象可以处理同一个请求，待运行时刻再决定； 向多个对象中的一个提交一个请求； 可动态指定一组对象处理请求。		纯的职责链模式：一个具体处理者对象只能承担全部责任，或者将所有责任推给下家。一个请求必须被某一个处理者对象所接受。 不纯的职责链模式：处理者可以部分处理职责，并转发给下家。
命令模式 **Command Pattern**	将一个请求封装为一个对象，从而让你可以用不同的请求对客户进行参数化，对请求排队或者记录请求日志，以及支持可撤销的操作。 Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. 对象行为型模式	Command 抽象命令类 Concrete Command 具体命令类 Invoker 调用者 Receiver 接收者	降低系统的耦合度； 符合开闭原则； 可较容易地设计一个命令队列或宏命令； 为请求的撤销Undo和恢复Redo提供一种方案。	可能会导致某些系统又过多的具体命令类。	需要将请求调用者和请求接收者解耦； 需要在不同的时间指定请求、将请求排队和执行请求； 需要支持命令的撤销和恢复操作； 需要将一组操作组合在一起形成宏命令。		实现命令队列：增加一个 CommandQueue类，负责存储多个命令对象； 记录请求日志：通过序列化 Serializable 写到日志文件（Log File）中； 实现撤销操作：根据需要修改代码； 宏命令 Macro Command（组合命令 Composite Command）：递归调用它所包含的每个成员命令的 execute 方法，实现对命令的批处理。
观察者模式 **Observer Pattern**	定义对象之间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象都得到通知并被自动更新。 Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. 对象行为型模式	Subject 目标 Concrete Subject 具体目标 Observer 观察者 Concrete Observer 具体观察者	可以实现表示层和数据逻辑层的分离； 在观察目标耦合观察者自建建立一个抽象的耦合； 支持广播通信，简化了一对多系统设计的难度； 符合开闭原则。	将所有观察者都通知到会花费很多时间； 如果存在循环依赖时，可能导致系统崩溃； 没有相应的机制让观察者直到所观察的目标对象时怎么发生变化的。	一个抽象模型有两个方面，其中一个方面依赖于另一个方面，两个方面可以各自独立地改变和复用； 一个对象的改变将导致一个或多个其他对象发生改变，并且不知道具体由多少对象将发生改变，也不知道这些对象是谁； 需要在系统中创建一个触发链。		
策略模式 **Strategy Pattern**	定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法可以独立于适用它们的客户变化。 Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. 对象行为型模式	Context 环境类 Strategy 抽象策略类 Concrete Strategy 具体策略类	提供了对开闭原则的完美支持； 提供了管理相关的算法族的办法； 提供了一种可以替代继承关系的办法； 可以避免多重条件选择语句（if-else）； 提供了一种算法的复用机制。	客户端必须知道所有的策略类，并自行选择一种； 将产生很多具体策略类； 无法同时在客户端能使用多个策略类。	需要动态地在几种算法中选择一种； 避免适用难以维护的多重条件选择语句； 提高算法的保密性与安全性。		
模板方法模式 **Template Method Pattern**	定义一个操作中算法的框架，而将一些步骤延迟到子类中。模板方法模式使得子类不改变一个算法的结构，即可重定义该算法的某些特定步骤。 Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. 类行为型模式	Abstract Class 抽象类 Concrete Class 具体子类	子类实现详细的处理算法时，并不会改变算法中步骤的执行次序； 将公共行为放在父类中； 可实现一种反向控制结构； 符合单一职责原则和开闭原则。	如果父类中可变的基本方法太多，将会导致类的个数增加（可结合桥接模式）	一次性实现一个算法的不变部分，将可变部分的行为留给子类来实现； 提取公共的行为到一个公共父类中，避免代码重复； 需要通过子类来决定父类算法中某个步骤是否执行，实现子类对父类的反向控制。		
迭代器模式 **Iterator Pattern** **游标模式** **Cursor Pattern**	提供一种方法，顺序访问一个聚合对象中各个元素，且不用暴力该对象的内部表示。 Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. 对象行为型模式	Iterator 抽象迭代器 Concrete Iterator 具体迭代器 Aggregate 抽象聚合类 Concrete Aggregate 具体聚合类	支持以不同的方式便利一个聚合对象，可以定义多种便利方式； 简化了聚合类； 增加新的聚合类和迭代器都很方便，符合开闭原则。	类的个数成对增加，增加了系统的复杂性； 抽象迭代器的设计难度较大，需要充分考虑到系统将来的发展。	访问一个聚合对象的内容而无需暴露它的内部表示； 需要为一个聚合对象提供多种遍历方式； 为遍历不同的聚合饥饿狗提供一个统一的接口，客户端可以一致性地操作该接口。		