# 图解设计模式读书笔记

设计模式是GoF the Gang of Four-四人帮的经验整理

我第一次看到GoF以为是什么专业术语，没想到是这样一个有趣的简称，四个人能互相交流共同进步学习的感觉一定很赞吧。

经典数字设计模式根据模式的使用功能范围分类为：创建型、结构型和行为型，但是按照这样的目录去讲解，对于初学者可能会比较生涩难懂，这本《图解设计模式》根据设计模式的复杂程度以及特点，由浅入深的对设计模式进行细致的讲解。

类图特点：上中下三个框分别表示类名、字段名和方法名，斜体代表抽象，下划线代表静态，可见性：public(+)、private(-)、protected(#)、默认(~)，各个类的关系有：聚合（菱形指针）、继承（实线三角）、实现接口（虚线三角）。

## 适应设计模式

### Iterator模式

一个一个遍历，JAVA集合中基本都实现的迭代器，这样使用这样的模式，可以拓展成为满足特定要求的迭代器。

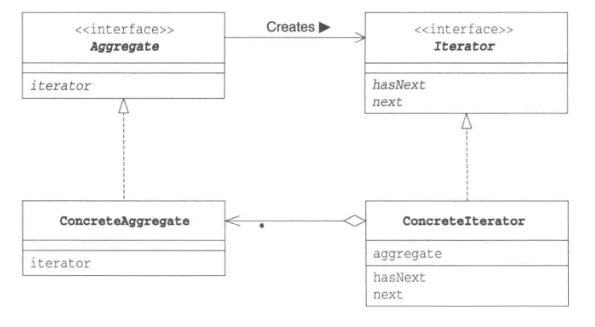


图1 迭代器模式

角色：

* Iterator(迭代器)：遍历集合元素接口，next和hasNext方法，获取下一个元素和是否有下一个元素；
* ConcreteIterator(具体的迭代器)：实现迭代器两个方法，保存集合大小下标等；
* Aggregate(集合)：由集合来定义创建迭代器，iterator生成这个集合的迭代器;
* ConcreteAggregate (具体集合)：实现集合，自定义的集合。

特点：

* 可以对外部隐藏实现：不管内部遍历怎么变化，对于外部来说只需两步：集合->得到迭代器，迭代器->获取下一个元素遍历，都可以使用迭代器；
* Visitor观察者模式是在迭代的基础上，在遍历的过程中对元素进行相同处理；

建议：

* 不要只使用具体类编程，要优先使用抽象类和接口编程。（只有在复杂的系统只改BUG的时候，就会发现这样的建议真的很有用）。

### Adapter模式

加个“适配器”以便于复用，用于填补现有的程序和已有程序的差异，通俗来讲，扮演中间调解人角色。其中适配器根据API由抽象类还是接口类定义分为两类：委托和继承。

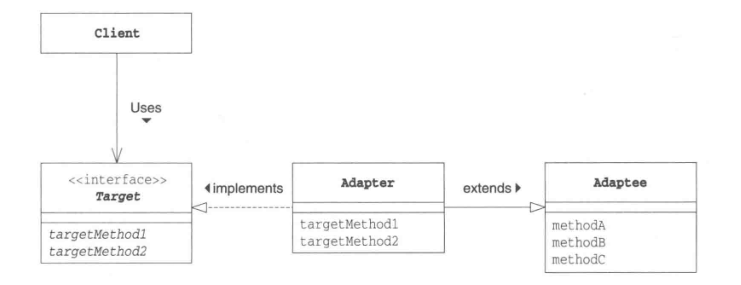


图2.1 使用继承的类适配器模式

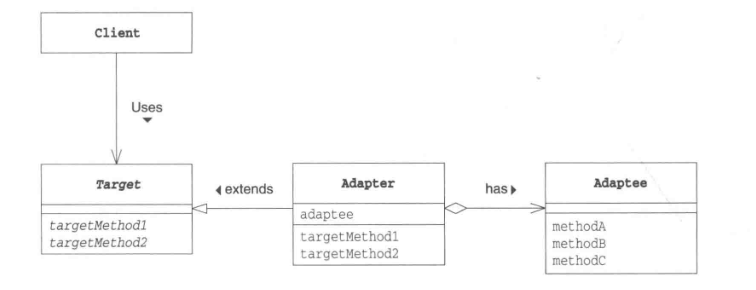


图2.2 使用委托的类适配器模式

角色：

* Target(对象)：定义所需的方法，定义输出内容格式1、2；
* Client(请求者)：对请求进行具体的处理；
* Adaptee(被适配)：持有既定方法，需要被Shape去适应新需求的对象；
* Adapter(适配器)：实现具体的API将不满足需求的对象Shape到满足使用条件；

特点：

* 一般这样的设计模式被使用到没有现成的代码，但是功能可以被提取出来的情况，如版本兼容与升级，如果不能被提取，就相当于从自来水管中提取出电肯定是不行的；
* 适配器模式用于填补不同的接口之间的缝隙，但是装饰者模式则是在不改变接口的条件下增加功能。

## 交给子类

### Template模式

模板模式，将具体处理交给子类，模板定义整个流程，但具体处理由子类决定，这样可以有一类产品呈现出多样性。统一流程是一套，怎么做下面的手下自己决定。

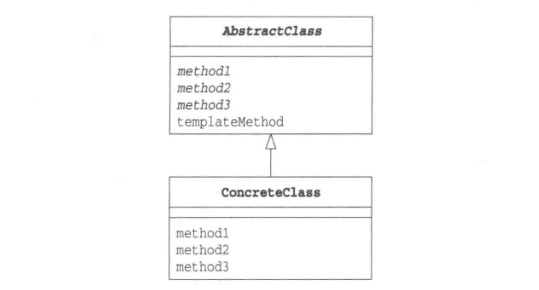


图3 模板方法模式

角色：

* AbstractClass(抽象类)：不负责实现模板方法，只负责声明；
* ConcreteClass(具体类)：实现预声明的方法，根据自己的要求处理。

特点：

* 模式讲究父类子类之间的协作、一致性，就体现在了父类管理、子类实现和父类一致的接口；
* 工厂模式就是模板模式的拓展，虚拟工厂就是模板父类，具体工厂就是实现的具体子类；
* 策略模式和模板模式类似，模板模式使用继承改变程序行为，但是策略模式使用接口委托来改变程序行为，并且策略模式改变可以更彻底

建议：

* 多使用模板模式使父类和子类协作，相同逻辑用一个父类统一定义，使逻辑处理通用化。

### Factory模式

将实例的生成交给子类，模板模式的拓展，虚拟工厂定义规范流程，具体工厂去具体实现，实现框架和实际生成解耦。

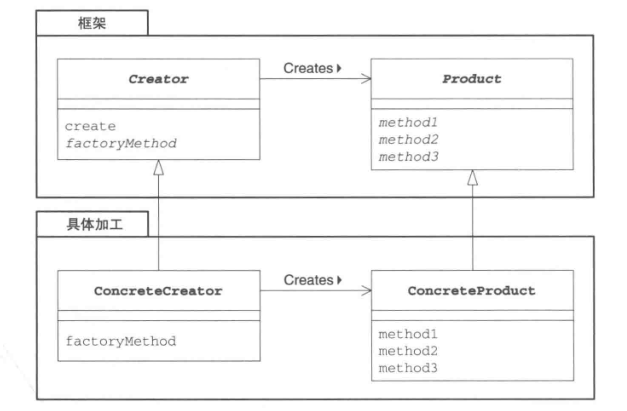


图4 工厂模式

角色：

* Product(产品)：框架中的概念产品，可以被继承定义为各种不同的产品，框架中使用解耦；
* Creator(创建者)：框架中的抽象工厂，不负责具体实现，只提供工厂的统一描述，抽象的东西可以使代码复用率变高；
* ConcreteProduct(具体产品)：决定具体的产品是什么样，实际产品；
* ConcreteCreator(具体创建者)：实现具体的加工方法，实际的工厂。

特点：

* 需要使用框架进行解耦，这样外部使用都不用了解具体实现，使用虚拟工厂的接口就可以得到不同的具体产品；
* 通常创建者（工厂）都会使用单例模式，同一个工厂实例可以重复调用；
* 迭代器模式中，在集合对象生成迭代器的时候，迭代器角色和具体的迭代器其实使用的就是工厂模式。

建议：

* 使用框架定义逻辑，具体实现交给子类，框架中不要使用具体子类可以解耦。

## 生成实例

### Singleton模式

在程序的生命周期内只有一个实例，并且能被重复使用，如数据集合对象，虚拟工厂对象等。例如，虚拟工厂可以满足生成不同的产品，不管何时，需要工厂生成一个产品就实例化一个工厂是非常占用空间且浪费的，所以需要重复利用一个实例。

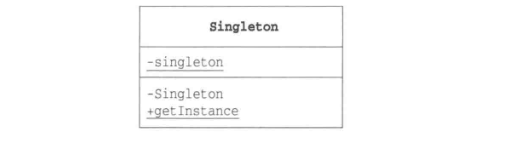


图5 单例模式

角色：单例角色。

特点：

* 私有的Constractor，保证对象的不被外界实例化，只能由自己控制；
* 根据何时生成这个对象，单例模式被分为：懒汉式、饿汉式，实现方式分为：静态变量、静态内部类、枚举等。

建议：

* synchronized 关键字修饰getInstance方法保证线程安全。

### Prototype模式

通过复制生成实例，根据实例原型来生成新的实例。不能指定类名生成新的实例，概念类似于克隆，用于类名过多无法整合难以生成实例时，这样的模式会非常管用。可以联想到Java Object中的clone方法，Object是原型，对象可以自己实现clone方法，可以对实例进行克隆。

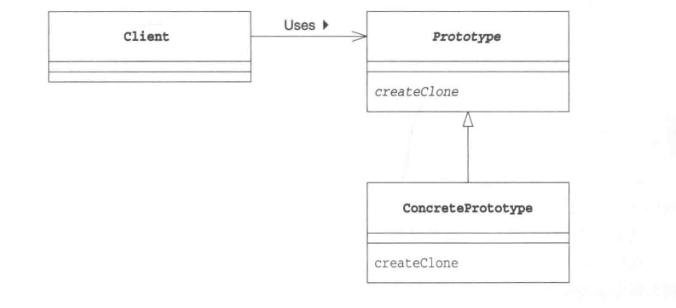


图6 原型模式

角色：

* Prototype(原型)：product角色定义用于复制现有实例来生成新实例的方法；
* ConcretePrototype(具体原型)：角色负责实现复制现有实例生成新实例的方法；
* Client(使用者)：使用实例，调用克隆方法根据原型复制实例对象。

特点：

* 原型模式根据当前实例生成状态相同的另一个实例，而FlyWeight模式可以在不同的地方使用同一个实例；
* 快照Memento模式，可以保存当前实例的状态实现快照和撤销功能；

建议：

* 在框架中，想对于具体的实现子类进行解耦时，就会使用克隆方法，是框架能通用，避免耦合。
* 一旦代码中出现要使用的类名，就紧密得耦合在一起，无法分离复用，这与目标“作为组件复用”背道而驰。
* Object可以被克隆是需要实现Cloneable接口，其实接口没有实现任何事，只是作为是否可以克隆的标记，这样的接口称为标记接口。克隆方法是浅复制，当使用数组指针时，只会复制指针值，还是会指到同一内存地址，容易造成错误，需要规避。

### Builder模式

组装复杂的实例，将各个不同的组件组装成一个复杂的结构。单个零件技巧学习后，开始学习盖楼，复杂的设计模式就此开始。其实构建模式和工厂模式很容易混淆，工厂着重点在于虚拟和具体的关系，构建着重于各个各个具体如何组装，在我的理解中，构建模式就是在工厂模式下，赋予虚拟工厂定义组装流程的功能。

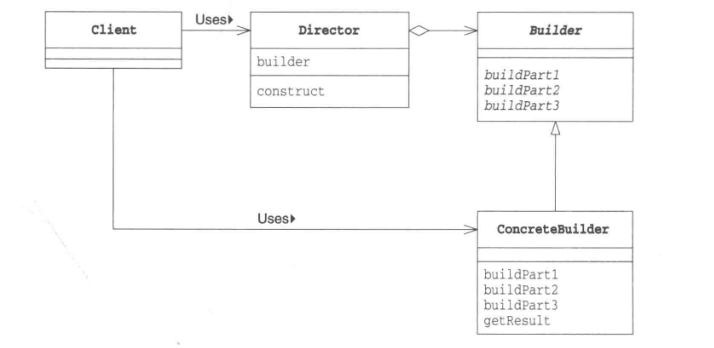


图7 构建模式

角色：

* Builder(建造者)：定义用于生成实例的接口；
* ConcreteBuilder(具体构建者)：生成实例的具体实现，并且定义获取最终生成结果的方法；
* Director(监工):使用Builder角色的接口来生成实例，并对构造后的成果进行统一的拼装，不依赖于具体的Builder角色；
* Client(使用者)：整个构建开始的调用过程。

特点：

* 构造模式中，监工角色使用聚合关系控制Builder角色；在模板模式中使用父类控制子类；
* 构造模式与抽象工厂模式都是用于生成复杂的实例，但是着重点不同，前者着重组件组装构建，后者强调将流程逻辑抽象出来；
* 构造模式通过组合Builder角色中的复杂方法向外部提供简单生成实例API；Facade模式中Facade角色则是通过组合内部模块向外部提供简单调用API。区别在于如何组装前者由调用者决定，Builder只负责生成实例；后者由内部决定，提供组装好的结果。

建议：

* 设计的时候要遵循知识最少原则，一个类只要了解并管理自己该管理的部分即可，父类只有在不知道子类的情况下，才会提高可替换性。

### Astract Factory模式

将关联零件组装成产品，将零件组装成产品的过程抽象化，和具体实现分离。其实也是抽象工厂的拓展。抽象工厂的工作是将“抽象零件”组装为“抽象产品”，外部不需要了解零件和组装的具体实现，只关心生成产品的API。就像客户只关心你的产品，不关心你如何去生产一样。

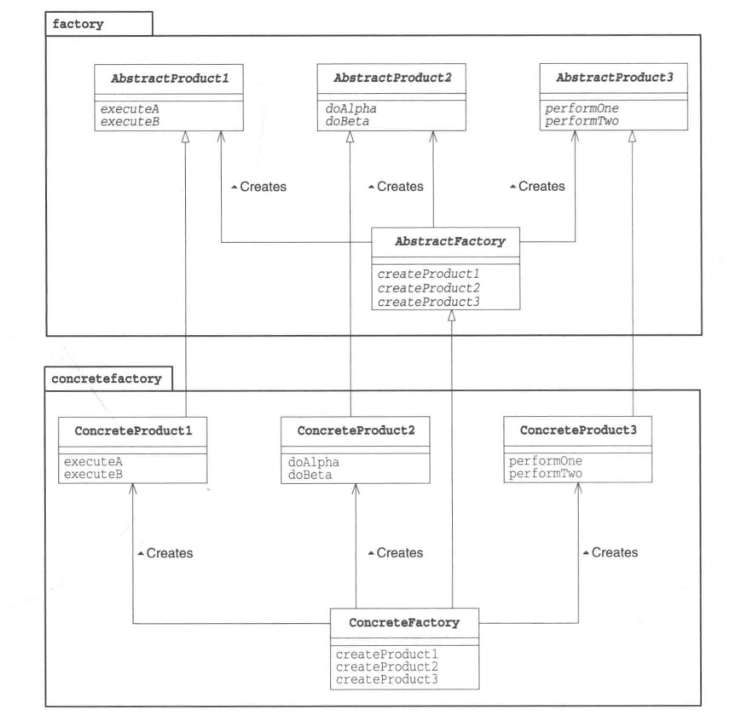


图8 抽象工厂模式

角色：

* AbstractFactory(抽象工厂)：利用抽象零件产品角色构建描述抽象产品；
* AbstractProduct(抽象产品)：概念产品没有实体，只有一个框架角色；
* Client(委托者):调用抽象的工厂产品角色，进行订单工作；
* ConcreteProduct(具体产品)：继承抽象产品，具体实现；
* ConcreteFactory(具体工厂)：抽象工厂的具体实现。

特点：

* 抽象工厂通过调用抽象产品API来组装抽象产品，易于增加具体的工厂，难以增加新的零件；
* 构建模式由外部分阶段制作复杂实例；

建议：

* 生成实例的基本方法：New、Clone、newInstance和反射机制。

以上是本书的前三部分，适应设计模式、交给子类和生成实例的相关的设计模式读书笔记，每个部分由浅入深的讲解了设计模式基本的设计结构，以及如何将这些设计结构组合运用到更复杂的设计中去，前两部分提出来一些基础的设计如工厂模式、模板模式、单例模式等，后面将这些基础的设计模式进行拓展，去构建更复杂的设计模式，如抽象工厂模式、原型模式、构建模式。设计模式看起来视乎对代码功能是否能实现并没有什么太大的作用，但是当要维护一个庞大的系统时，再去看整个系统，会发现有着良好架构的系统会更清晰并且更易维护，无论是修改还是拓展都会在修改量很小的条件下达到要求。简单的设计结构，系统只会越做越大，越做越复杂且难以维护。所以良好的架构真的很重要。

## 分开考虑

### Bridge模式

桥梁模式，将类的功能层次结构和类的实现层次结构连接起来。类的功能层级结构指的父类具有基本功能，子类中有增加的额外的新的功能；类的实现层次结构指的父类声明抽象方法定义接口，子类通过继承具体实现接口；一个在子类增加功能，一个是增加实现。（一个利用继承层级实现功能分层，一个利用抽象继承统一规范子类行为，学习这个模式区分这两个点很有必要）

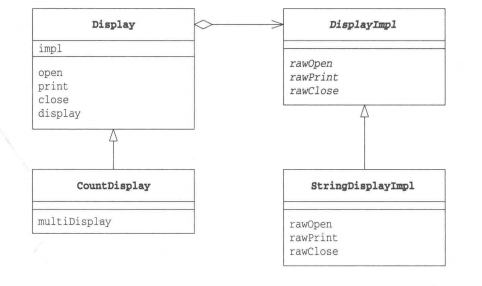


图9 桥接模式

角色（左功能层次，右实现层次结构）：

* Abstraction抽象化-Display类：有一个抽象的实现层的父类的实例，这个抽象类定义了display类的功能接口。Impl则是两个层次结构的桥梁，右边是类的实现层级结构，左边是功能层级结构。
* RefinedAbstraction改善后的抽象化-CountDisplay类：在display的基础上实现新的功能-显示规定次数，同时也继承了父类的功能
* Implement实现者-DisplayImpl类：抽象类，声明抽象方法，这几个方法是从显示功能抽象出来的底层方法，可以有不同的底层实现，作为参数传递给display即可使用。
* ConcreteImplemnet具体实现-StringDisplayImpl类：针对String对象具体的实现，如何打开打印并关闭，也可以有StreamDisplayImpl类等等。

特点：

* New \*\*DisplayImpl，传递给display构造器，实现\*\*的打印；传递给CountDisplay实现多次打印。\*\*可以是String也可以是其他对象，底层的实现脱离了这个功能实现框架。
* 功能实现框架，抽象出实现的功能方法，将具体的实现实例传递给功能框架作为域，功能调用抽象接口即可，实现不同的实现，同一个框架复用。
* 易拓展，具体的实现只需要继承抽象接口，即可传递给功能框架被使用。
* 功能结构层次将功能抽象出来委托给实现层级结构，委托为弱关联，功能结构层次不必强关联具体实现。

### Strategy模式

策略模式 是指整体部分替换算法的部分实现，解决同一个问题轻松的替换不同的算法。可以应用于不同的数据特点使用不同的数据计算或者查询的方法；内存换取高性能还是低性能换取低内存；下棋游戏人机策略好坏区分难度等级，等等应用。

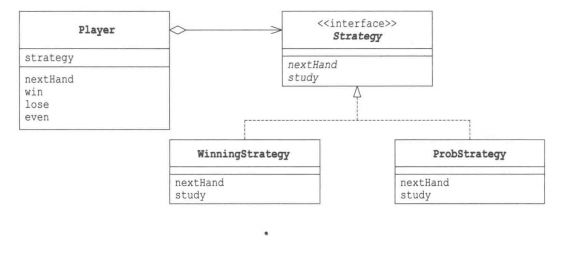


图10 strategy模式

角色：

* 策略-Strategy接口：nextHand计算下一局要出的手势，study接口方法供外部返回输赢结果状态，提供分析依据。
* 具体策略-WinningStrategy类：
* 上下文-Player类：选手，包含姓名和策略，nextHand调取接口获取策略结果。

特点：

* 委托，上下文将计算委托给策略接口，并返回结果给策略，参与到策略的计算中。策略是上下衔接到上下文中的，不似桥接模式。

## 一致性

### Composite模式

文件夹为例子，文件夹下面可以有文件也可以有子文件夹，子文件夹又可以有自己的文件和跟下级的子文件夹。这种树被统称为目录，文件和文件夹都被称作条目。在这个目录树中，文件和文件夹都被当做条目这一种东西存在，并且以递归结构层级包含。Composite（复合）模式就是用于这种结构，能够是容器与内容具有一致性，创造归纳出递归结构。

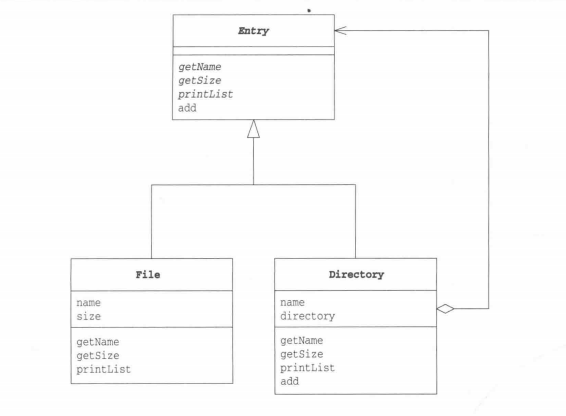


图11 复合模式

角色：

* Leaf树叶-File类：内容角色不能再作为容器放置其他对象，文件中不能包含另一个文件
* Composite复合物-Directory类：表示容器的角色，可以容纳条目，本身和leaf一样可以作为条目
* Component-Entry类：容器可以被放置的对象，是一种统一描述对象。
* Client使用者-Main类：使用这个复合的容器的对象。（可以再这里拓展功能，如名字查找，计算文件大小，统计文件数量等功能，所以单独列出来一个角色）

特点：

* 使用抽象继承实现 多个和单个的一致性，对个对象结合在一起，被当做一个对象进行处理，其实也可以被称作使用数据结构-表现为一个对象。
* 针对特定条目对象拓展功能时，只能在对应的条目子类中去拓展（指的新建条目只能在文件夹中进行这样的功能）
* Commond模式编写宏命令时，就使用了复合模式，Visitor模式可以用来访问递归模式。

### Decorator模式

蛋糕涂上奶油，变成奶油蛋糕，奶油蛋糕加上草莓就变成了草莓奶油蛋糕，再加上果酱写的生日快乐就成了生日蛋糕。不论是什么蛋糕，都是由其他东西装饰蛋糕而来，装饰使得蛋糕更加明确，贴近食用者的需求。这样的不断给核心的东西添加装饰，满足不同的需求的设计模式就叫装饰模式。

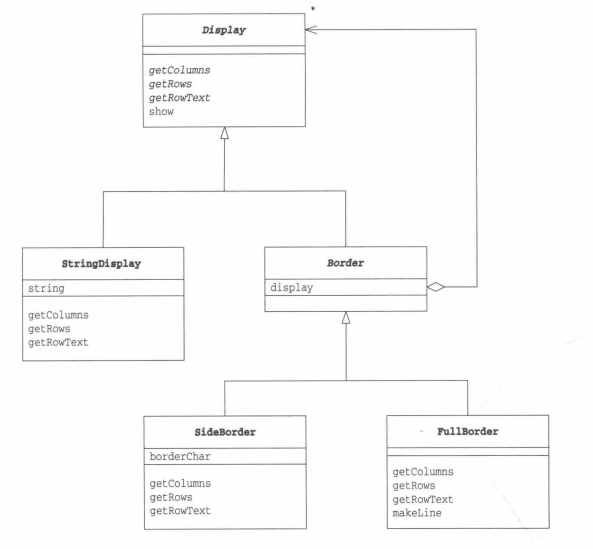


图12 装饰模式

角色：

* Component组件-Display类：具体组件抽象出来的特点，装饰模式需要用的方法，比如边框需要知道内容有多少行多少列，边框装饰才能适应内容。
* ConcreteComponent具体组件-StringDisplay类：实现具体的特点。
* Decorator装饰物-Border类：装饰的抽象类,含有一个被装饰物的实例。
* ConcreteDecorator具体装饰物-SideBorder/FullBorder类：具体装饰物，定义不同的草莓奶油果酱抹在蛋糕上。

特点：

* 抽象父类，使用的就是模板模式;
* Border类继承Display，通过继承，装饰边框于被装饰物具有了相同的方法，装饰物和被装饰物有了一致性。可以层层套用，层层装饰。抹了奶油的蛋糕上再放草莓，草莓奶油蛋糕上再放果酱。形成了类似复合模式的递归结构，不过装饰模式主要目的是增加对象的功能。
* 不修改被装饰的类即可增加功能。
* 导致会增加许多小的类

建议

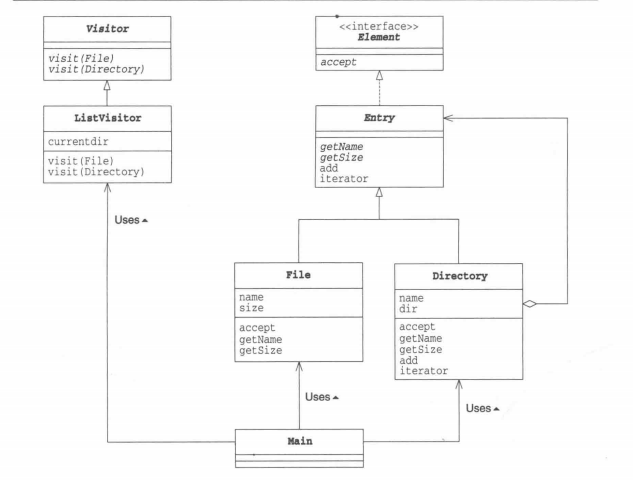
* Java io包中，new BufferReader(new FileReader(“\*.txt”)),读取文件流到buffer中，我们也可以拓展一个LineNumberReader(new BufferReader…)拓展行号管理的功能；不读取文件的数据，获得文件内容的行号；
* Javax.swing.border包中也使用了装饰模式，也可以拓展自己的装饰类，自定义边框。
* 巧用继承和委托。

## 访问数据结构

### Visitor模式

数据处理与数据结构分开，访问者独立于数据结构之外，作为一个观察者存在，每增加新的数据处理的时候，不用修改数据结构，只需要增加一个观察者即可。

下面的例子使用的数据结构为复合模式中的文件夹结构。



角色：

* Visitor访问者-Visitor类访问者的抽象类，visit方法重载接受不同的对象，分别处理。处理依赖于访问的数据结构。
* ConcreteVisitor具体访问者-ListVisitor类：访问数据结构并显示一览
* Element元素-Element接口：接受访问者的访问接口。
* ConcreteElement具体元素-Entry/File/Directo类
* ObjectStructure对象结构- Director类

特点

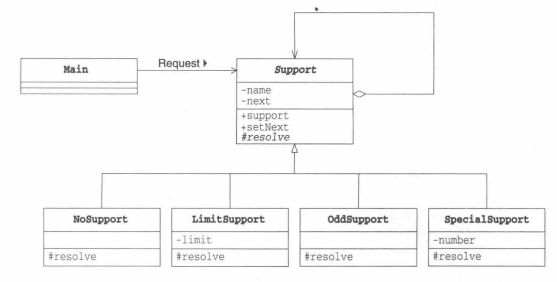
* Accept方法开一个入口，给不同的被访问者询问回调访问者的访问方法，重载保证了对不同被访问者的兼容。Accept方法和visit方法之间相互递归调用。
* 将处理从数据结构中分离出来，保留数据结构作为组件的独立性。
* 易于增加具体访问者角色，难以增加具体被访问者角色
* 迭代器模式趋向于访问数据结构中的元素，访问者趋向于对元素进行特殊处理

建议

* 开闭原则-对扩展开发，对修改关闭。可以使在不修改现有代码的前提下进行扩展。

### Chain Of Responsibility模式

一个请求可能会分布在不同的处理器处理，外部不知道处理器内部的分类，只能由内部的处理器逐个检查是否是自己可以处理的请求，不是就把请求转交给下个处理器。这样的多个处理器对象组成一条职责链，然后按照职责链的顺序，逐个找出到底应该谁来负责处理，这种模式被称作责任链模式。这种模式的好处就是弱化请求和处理方的关联关系，各自都是可独立复用的组件。（还有种办法就是做一个请求分配处理器，专门负责分配请求任务）



角色

* 处理者-Support类：响应请求，解决问题的抽象类。Resolve没有解决，找到next处理者处理，直到解决。
* 具体处理者-odd/No/Limit/SpecialSupport：实际处理问题的类，处理特定问题
* 请求者-Main:生成问题对象，请求处理。

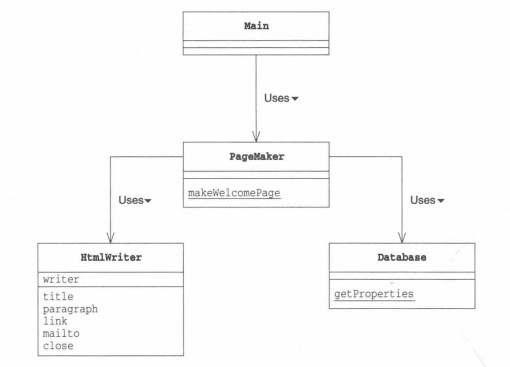
特点

* 弱化请求者和处理者之间的联系，请求者不必区分我的请求该由谁处理，统一了请求入口
* 可以动态改变职责链，某个处理者下班或者故障，就可以把问题分配给其他的处理者帮忙处理，或者有新的处理者参与处理，改变处理者序列即可，无论怎么变请求者都不会受到干扰。
* 职责链模式使得个类型的处理者专注于处理自己的工作。
* 每次请求可能会被很多处理者检查是否可以处理，这样会造成处理延迟，因为请求到达对应的处理类可能需要被检查很多次。
* Resolve为什么是protected方法？因为不希望请求直接到某个具体处理者。

## 简单化

### Façade模式

Façade本意为建筑的正面，facade模式又称外观模式,,隐藏内部的内容，只提供一个前台给来访问的人，通过前台连接后台服务，这样将错综复杂的API对外简化为一个简单的高层接口的模式即为外观模式。



角色

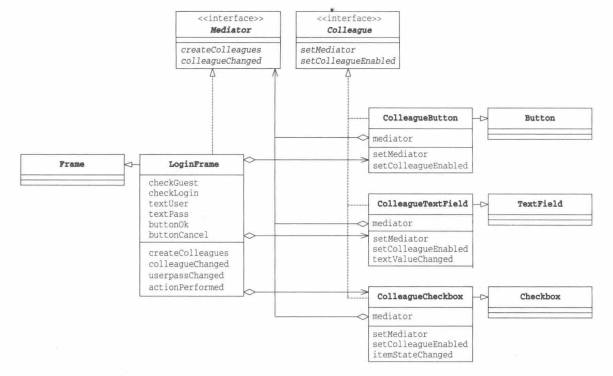
* Façade窗口-Pagemaker :整合下面俩个类，对外只提供一个makeWelcomePage接口
* 系统其它角色-HtmlWriter/Database：提供构成系统的功能
* 请求者-Main：使用MakeWelcomePage接口，不用调用系统的其它角色

特点

* 复杂的东西对外变简单，接口内容变少，内部和外部关系弱化，更容易被复用。
* 过多依赖的关系，前后步骤有关联、顺序要求，都需要用外观模式。

### Mediator模式

Mediator模式，又称仲裁者模式，组员向仲裁者报告仲裁者向组员下达指示。相对于外观模式和责任链模式，这个仲裁者具有管理协调内部下属类的功能，下属和仲裁者之间有个双下的沟通。



角色

* 仲裁者-Mediator接口：声明管理组员和组员报告仲裁者的两个方法接口。
* 具体的仲裁者LoginFrame：实现具体的仲裁策略，统一反馈给使用者。
* 同事Colleague接口：声明组员向仲裁者进行报告和接受仲裁者下达的指令的接口
* 具体同事-Button等，都是组员。实现Colleague接口

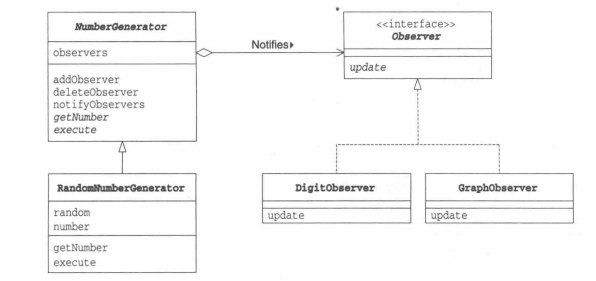
特点

* 统一有一个管理者，把管理脱离于对象组合之外。其实模式改名为管理者跟贴切一些，管理者逻辑控制。（修改内容可以用管理模式管理，只读取修改的内容更新进数据库，可以节省下不必要的数据库内容更新）
* 分而治之，frame管理组件，组件中的逻辑由仲裁者单独管理，便于处理。
* 对于逻辑不同的仲裁，具体仲裁者不能复用。

## 管理状态

### Observer模式

观察者模式，内部发生变化时，通知给观察者。适用于根据对象进行相应的处理场景。单向沟通的关系意味更明显。



角色

* 被观察对象Numbergenerator:管理包含观察者实例，在状态变化时，调用观察者的通知接口
* 具体观察对象RandNumberGenerator：具体实现。
* 观察者Observer:声明观察者接受通知的接口。
* 具体观察者Digit/GraphObserver:接受到通知具体的处理。

特点

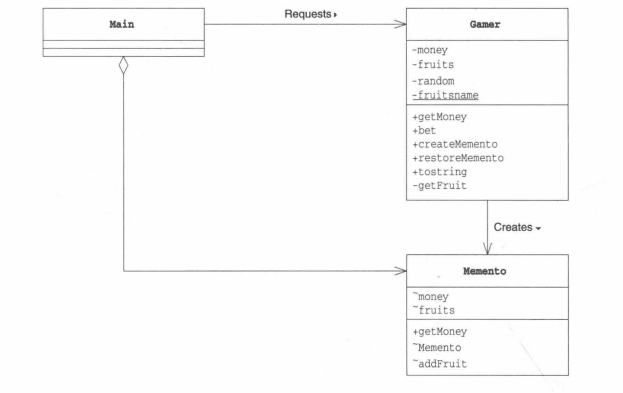
* 使用抽象类或接口对象保存在关联的对象中，而不是具体子类，提高可替换性。
* 观察者在改变观察对象状态时要注意，不要陷入死循环。
* 观察者不是主动在观察，而是把自己实例交给对象，等待对象的通知。
* 仲裁模式通知是为了仲裁，观察者模式是为了两者状态同步。观察者更趋向于内外部沟通，仲裁者模式趋向于内部上下级管理沟通。

拓展

* Java.util.Observer接口和Observable类，观察对象是一个类，单继承，导致具体观察对象不能继承其他的类，所以util包的东西不能套用。

### Memento模式

备忘录模式，在保护对象封装性的前提下，保存修改状态，并支持恢复实例状态。



角色

* 生成者Gamer:最新状态保存在纪念品中，提供一个接口接受纪念品，恢复状态。
* 纪念品Memeto:纪念品，内部信息的集合，提供宽接口给生成者，窄接口给外部避免泄漏
* Caretaker负责人main:负责管理什么时候生成纪念品什么时候恢复纪念品。虽然能得到纪念品对象，但只能访问窄接口，保护了内部信息。

特点

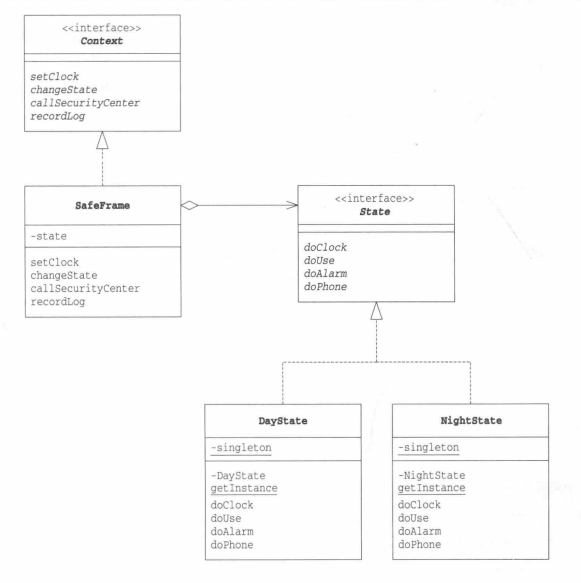
* 纪念品需要额外增加管理，不能无限增长，所以需要提供一个有限期
* 负责人和生成者是根据职责分担，分离两个角色。生成者和纪念品就可以成对被复用。

拓展

Java.io.ObjectOutputStream/InputStream ObjectOutput/Input

### State模式

状态模式，将对象实体本身和状态分开考虑，以单独的类来表示状态，通过切换类来方便修改对象的状态。不管是增加状态还是删除状态，管理状态就会很方便。



角色

* 状态接口-State接口：表示状态的接口，抽象状态不同结果不同的方法
* 具体状态-DayState/NightState:两个具体的状态,实现状态接口的方法
* 上下文-Context接口：负责管理状态，联系外部的接口。
* 对象SafeFrame：状态描述的对象。

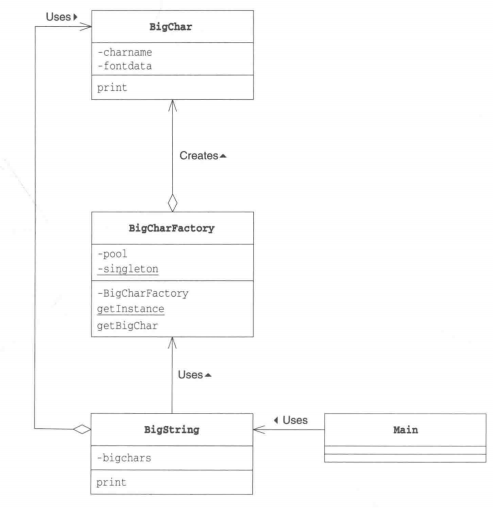
特点

* 分而治之，状态抽象为对象，独立于对象之外，状态相关的操作都被抽象为方法，并定义在一个接口中，增加删除状态不用修改对象。
* 注意不能存在多个状态对象避免自相矛盾
* 实例的多面性，SafeFrame当做一个对象的同时，也是一个ActionListener，同时也是一个Context接口

## 避免浪费（共享对象）

### Flyweight模式

轻量级模式，让对象变轻，让对象使用的内存变小，通过尽量共享实例来避免new出不必要的实例。



角色

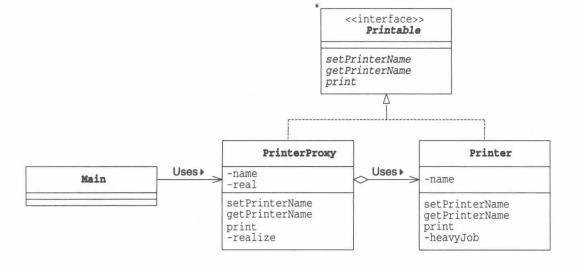
* Flyweight-BigChar类：单个字符实例对象。
* FlyweightFactory-BigCharFactory类：共享BigChar实例的工厂，管理固定BigChar集合
* Client-BigString类：使用工厂组成自己的实例。比如数字12就可以被BigChar1+BigChar2表示，21也可以用这两个实例表示这样实现了单个字符实例的重用。

特点

* 共享的致命弱点是，修改也可能被同步到所有使用的地方，一个地方修改对多个地方产生影响。
* Intrinsic和Extrinsic，应当共享和不应该共享的信息，这两类信息需要在设计这个模式的时候加以区分，应当共享的信息应该是静态的，不随状态变化而变化的信息，否则不能共享。
* 注意不要让共享的实例被GC。

### Proxy模式

代理模式，是增加一个代理对象分担业务压，代理人代替本人进行工作，遇到无法解决的事情再寻求本人帮助，这里的本人和代理人都是指的对象。



角色

* 主体-Printable接口：代理人和本体一致性的接口
* 代理人-printerProxy：代理人，复制部分本人的属性，保存一个本体对象，避免不能解决的时候向本体求助。
* 实际主体-Printer：本体，负责所有基本业务，重活。
* 请求者-main：接口使用者。

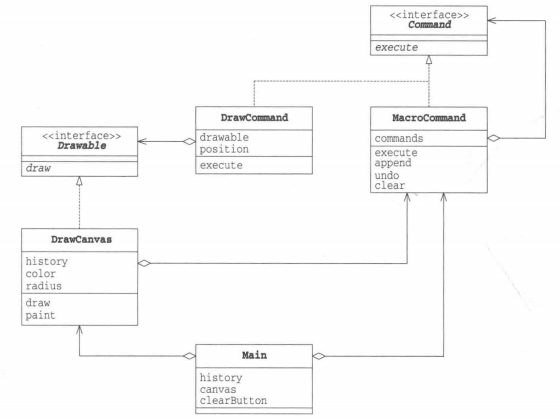
特点

* 使用代理人可以提升处理速度，因为没有必要的时候，可以不用初始化本体。
* 代理是本体将部分业务交给代理人解决，代理无法解决时也是会转交给本体处理。
* 代理的方法需要synchronized,避免代理对象发生混乱。
* http使用的缓存使用的就是代理模式，缓存不能找到客户请求时，再交给服务器处理。
* 代理分为虚拟代理（上面的例子）、远程代理（javaRMI）、访问代理（访问拦截器）三类。

## 用类来表现

### Command模式

命令也可以被当做一个类，使用一个类来保存命令的内容并保存结果，这样的好处就是多次调用时，只需要使用这个类，不需要再次执行。将命令物化，这样的模式称之为命令模式。（让我联想到了excel的宏命令，将录制的命令保存为宏，可以多次重复执行）



角色

* 命令command接口：表示命令的接口
* 具体命令DrawCommand：具体的命令实例，执行具体命令。
* 接受者Drawable/DrawCanvas；接受命令,接受命令对象执行
* 请求者main类：使用命令发动者和接受者。
* 发动者MacroCommand：表示一个开始执行命令的入口对象，发起执行命令的命令

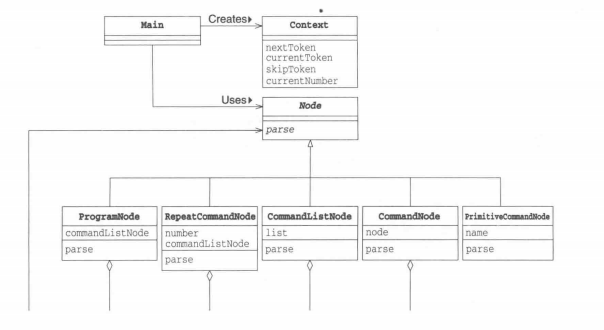
特点

命令发动者可以保存命令对象，并可以保存文件中永久。

将命令当做对象，保存在列表中，保存操作的历史，支持撤回等功能。

### Interpreter模式

解释器模式，就像java语言和java解释器一样，无论java代码写的什么内容，解释器都不用改变，只需要按照特定的步骤解析可以，这样的好处就是，解释器可以重用，不用写一个程序改一下运行环境。[[1]](#footnote-1)



角色：

抽象表达式Node抽象的所有命令对象

终结符表达式primitiveCommandNode/CommandNode：终结对象，就是单行命令，不是一个集合。

非终结符表达式ProgramNode/RepeatCommandNode：非终结对象，就是多行命令，大括号，循环内可以包含多行命令，需要一个包含多个单行命令的集合

上下文context：运行环境

请求者：main：请求解释并运行

特点

迷你语言-都有一个解释器，还有一些比较常见的迷你语言，如正则表达式、检索表达式、批处理表达式。

1. 文中提到BNF 巴特范式，描述语言发的通用规则 [↑](#footnote-ref-1)