设计模式测试文档——天天农场

张　喆 1754060

卜　滴 1753414

陈开昕 1753188

刘　雨 1752461

叶一凡 1752580

吕雪飞 1752289

刘一默 1752339

罗　浩 1752547

张子健 1752894

李一珉 1752882

完成日期 2019-11-04

设计模式测试说明

1 Abstract Factory

1.1 测试逻辑

抽象工厂模式说的是当创建一整族相关的对象是，不需要点名对戏那个真正所属的具体类，而是为访问者提供一个创建一族对象的接口。

在该测试中，使用产品工厂基类的对象，而让用户自定义产品工厂的类型（1 春季农产品；2 夏季农产品），用户可以自由切换工厂的类型，而且工厂仓库中保存了所有工厂生产的产品。

对于产品工厂自身而言，基类产品工厂生产蔬菜和水果，同样由用户（二次开发的程序员）指定具体的蔬菜和水果。这里春季产品工厂生产苹果和土豆；夏季工厂生产樱桃和番茄。

如果需要添加秋季产品工厂，只需新增AutumnProductFactory，在产品工厂内部指定生产的具体蔬菜和水果；在使用位置用基类引用指向秋季产品工厂即可。不需要影响原始结构，符合OCP原则。

1.2 测试用例

1. 用户选择产品工厂（1 春季农产品； 2 夏季农产品）
    - 春季生产苹果和土豆
    - 夏季生产樱桃和番茄
2. 相应的工厂生产相应的水果和蔬菜，并加入到产品工厂仓库中
3. 下一次选择时
    - 如果仍为相同的工厂，则继续生产
    - 如果更换工厂，则将现有水果蔬菜加入新工厂的仓库中，开始生产新水果和蔬菜

2 Adapter

2.1 测试逻辑

适配器模式是将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。

在该测试中，使用适配器Adapter的对象来实现接口看门狗Watchdog，用户可以直接通过接口Watchdog来执行dog类有的eat, sleep操作。

2.2 测试用例

1. 用户创建一个Dog类的对象
2. 用户用该Dog类对象创建并初始化Watchdog的实现类Adapter的对象watchdog
3. 用户通过watchdog来执行eat, sleep等从Dog类中抽取的操作
4. 用户通过watchdog来执行bark等在Adapter内部实现的操作bark

3 AOP

3.1 测试逻辑

AOP 是为了将一个系统中的各个模块的共有部分抽取出来，将这些功能实现为对象，从而实现集中的管理与复用。在我们的例子中，我们就成功的演示了在不修改原有的类（原有的类对此改动是无知的）的情况下，拓展一些方法的可能性。

我们将 qualityAssuranceTeam 以及 supplyTeam 这两个普适性的类和 BreedSceneBuilder 以及 WarehouseBuilder 这样针对某个特定功能的类进行了交叉，展现出较好的可拓展性。

3.2 测试用例

通过创建层层嵌套的类（QA_Breed、Supply_Warehouse 以及 QA_Supply_Warehouse）来向用户展示一种在不修改原有类的基础上进行拓展的可能性。

4 Bridge

4.1 测试逻辑

桥接器模式通过将可变维度抽象，并与实现类解耦，使它们可以独立变化，用组合关系代替继承关系

在该测试中，使用颜色基类的对象定义两种颜色：green 和 yellow， 分别用他们初始化豌豆基类的子类：圆粒豌豆SmoothPea 和皱粒豌豆 WrinkledPea，并输出他们的全名：x色x种豌豆。

如果需要添加新的颜色和豌豆种类，也只分别写继承自颜色基类的新颜色子类和继承自豌豆基类的豌豆子类，实现了颜色和豌豆种类的解耦，不需要修改原有类，符合OCP原则。

4.2 测试用例

1. 分别创建两个颜色基类的对象green 和 yellow
2. 分别用不同颜色初始化两种豌豆子类，共4个豌豆子类对象，即用green初始化的圆粒豌豆和皱粒豌豆及用yellow初始化的圆粒豌豆和皱粒豌豆
3. 分别输出这4个对象的全名

5 Builder

5.1 测试逻辑

Builder 模式是为了解决在初始化一个具有多个属性的复杂对象的过程中可以出现的问题而提出的设计模式。我们可以看到，如果没有 Builder 模式，那么在我们初始化一个具有多个属性的复杂对象的过程中就会不得不使用一个巨大的构造函数或者是将构造的逻辑分布在各个不同的类中。

而在实现了 Builder 模式之后，我们的创建过程就变得更加简单了。在我们的测试中，用户既可以通过内置于 ShortTermWorker 各个子类的 buildSomeThingNormal() 函数中的 Director 来实现简单的对象生成，也能通过直接调用类似于 breedSceneBuilderImpl 中的各类 set 函数来完成对象生产工作，这确实是 Builder 模式的标准体现。

5.2 测试用例

通过 BreedSceneBuilder、HouseBuilder 以及 WarehouseBuilder 中的 buildSomeThingNormal 函数中内置的 Director 类来实现一种普遍情况的创建。

通过邀请用户设置建筑的种类和位置来直接使用 BreedSceneBuilderImpl 等类中的各个 set 函数。

6 ChainOfResponsibility

6.1 测试逻辑

在责任链模式中，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，知道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，这使得系统可以不影响客户端的情况下动态的重新组织和分配责任。

在test中，我们创建出一个由Shower组成的链条，链的每个节点对应一种ShowerPattern，并由用户指定该Pattern适用的动物种类以及链条的连接方式。之后随意给出一个或多个待处理需求（有一种动物需要进行Shower行为），就能通过指定动物的种类和响应Shower的节点来判断链条的功能是否合理。

6.2 测试用例

用户选择操作（1 新建Shower节点 | 2 链接两个节点 | 3 查询动物的ShowerPattern）

- 新建节点
    - 设置节点编号
    - 设置节点的适用动物信息
- 链接节点——通过给定前驱节点和后继节点的编号，将两个节点按顺序链接起来
- 查询——通过给定动物的名称，通过每个Shower节点中存储的适用动物信息判断是否满足待处理需求。一个节点若满足，则进行处理（输出某动物用某种方式清洁）；若不满足，则往下传递；若无节点响应，输出该动物此刻不能进行清洗的信息。

7 Command

7.1 测试逻辑

Command 设计模式是为了解决各种界面所接受的用户操作的过程中所产生的大量重复问题而实现的一种将一种请求转化为单独的对象的设计模式。这种转化能够让我们复用命令、延迟调用并且实现撤销操作。在我们的例子中，我们的农场主将命令发给了ResidenceAdministrator 以及 ResourceAdministrator，从而使得它们能在合适的时候执行被给予的命令（首先是定时的任务，而后是邀请用户来要求它们执行指令），从而完整的体现了command 设计模式的优越性。

7.2 测试用例

1. 通过 lambda 表达式创建一些 command 的包装类（并无设计模式上的特殊意义，只是Java 语法的限制）
2. 通过 sleep 函数来在一些时限之后（每1s）执行一些记录日志的命令，然后打印出来当时的日志。而后执行两次 undo 指令，从而测试撤销功能。
3. 邀请用户来输入指令，我们进行执行。

8 Composite

8.1 测试逻辑

组合模式通过将对象组合成树形结构以表示"部分-整体"的层次结构，使得用户对单个对象和组合对象的使用具有一致性。整个测试主要在于，结构是否满足树型结构的部分-整体的要求，对象的使用是否具有一致性。

在测试中，创建许多的Succulent的节点，并且把他们放置到树型结构中（将叶节点add进根节点中）。通过最终按结构的输出来判断是否满足树型结构以及是否有一致的使用方法（显示名字）。

8.2 测试用例

用户选择操作（1 新建Shower节点 | 2 添加叶节点 | 3 输出结构）

- 新建节点——设置节点（多肉植物）名称
- 链接节点——通过给定根节点和叶节点，将叶节点add进根节点的目录中
- 输出结构——通过给定根节点，输出以此节点为根节点之下的全部叶子节点信息。

9 COW_ReferenceCount_Sharable

9.1 测试逻辑

写入时复制是说如果有多个调用者同时请求相同资源，他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源的内容时，系统才会真正复制一份专用副本给该调用者。

引用计数是说将对象的引用次数记录下来，当引用次数变为0时就自动释放。

共享体现在写时复制的过程中。

在我们的场景中，相同的资源是说一间宿舍的床、桌子和椅子。在创建一栋宿舍楼的过程中需要创建大量宿舍（我们这里简化为10间宿舍），当新建时所有宿舍的家具完全是可以共享的，直到它们被修改（我们这里简化为移动家具）时我们才需要为该宿舍新创建一个备份然后再修改。这样可以加快一间宿舍的新建速度，而不需要一开始就新建大量对象。

每次新建一个家具对象的备份时，该家具对象的引用计数应该减去一，减到0时我们就释放该对象（虽然Jvm自己会处理）。

然后为某间宿舍新建一个家具对象的备份后，下一次再修改该家具对象就没必要再创建一个新的备份了，所以我们把它的Sharable属性修改为false，这样宿舍修改家具对象前就不会创建备份。

9.2 测试用例

1. 新建一间宿舍，这时会打印出新建每件宿舍的信息，包括该宿舍的家具是新建的还是共享的。
2. 选择第一间宿舍，移动这间宿舍的床，这时应该会产生一个备份。
3. 再次移动这张床，这时就不会再产生备份了。
4. 把所有宿舍的床都移动一下，这时原始的床对象应该被释放了。

10 Decorator

10.1 测试逻辑

装饰模式将开放封闭原则发挥到了极致。在最初的设计中，我们默认每种住宅都有固定的价格，而在后续开发过程中，我们给住宅添加了一些附加功能，但是住宅类本身的设计并没有任何改变。

在DecoratorTest测试类中，我们首先声明一个不加任何附加功能的住宅，此时可以查看他的价格。之后我们可以给这个住宅添加各种附加功能，此时仍然可以运用之前的函数调用方法查看这个住宅的价格。通过测试我们可以发现，我们给系统增加新功能时，既没有改变原有的类的设计，也没有改变的类的使用方法，这确实满足了开放封闭原则。

10.2 测试用例

1. 用户选择要新建的建筑（1 接待中心； 2 豪宅）
2. 系统新建一个建筑
3. 用户选择以下操作（1 增加一个红外报警器 | 2 增加一个中央空调 | 3 增加一个监视器 | 4 显示建筑当前造价 | 5 退出）【注】在用户选择退出之前，步骤3将重复执行多次。

建议测试方法：先创一个新建筑，并输入其造价，再给建筑添加附加功能，并再次输出造价。

11 Facade

11.1 测试逻辑

住宅管理员（ResidenceAdministrator）的工作非常复杂，他需要在早晨叫醒员工宿舍中的所有员工，在晚上关掉员工宿舍的灯，在白天整理员工宿舍和接待中心的内务，以及运走员工宿舍、接待中心和豪宅的垃圾。

但是当你在FacadeTest测试类中调用住宅管理员的工作函数时，你会发现调用过程非常简单，只需调用wakeUp(), lightOff(), sweep(), takeTrash()这四个函数就可以了。如果不使用外观模式，客户端将遍历所有住宅，甚至遍历住宅中的所有人，将住宅里的人一个一个叫醒，或是将住宅的垃圾挨个运走，这将非常繁琐。使用外观模式后，住宅管理员的工作被封装了起来，简化了调用过程，降低了客户端与住宅类的耦合度。

11.2 测试用例

1. 系统自动创建好一个住宅管理员，他将负责两个宿舍，两个接待中心，两个豪宅的工作。
2. 客户端选择住宅管理员要执行的工作[1 叫床 | 2 打扫内务 | 3 运走垃圾 | 4 关灯 | 5 退出]
   【注】在客户端选择退出之前，步骤2可重复执行。

12 Factory

12.1 测试逻辑

工厂方法模式是说通过定义工厂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的对象。

在测试中，工厂接口LongTermWorkerFactory定义了创建长工类LongTermWorker的函数，而具体实现由工厂接口的子类去完成。我们目前为三种长工Repairman、Guard、Buyer创建了工厂类，通过类型转换将相应长工工厂产生的长工对象转化为相应的长工对象，在创建时打印相应的信息。

这里每当要新添加一个长工类型时，就要创建该长工类型的LongTermWorker的子类和生产该长工类型的LongTermWorkerFactory的子类，它们必须成对产生，关系是新类型长工的工厂类去创建长工对象并通过类型转化为该类型长工类对象。如果要添加新的长工类型，只需新增长工和工厂接口的子类，不需要影响原始结构，符合OCP原则。

12.2 测试用例

依次创建Repairman、Guard、Buyer的工厂创建一个对应的对象

13 Flyweight

13.1 测试逻辑

在FlyweightTest类执行测试的过程中，每当一个土地块实例被创建时，控制台都将输出“创建一个**Tract实例”。一个farmland有12个土地块，按常理来说应该会有12个土地块实例被创建，且在以后改变土地块类型时还会有新的土地块实例被创建。但正是因为使用了Flyweight设计模式，你可以发现在控制台上最多只有三句“创建一个**Tract实例”被输出，其原因是我们的土地块类型只有三种，而每当需要相同类型的土地块时，系统不会新建一个土地块实例，而只是返回已被创建的同类型土地块实例的引用。

13.2 测试用例

1. 系统自动创建好一个farmland实例，其由12个土地块构成，大小为3*4的矩阵，默认所有土地块块均为荒地。此时控制台会输出“新建一个荒地实例”，注意，这句话仅被输出一次而不是12次。
2. 系统输出farmland的俯视图。
3. 客户端选择操作[1 设置一个土地块 | 2 查看farmland俯视图 | 3 退出] 如果客户端不选择退出，步骤3可重复执行。在重复执行的过程中，你可以发现对于每一种类型的土地块，在控制台中只会输出一次“创建一个**Tract实例”。

14 Interpreter

14.1 测试逻辑

Interpreter设计模式所强调的是对于会多次出现的、有一定规律性的问题通过为其设置语法，将其翻译为一种语言的方式来解决。

可以用编译原理中的语法树来理解这个设计模式的实现，TerminalExpression类即为终结符，AndExpression即为非终结符。我们通过判断一个句子是否是合法终结符与合法非终结符的组合来判断其是否符合我们的要求。

AndExpression中会包含两个终结符成员，分别记录使用者和工具，然后通过interpret方法分别调用这两个终结符成员的interpret方法，来判断使用者和工具的组合是否成立。

14.2 测试用例

1. 通过四个while循环，分别向低权限使用者、低权限物品、高权限使用者、高权限物品中添加元素。通过输入“停止”来停止输入。
2. 输入完毕，开始测试，通过输入“某人使用某物”来测试设计模式是否实现良好。

15 IOC

15.1 测试逻辑

IOC 模式是为了解决在创建有相互依赖关系的对象的过程中，大对象依赖于小对象的情况（一个小的类的修改会引发所有相互关联的类的改动）。IOC 模式由两个部分组成，一方面是通过 IOC 容器实现对象的创建，另一方面是将创建好的对象注射到有依赖关系到类中去。限于时间与能力，我们只实现了第一部分，作为一种概念的演示。

我们的 test 中就演示了从 XML 文件中创建 IOC 容器的过程。

15.2 测试用例

1. 通过获取定义在 XML 文件中的内容，来向用户展示我们构造的相应的类。
2. 邀请用户自己去定义一些对象，进行实验。

16 Iterator

16.1 测试逻辑

所有实现了接口Aggregate的类都会实现方法getIterator(),返回一个Iterator作为外部遍历内部元素所需要的迭代器，返回的Iterator会复制Aggregate的内部元素以供遍历。

子类在实现Iterator接口时，可通过不同的实现方式来进行不同方式的遍历，让first()指向第一个，next()指向下一个则是正常的遍历。如果想要根据设定的条件来遍历，则让first()指向满足条件的第一个，而让next()指向满足条件的下一个。

16.2 测试用例

1. 向supplyAggregate中添加三个元素。
2. 通过supplyAggregate的getIterator()方法获取迭代器。
3. 使用supplyStorageIterator的first()和next()方法访问supplyAggregate中的所有元素。

17 Mediator

17.1 测试逻辑

Mediator解耦多个同事对象之间的复杂交互关系。创建中介者，每个同事对象都用与中介者的交互来代替原本同事对象之间的交互。反过来client则可以通过中介者统一管理所有对象。其优点为解耦多个相似对象之间的复杂交互，从而可以独立的改变他们之间的交互逻辑，从而降低了类结构的复杂度，将多对多模式转化为多对一模式。但是原本的同事对象之间交互越复杂，中介者的逻辑就会越复杂。

在该测试中，中介者模式的运用体现在植物自然传粉这一行为体系中，植物群体之间的雌蕊（Pistil类）与雄蕊（Stamen类）将会直接与中介者（PollinationMediator类）进行交互，在逻辑上利用该中介者封装了雄蕊在植物集群中定位某一植株，进而定位某一雌蕊的过程。

17.2 测试用例

1. 生成相应状态的植物作为测试环境。
2. 植物调用pollinate()进行传粉，pollinate()函数的实现采用了以Mediator类为接口的PollinationMediator对象。

18 Memento

18.1 测试逻辑

备忘录模式即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，以便以后当需要时能将该对象恢复到原先保存的状态。该模式又叫快照模式 。

备忘录模式有三个主要角色：

1. 发起人（Originator）角色：记录当前时刻的内部状态信息，提供创建备忘录和恢复备忘录数据的功能，实现其他业务功能，它可以访问备忘录里的所有信息。
2. 备忘录（Memento）角色：负责存储发起人的内部状态，在需要的时候提供这些内部状态给发起人。
3. 管理者（Caretaker）角色：对备忘录进行管理，提供保存与获取备忘录的功能，但其不能对备忘录的内容进行访问与修改。

在该测试中，备忘录存储的内容为农场在一次交易中创建的订单以及交易前的所有动植物产品的状态。

1. Traders为管理农场贸易的角色，它在备忘录模式中担任发起人角色。
2. TransationRecords为备忘录角色。
3. ManageRecords为管理者角色。

在该测试中：

1. 用户可以查看当前所有产品的状态，据此决策进行买入卖出产品，在买卖产品时Traders会记录当前信息并存储到备忘录角色中，满足备忘录模式中捕获系统状态的特性。
2. 用户可以查看所有历史订单（即所有的备忘录），可以选择历史订单进行状态恢复，满足备忘录模式中可恢复到历史状态的特性。

18.2 测试用例

1. 用户查看当前所有产品的状态，据此决策执行买卖操作。
2. 用户买入产品（CHICKEN，DOG、PERCH、APPLE、CHERRY、POTATO、TOMATO）
3. 用户查看所有的历史订单，据此选择恢复到哪一历史状态。
4. 用户根据定单号选择恢复到哪一历史状态。

19 Observer

19.1 测试逻辑

子类通过实现Observer接口中的response方法来决定观察者要观察的内容。

消耗品类中可以添加或删除指定的观察者，同时，当使用consume()方法时（即消耗了消耗品的数量），会调用所有观察该类的观察者的response()方法，判断该改变是否引起了某种危险。

19.2 测试用例

1. 创建一个Consumable类的示例。
2. 为这个类添加一个观察者。
3. 消耗相应数量，观察者产生效果。

20 Prototype

20.1 测试逻辑

原型模式即用一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型相同或相似的新对象。

在该测试中，以Animal中的Chicken为例，首先创建一个Chicken对象chicken，然后调用根据原型模式设计的clone方法，直接根据chicken复制出一个Chicken对象chicken1。

为了检测chicken和chicken1是否的确是两个对象而不是二者指向同一地址，可以使用java的“==”与equals（equals继承自Object中的equals方法，需要在类中重新实现）的不同来测试。

“==”用来判断两个对象是否指向同一地址，“equals”则用来判断两个对象中内容是否相同。

如果“chicken==chicken1”返回false，而"chicken.equals(chicken1)"返回true，则说明Prototype pattern得到了实现。

20.2 测试用例

1. 创建一个Chicken对象chicken
2. 调用chicken的clone方法复制一个新的Chicken对象chicken1.
3. 输出“chicken==chicken1”和"chicken.equals(chicken1)"的结果。

21 Proxy

21.1 测试逻辑

代理模式的核心在于将业务交管给中间代理去处理，使得复杂的业务能快速完成。

在该测试中，我们将测试工人们通过代理拿取工具的业务。由于工人和工具的数量都可能很大，而且每个工人和工具间都可能产生交互，便需要为提供代理来管理工具，保证不会发生拿取已经被拿取的工具，或者归还已经被归还的工具的错误业务结果。

对于每一个新增的工具，都会自动生成一个专门的Proxy来负责。这样保证了所有工具之间的相互独立。

21.2 测试用例

1. 用户查看所有工具信息，其信息包括：工具No.、工具状态；
2. 用户新建一个工具，并加入工具库，在新建的同时也为其分配了一个代理；
3. 用户申请取走一个工具，代理对该请求进行相应处理；
4. 用户申请归还一个工具，代理对该请求进行相应处理；

22 Singleton

22.1 测试逻辑

单例模式是为了确保在整个程序中只有一个某个类的实例，并且提供一个全局的接入点。在我们的测试中，显然农场主是一个单例模式，如果调用它的 new 函数将会直接报错，而调用 getInstance() 时只会在头一次创建新对象，而我们也会在此时 print 一些内容来证明这一点。

22.2 测试用例

两次调用 FarmOwner.getInstance() 通过输出可以判断我们只在第一次调用的时候创建了对象。

23 State

23.1 测试逻辑

状态模式的核心在于将某类的相关状态用一个相应的环境类来进行托管，从而简化其他类访问该类时对其状态的相关操作。在我们的场景下，我们将工具的新旧状态提取出来托管。

在测试中，我们将通过一系列的对工具耐久度的改变来展现工具的状态变化。其中涉及到的边界问题以及越界问题（如耐久度超过上限，或低于底限），都能得到妥善的处理和解决。

在这个过程中，我们只对其环境类的耐久度进行操作，其他的一切有关状态的参数以及属性都统统封装在状态类或者环境类内部自行运行和维护，因此体现了状态模式的设计思想。

23.2 测试用例

模拟一个工具被多次频繁重度使用后，再进行不同程度的维修的过程中的状态变化。该过程在整个农场中发生频度较高，也适合用来展现该设计模式。

23.3 测试用例

模拟一个植物随着时间的推移而生长的过程中，植物生长状态的变化。该过程在整个农场中发生频度较高，也适合用来展现改设计模式。

通过调用植物类的moveToNext()方法完成植物生长状态由Grow-Mature-Harvestable-Dead状态的推进转变。

24 Strategy

24.1 测试逻辑

Strategy模式定义并封装一系列算法，由具体对象根据场景选择不同的策略，从而调用到对应的不同算法。

在我们的场景下，我们为植物提供了多种受粉方式，如自然传粉和人工授粉，分别对应的策略类为SpontaneousPollination和ArtificialPollination。继承自Mediator的类PollinationMediator将会根据action的不同，选择不同的策略来为对应植株进行授粉操作。

在该测试中，我们将对不同植物根据action的不同选择不同的策略来为对应植株采取授粉策略，对某些植物采取自然传粉策略，对某些植物采取人工授粉策略。

24.2 测试用例

1. 生成相应状态的植物作为测试环境。
2. 为不同的植物选择不同的受粉策略PollinationStrategy()，需要自然传粉的植物选择SpontaneousPollination()，需要人工授粉的植物选择ArtificialPollination()。
3. 选择好受粉策略后，对各个植物分别依据选好的受粉策略调用pollinate()函数进行受粉操作。

25 Visitor

25.1 测试逻辑

访问者模式的重点在于根据不同访问者的需求和侧重点，被访问者在被访问时表现出不同的业务逻辑与属性。

而在我们的场景下，根据不同访问者例如农夫和管理员的需求和注重点，工具和消耗品接受不同访问者的访问的业务逻辑不同。

在该测试中，我们将测试同样当农夫和管理员去访问同一个或一批补给品时，其展现的业务逻辑和反馈的信息是否相同，以此来展现访问者模式的设计思想。

25.2 测试用例

在仓库中各有一个工具和一个消耗品，当农夫（关注工具和消耗品是否能使用）和管理员（关注工具的状态，是否需要修理以及消耗品的量和过期日期）先后访问它们时，都展现出不同的业务逻辑和信息输出。