1 Abstract Factory Pattern

2 Adapter Pattern

2.1 测试逻辑

动物的吃分为两种, 一类是食草动物(`GrassEat`), 食草动物吃草, 而草是无限量的; 另一类是其他动物(`NormalEat`), 这类动物的食物在吃完后需要向商店进行购买.
这里的`GrassEat`提供了一个`NormalEat`的适配器, 使得其他动物也可以通过`GrassEat`调用`NormalEat`的方法.

2.2 测试用例

用户创建`Sheep`, `Cat`, `Chicken`的实例.
用户创建`GrassEat`的实例`grassEat`.
用户调用`grassEat`的方法.

```
Animal cat = animalFactory.run("Cat");
Animal sheep = animalFactory.run("Sheep");
Animal chicken = animalFactory.run("Chicken");
GrassEat grassEat = new GrassEat();
grassEat.eat("Grass", sheep);
grassEat.eat("Normal", cat);
grassEat.eat("Normal", chicken);
```

2.3 测试结果

```
-------------Adapter Test-------------
There are infinity Grass in store, and the Sheep needs 1 Grass.
Animal (id: 2) have eaten 1 Grass!
There are 2 CatFood in store, and the Cat needs 2 CatFood.
Animal (id: 1) have eaten 2 CatFood!
There are 2 ChickenFood in store, and the Chicken needs 3 ChickenFood.
Animal (id: 3) have eaten 2 ChickenFood!
```

3 Bridge Pattern

3.1 测试逻辑

yell动作有两个维度, 次数和类型. 这里通过桥接模式分离了这两个维度.

3.2 测试用例

Cat yell 3 times:

- 创建一个`AnimalYell`实例`sheepYell`.
- 创建一个`YellTimes`实例`yellThreeTimes`.
- 向`sheepYell`中传入`yellThreeTimes`.
- 调用`catYell`.

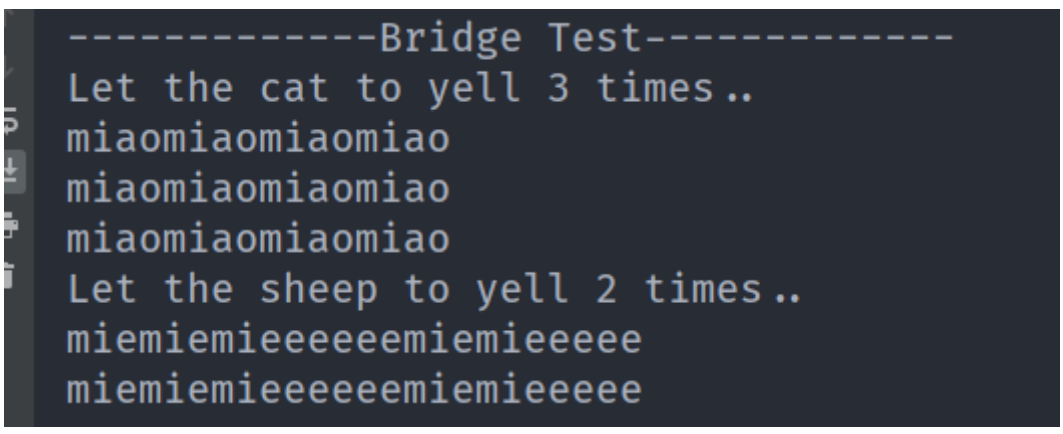
Sheep yell 2 times:

- 创建一个`AnimalYell`实例`SheepYell`.
- 创建一个`YellTimes`实例`yellTwiceTimes`.
- 向`sheepYell`中传入`yellTwiceTimes`.
- 调用`sheepYell`.

```
System.out.println("Let the cat to yell 3 times..");
AnimalYell catYell = new CatYell();
YellTimes yellThreeTimes = new YellThreeTimes();
catYell.setYellTimes(yellThreeTimes);
catYell.yell();

System.out.println("Let the sheep to yell 2 times..");
AnimalYell sheepYell = new SheepYell();
YellTimes yellTwiceTimes = new YellTwiceTimes();
sheepYell.setYellTimes(yellTwiceTimes);
sheepYell.yell();
```

3.3 测试结果



4 Builder Pattern

4.1 测试逻辑

建造者模式是为了由多个简单的对象一步一步创建成一个复杂的对象，它很方便的提供了一些构造函数所拥有的功能。在商店创建订单时会出现订单中包含不同种类和不同数量的商品的情况，这个时候我们就需要采用建造者模式，将这些不同的类的商品，根据他们的单价以及数量来计算他们的总价格。

4.2 测试用例

我们要创建一个分别出售100单位的鸡蛋，玉米和小麦的订单：

1.先给库存中添加好足够数量的货物。

2.将他们加入一个map中，调用Employee中的createSaleOrder(Map<Class, Integer> map)来创建订单。

3.在创建订单前，需提前算好订单的价格，调用MultipleItemOrderBuilder里面的getCost(Map<Class, Integer> List)方法，该方法会逐个算出List中每一项的价格然后build一个总的价格。

4.从输出判断出创建订单成功。

4.3 测试结果

```
----------  Builder Test----------
Firstly,we will add some different goods in the repository
RepositoryProxy: Corn in repository add: 100, now: 100
Repository notifies all observers.
RepositoryProxy: Egg in repository add: 100, now: 3200
Repository notifies all observers.
RepositoryProxy: Wheat in repository add: 100, now: 100
Repository notifies all observers.
Now we creat a new order with multiple goods
We use Wheat to build the order!
We use Egg to build the order!
We use Corn to build the order!
Payment received
RepositoryProxy: Wheat in repository consume: 100, now: 0
Egg in repository consume: 100, now: 3100
Corn in repository consume: 100, now: 0
Repository notifies all observers.
Inventory updated
Summary updated
Order success
```

5 Chain of Responsibility Pattern

5.1 测试逻辑

在责任链模式中，由很多个会对请求进行处理的对象拼接在一起形成一条责任链，请求在这条链上传递，责任链根据对象的顺序依次让对象对请求进行处理，直到到达链的尾端。这使得系统可以对客户端传来的各种请求动态地组织和分配责任。
在test中，我们创造出了一个由factory组成的链条，链条的每个节点对应一种factory，并根据用户的需

求，也就是需要生产的东西来决定所需的factory相应的方法。通过给出一个或多个待处理的需求，就能通过生产工厂的回应确定责任链的运作是否合理。

5.2 测试用例

1. 新建一个责任链，将农作物加工厂和肉类加工厂放进去。
2. 要求生产各种产品。
3. 责任链依次把请求交给两个加工厂，各自对自己负责的产品进行加工。
4. 加工完成或者原材料不足则会放出通知。

5.3 测试结果

```
-------------Responsibility Chain Text-------------
RepositoryProxy: Chicken in repository add: 50, now: 50
RepositoryProxy: Wheat in repository add: 50, now: 50
RepositoryProxy: Egg in repository add: 50, now: 50
```

6 Command Pattern

6.1测试逻辑

在完成农田与仓库交互的时候，采用Command设计模式。当调用seeding()、giveFertilizer()等方法时，StockManager执行FarmManager的命令，询问数量，并在成功后将其减一，在harvest()调用时则将果实加一。

6.2 测试用例

在Farm Manager的harvest()被调用后，将发送AddItemCommand给仓库。

6.3 测试结果

```
-------------Command Test-------------
The harvest() function would send a AddItemCommand to add a corresponding product after successfully harvesting.
Fail, no plant here.
The seeding() function would send a QueryItemCommand to check the num of the seeds in repository, and send a DelItemCommand to subtract one if have planted the seed.
Succeeded to  plant a new Potato Planton No.0.
RepositoryProxy: PotatoSeed in repository consume: 1, now: 95
Succeeded to  plant a new Potato Planton No.51.
RepositoryProxy: PotatoSeed in repository consume: 1, now: 94

Our field:
No.0 fieldPotato Plant, growth point:0/100.
No.10 fieldCorn Plant, growth point:0/120.
No.11 fieldPotato Plant, growth point:0/100.
No.12 fieldWheat Plant, growth point:0/80.
No.51 fieldPotato Plant, growth point:0/100.
```

7 Composite

7.1测试逻辑

我们在种植这一过程中使用Composite设计模式。调用Farm Manager的操作时，即可展示Farm Manager包含FieldManager和StockManager，FiledManager类中包含Field类，Field类包含Plant类。

7.2 测试用例

调用Farm Manager的seeding()，将由FieldManager和StockManager分别执行各自的操作。

7.3 测试结果

8 Decorator Pattern

8.1 测试逻辑

对羊毛进行染色. 每进行一次装饰返回一个新的颜色的羊毛. (这里假定颜色是附加在羊毛上的而不是羊毛自身的一个属性)

8.2 测试用例

调用`sheep`的`` `beSheared()` ``函数, 得到羊毛.
创建3个不同的装饰器, 依次对羊毛染色.
再重复一次, 这次更换新的羊毛以及调换三个染色步骤的顺序.

```
Animal sheep = animalFactory.run("Sheep");
Animal sheep2 = animalFactory.run("Sheep");

System.out.println("Start getting 3 normal wool from sheep (id: " + sheep.id +
").");
Wool normalWool = ((Sheep) sheep).beSheared(3, "Normal");
System.out.println("Now start dyeing...");
System.out.println(
        new GreenWoolDecorator(
            new BlueWoolDecorator(
                    new RedWoolDecorator(normalWool)
            )
        ).getDescription()
);

System.out.println("\nStart getting 3 long wool from sheep (id: " + sheep2.id +
").");
Wool longWool = ((Sheep) sheep2).beSheared(3, "Normal");
System.out.println("Now start dyeing...");
System.out.println(
        new RedWoolDecorator(
                new BlueWoolDecorator(
                        new GreenWoolDecorator(longWool)
                )
        ).getDescription()
);
```

8.3 测试结果

```
Create a new sheep!
Start getting 3 normal wool from sheep (id: 2).
Having gotten 3 normal wool from sheep (id: 2)!
Now start dyeing...
This is 3 normal wool
____And it has been set to red now.
____And it has been set to blue now.
____And it has been set to green now.

Start getting 3 long wool from sheep (id: 4).
Having gotten 3 normal wool from sheep (id: 4)!
Now start dyeing...
This is 3 normal wool
____And it has been set to green now.
____And it has been set to blue now.
____And it has been set to red now.
```

9 Facade Pattern

9.1 测试逻辑

不同的动物有不同的排泄方式. 这里使用一个外观类`animaPooMaker`来统一调用`CatPoo`, `ChickenPoo`以及`SheepPoo`的逻辑.

9.2 测试用例

创建一个`AnimalPooMaker`实例.
调用该实例的方法.

```
AnimalPooMaker animalPooMaker = new AnimalPooMaker();
animalPooMaker.pooForCat(2);
animalPooMaker.pooForChicken(1);
animalPooMaker.pooForSheep(1);
```

9.3 测试结果

```
-------------Facade Test-------------
The cat poo 2 cat stool.
The chicken poo 1 chicken stool.
The sheep poo 1 sheep stool.
```

10 Factory Pattern

10.1 Farm测试

10.1.1 测试逻辑

我们将测试该模式运用于植株的种植过程，首先调用Farm Manager类的seeding()函数，它封装了PlantFactory类，实现了 Factory Method模式，判断输入的种子后可以在通过Plant Factory中生产3种不同的plant：Corn、Potato、Wheat。

10.1.2 测试用例

执行FarmManager的seeding()创建一个CornPlant、Wheat Plant或PotatoPlant。

10.1.3测试结果

```
------------Factory Method Test------------
PlantFactory class would produce corresponding plant according to the input class of seed.
Succeeded to  plant a new Corn Planton No.10.
RepositoryProxy: CornSeed in repository consume: 1, now: 197
Succeeded to  plant a new Potato Planton No.11.
RepositoryProxy: PotatoSeed in repository consume: 1, now: 96
Succeeded to  plant a new Wheat Planton No.12.
RepositoryProxy: WheatSeed in repository consume: 1, now: 97

Our field:
No.10 fieldCorn Plant, growth point:0/120.
No.11 fieldPotato Plant, growth point:0/100.
No.12 fieldWheat Plant, growth point:0/80.
```

10.2 Pasture测试

10.2.1测试逻辑

这里作者使用工厂模式，实现对不同种类动物的初始化，或者说生产/购买的动作。其最大的特点，在于提供一个Animal类的接口，使得编程者不需要一个特定的类，比如Cat或者Sheep类来接住他们。这样在调用一些共属于Animal类的函数的时候，工厂模式的优势就体现出来了。

作者利用两部分来实现工厂模式。首先，他们在pasture根目录下构建了Animal基类（接口）和Cat，Sheep，Chicken三个动物子类，并写好了他们的从属关系。然后，他们在Factory文件夹的AnimalFactory.java文件中，实现对动物的生产。具体是，当用户输入关键词字符串“Cat” “Sheep” “Chicken”的时候，函数会反对给他们一个新建的对象。因为这些对象都是基于Animal接口的，所以可以用一个Animal类型的变量来接住它。最后，函数做了鲁棒性考虑。当输入非法字符串，即不属于特定动物的字符串时，函数会输出警告，并且不返回任何结果。

10.2.2 测试用例

首先新建一个工厂animalFactory，调用这个工厂的run函数，分别输入Cat，Sheep，Chicken，就可以返回一个Cat，Sheep，Chicken类型的对象，用Animal类型的cat，sheep，chicken接住。这些对象供后续的设计模式使用。

10.2.3 测试结果

```
-------------Factory Test-------------
Create a new cat!
Create a new sheep!
Create a new chicken!
```

10.3 Shop测试

10.3.1 测试逻辑

Factory 模式提供了一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。十分便捷的划分了不同产品组，但是在OCP原则的规范下难以实现产品组内的扩展，但可以实现产品组的扩展。

在测试中，由工厂类OrderFactory控制创建相应类型的订单，Order接口定义了创建订单createOrder函数，具体的实现由BuyOrder、SaleOrder和Nullorder类实现，完成订单创建操作。

每当要新创建一个订单时，先创建订单工厂OrderFactory，之后通过工厂获取 BuyOrder 的对象，并调用它的createOrder 方法创建订单。

10.3.2 测试用例

1. 创建订单工厂
2. 新建购买订单
3. 新建出售订单

10.3.3 测试结果

```
---------- Order Factory Test ----------
Create order factory.


---------- Order Status Test ----------
Buy two kilogram corn.
Payment received
RepositoryProxy: Corn in repository add: 2, now: 2
Repository notifies all observers.
Inventory updated
Order success

Sale two kilogram corn.
Payment received
RepositoryProxy: Corn in repository consume: 2, now: 0
Repository notifies all observers.
Inventory updated
Summary updated
Order success
```

11 Flyweight Pattern

11.1 测试逻辑

调用FarmManager的giveFertilizer()方法时，使用肥料给农田施肥时，我们将直接采用引用肥料实例的方法，通过这种设计模式避免重新的创建和占用内存。

11.2 测试用例

多次调用Farm Manager的giveFertilizer()，每次使用同一个肥料实例。

11.3测试结果

```
------------Flyweight Test-----------
Fertilizers are using flyweight pattern which generate only one instance for each type of fertilizer.
Fail to give fertilizer. Plant is already mature.
Harvest: No.0 field, Potato Plant
RepositoryProxy: Potato in repository add: 1, now: 1
```

12 Interpreter Pattern

12.1 测试逻辑

interpreter，顾名思义，就是提供一个类似可解释的语句给解释器，让它“解释”出这个语句的意思。那么在这里，作者用来判断一些动物是否属于牲畜。

作者在Interpreter文件夹中实现了相关功能。首先他们设计了一个接口Expression。它有一个返回值为bool的抽象函数interpret，之后基于接口写了三个表达逻辑。ternimal，and和or。

- Ternimal构造函数，需要输入一个子字符串，它的interpret输入一个母字符串，用来判断母字符串是否包含子字符串的语句。
- and函数。输入两个表达。其表达函数是如果两个表达的interpret函数，返回值都为真，结果返回真，否则返回假。
- and函数。输入两个表达。其表达函数是如果两个表达的interpret函数，返回值只要一个为真，结果返回真，否则返回假。

之后实现解释器类，是Interpreter，作为暴露给外界的“接口”，给用户使用。这里的Interpreter亦包含了工厂模式。

12.2 测试用例

首先调用静态工厂的getLivestockExpression和getAllLivestockExpression两个函数，返回两个expression。这两个函数是两个Terminal语句的and和or语句，其中Terminal语句的子字符串已经写好。

之后给两个函数输入母字符串。函数先调用and和or语句的interpreter函数，再把母字符串分配给terminal的interpreter函数。

12.3 测试结果

```
-------------Interpreter Test-------------
Chicken is livestock? true
Sheep and chicken are all the livestock in pasture? true
```

13 Iterator Pattern

13.1 测试逻辑

仓库中所有遍历Map的情形使用的是JAVA封装的Iterator迭代器。

我们在输出仓库中所有库存时，使用Iterator迭代器来遍历每一个entry对象。

13.2 测试用例

1.调用仓库代理类的showAllItems()

13.3 测试结果

```
------------ Test Iterator -----------
Egg in summary: 6000
Corn in summary: 2
```

14 Mediator Pattern

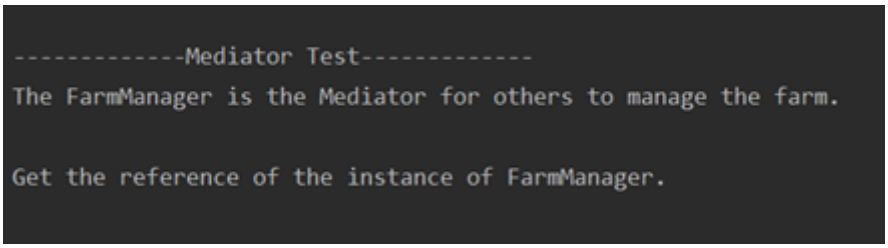
14.2.1测试逻辑

在FarmManager中使用中介者模式，作为StockManager和FieldManager类的中介者，使直接和中介者(FarmManager类)进行交互。

14.2.2 测试用例

调用FarmManager的seeding()，通过FieldManager实际在田地里种下植株。

14.2.3 测试结果



15 Memento Pattern

15.1测试逻辑

调用Farm Manager的save()和restore()函数即可间接调用FieldMementoManager，使用备忘录模式，用于存储或恢复植物的当前状态，每一次，所有植物都对应一个FieldMemento类，FieldMementoManager就是FieldMemento的管理类。FieldMemento备份FieldManager的fieldList数据，由FieldMemengtoManager管理，多种的全部拔掉。其中FieldMementoManager的展示所有备忘录函数。

15.2测试用例

调用Farm Manager的save()后，更改农田的植株，之后调用restore()传入必要的参数，恢复当时的农田。

15.3 测试结果

```
-------------Memento Test-------------

Save the status at the beginning.

Saving memento:

Our field:
Empty.

Succeed.



Start to show all mementos.

No.0 memento:
Empty.


All showed.

Succeeded to  plant a new Potato Planton No.0.
RepositoryProxy: PotatoSeed in repository consume: 1, now: 99
Fail. There is already a plant.
Succeeded to  plant a new Potato Planton No.1.
RepositoryProxy: PotatoSeed in repository consume: 1, now: 98
Succeeded to  plant a new Corn Planton No.2.
RepositoryProxy: CornSeed in repository consume: 1, now: 199
Save the second status.

Saving memento:

Our field:
No.0 fieldPotato Plant, growth point:0/100.
No.1 fieldPotato Plant, growth point:0/100.
No.2 fieldCorn Plant, growth point:0/120.

Succeed.

Succeeded to  plant a new Wheat Planton No.3.
RepositoryProxy: WheatSeed in repository consume: 1, now: 99
Succeeded to  plant a new Wheat Planton No.4.
RepositoryProxy: WheatSeed in repository consume: 1, now: 98
Succeeded to  plant a new Corn Planton No.5.
RepositoryProxy: CornSeed in repository consume: 1, now: 198

Saving memento:

Our field:
No.0 fieldPotato Plant, growth point:0/100.
No.1 fieldPotato Plant, growth point:0/100.
No.2 fieldCorn Plant, growth point:0/120.
No.3 fieldWheat Plant, growth point:0/80.
```

```
No.4 fieldWheat Plant, growth point:0/80.
No.5 fieldCorn Plant, growth point:0/120.

Succeed.

Start to show all mementos.

No.0 memento:
Empty.


No.1 memento:
No.0 field: Potato Plant. Growth point: 0/100.
No.1 field: Potato Plant. Growth point: 0/100.
No.2 field: Corn Plant. Growth point: 0/120.


No.2 memento:
No.0 field: Potato Plant. Growth point: 0/100.
No.1 field: Potato Plant. Growth point: 0/100.
No.2 field: Corn Plant. Growth point: 0/120.
No.3 field: Wheat Plant. Growth point: 0/80.
No.4 field: Wheat Plant. Growth point: 0/80.
No.5 field: Corn Plant. Growth point: 0/120.


All showed.

Restore the second status, by just destroying all the plants planted after that.
Restoring No.1 memento.

Our field:
No.0 fieldPotato Plant, growth point:0/100.
No.1 fieldPotato Plant, growth point:0/100.
No.2 fieldCorn Plant, growth point:0/120.

Succeed.
Restore the first status
Restoring No.0 memento.

Our field:
Empty.

Succeed.
```

16 Observer Pattern

16.1 Factory测试

16.1.1 逻辑测试

Observer设计模式指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。这种模式有时又称作发布-订阅模式、模型-视图模式，它是对象行为型模式。在我们的测试用例中，用户通过调用加工厂的getMachinesState()方法来获取当前工厂内所有机器的状态(0代表正常，1代表异常)

16.1.2 测试用例

1. 首先初始化农产品加工厂与牧产品加工厂
2. 通过调用加工厂的addMachine()方法来向工厂内添加机器
3. 用户通过调用工厂中的getMachineState()方法来获取机器的状态（为了模拟机器损坏的情况，程序中采用产生随机数的方法来设置机器的state属性）

16.1.3 测试结果

```
-------------Observer/Prototype Test-------------
Check the State of FarmProcessingFactory's Machines:
check machines' state
NO.0machine's state:0
NO.1machine's state:0
NO.2machine's state:0
NO.3machine's state:0
NO.4machine's state:0
NO.5machine's state:0
NO.6machine's state:0
NO.7machine's state:0
NO.8machine's state:0
NO.9machine's state:0
NO.10machine's state:0
NO.11machine's state:1
remove NO.11 machine
successfully cloned machine
successfully added machine
NO.12machine's state:0
NO.13machine's state:0
NO.14machine's state:0
Check the State of PastureProcessingFactory's Machines:
check machines' state
NO.0machine's state:0
NO.1machine's state:0
NO.2machine's state:0
NO.3machine's state:0
NO.4machine's state:0
NO.5machine's state:0
NO.6machine's state:0
NO.7machine's state:0
NO.8machine's state:0
NO.9machine's state:0
NO.10machine's state:0
NO.11machine's state:0
NO.12machine's state:0
NO.13machine's state:1
remove NO.13 machine
successfully cloned machine
successfully added machine
NO.14machine's state:0
```

16.2 Pasture测试

16.2.1 测试逻辑

观察者模式，作者在这里先建了一个observer接口。之后三个操作继承了该接口，分别是Telegraph, Telephone, WarningLight.

这三个操作的构造函数做了两件事，一是绑定他们的monitor，二是将他们添加到monitor的触发列表中。update函数代表他们被触发到会执行的命令。

AnimalMonitor实现了监视功能，用来监视动物。它有三个内置成员变量：observers, id, type. observers是触发列表，id和type是监视动物的id和种类。当调用animalRunAway函数时，它会存储id和type信息，再调用各个observer的update函数。

16.2.2 测试用例

首先新建一个AnimalMonitor对象monitor。给该监视器添加三个不同类型的observer。最后执行animalRunAway，看monitor的反应情况。

16.2.3 测试结果

```
------------Observer Test------------
Warning light! One Chicken (id: 3) has run away!
Telegraph from telegraph machine 1! One Chicken (id: 3) has run away from com.pasture! Please help searching!
Automatical telephone! One Chicken (id: 3) has run away from your com.pasture!
```

16.3 Shop测试

16.3.1 测试逻辑

商店类对仓库代理类形成观察者模式。商店对仓库容量进行观察，每当仓库的容量变化时，仓库会通知观察它的商店类。

我们使用attach()方法将仓库加入仓库代理类的观察池，每当仓库容量发生变化时，都会调用notifyAll()方法来调用所有它观察池中的对象的update()方法。

16.3.2 测试用例

1.将商店类加入仓库代理类的观察池

2.通过仓库代理向仓库添加9100的鸡蛋

16.3.3 测试结果

```
---------- Test SingleStore strategy and observer ----------
Single RepositoryProxy constructs.
RepositoryProxy gets a new observer.
The repository has attached the SingleStore!
RepositoryProxy: Egg in repository add: 9100, now: 9100
Repository notifies all observers.
The SingleStore has sold 1000 Eggs,whose initial price is 5000.0
```

17 Prototype Pattern

17.1 逻辑测试

ProtoType设计模式是指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。在这里，原型实例指定了要创建的对象的种类。用这种方式创建对象非常高效，根本无须知道对象创建的细节。在我们的测试用例中，在用户调用Observer的getMachinesState()方法来获取机器状态时，如果检测到机器损坏，就将机器从加工厂的List中移除，然后通过Machine类的clone()方法克隆一个新的机器加入列表中。

17.2 测试用例

1. 首先初始化农产品加工厂与牧产品加工厂
2. 通过调用加工厂的addMachine()方法来向工厂内添加机器

3. 用户通过调用工厂中的getMachineState()方法来获取机器的状态（为了模拟机器损坏的情况，程序中采用产生随机数的方法来设置机器的state属性）程序通过遍历机器的状态信息来判断机器是否损坏，若损坏则移除该机器，然后通过调用clone()函数来添加机器。

17.3 测试结果

```
-------------Observer/Prototype Test-------------
Check the State of FarmProcessingFactory's Machines:
check machines' state
NO.0machine's state:0
NO.1machine's state:0
NO.2machine's state:0
NO.3machine's state:0
NO.4machine's state:0
NO.5machine's state:0
NO.6machine's state:0
NO.7machine's state:0
NO.8machine's state:0
NO.9machine's state:0
NO.10machine's state:0
NO.11machine's state:1
remove NO.11 machine
successfully cloned machine
successfully added machine
NO.12machine's state:0
NO.13machine's state:0
NO.14machine's state:0
Check the State of PastureProcessingFactory's Machines:
check machines' state
NO.0machine's state:0
NO.1machine's state:0
NO.2machine's state:0
NO.3machine's state:0
NO.4machine's state:0
NO.5machine's state:0
NO.6machine's state:0
NO.7machine's state:0
NO.8machine's state:0
NO.9machine's state:0
NO.10machine's state:0
NO.11machine's state:0
NO.12machine's state:0
NO.13machine's state:1
remove NO.13 machine
successfully cloned machine
successfully added machine
NO.14machine's state:0
```

18 Proxy Pattern

18.1 Farm测试

18.1.1 测试逻辑

在农场的设计中，我们调用Farm Manager的方法后，将指派StockManager和FieldManager进行代理工作，而FarmManager本身无需工作，此时FarmManager代理所有StockManager和FieldManager的操作，且这两个被代理类无法在farm包外访问。

18.1.2 测试用例

调用FarmManager的seeding()后，将由FieldManager代理执行种植操作。

18.1.3 测试结果

```
------------Proxy Test------------
FieldManager is the proxy when FarmManager invoking seeding(), giveFertilizer(), or harvest().
It tries to add or delete plant to the field.
and StockManager is the proxy of FarmManager to operate on repository.

Succeeded to  plant a new Potato Planton No.0.
RepositoryProxy: PotatoSeed in repository consume: 1, now: 97
Give OrganicFertilizerto the plant (Potato Plant) on No.0 field, grow 50 points, present growth point: 50/100.
RepositoryProxy: OrganicFertilizer in repository consume: 1, now: 99
Give OrganicFertilizerto the plant (Potato Plant) on No.0 field, grow 50 points, present growth point: 100/100.
RepositoryProxy: OrganicFertilizer in repository consume: 1, now: 98
```

18.2 Shop测试

18.2.1 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。

我们将仓库类用仓库代理类来负责，所有对仓库的操作都要通过仓库代理类来实现，不能直接对仓库进行操作。

当代理类的方法被调用时，会输出对应的信息，确保对于仓库的操作确实地被代理的。被代理的仓库类也会输出对应的信息。

我们使用仓库的增加和消耗方法来测试代理类的正确代理。

18.2.2 测试用例

1.通过仓库代理向仓库添加100的玉米

2.通过仓库代理向仓库请求消耗150的玉米

3.通过仓库代理向仓库请求消耗100的玉米

18.2.3 测试结果

```
---------- Test RepositoryProxy Proxy ----------
Test: Add Corn 100.
RepositoryProxy: Corn in repository add: 100, now: 100
Test: Ask to consume Corn 150.
RepositoryProxy: Ask rejected.
Test: Ask to consume Corn 100.
RepositoryProxy: Corn in repository consume: 100, now: 0
```

19 Singleton Pattern

19.1 Farm测试

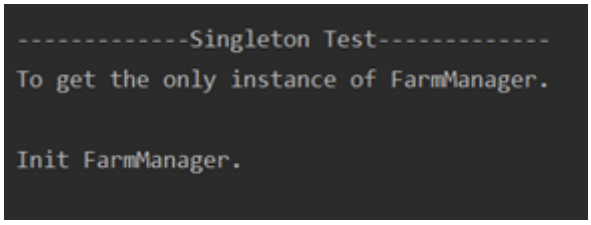
19.1.1 测试逻辑

我们可以测试我们的FarmManager类的getInstance()，该田地的实例在全局中只有一个，可以通过唯一方法取得。

19.1.2 测试用例

调用FarmManager的getInstance()。

19.1.3 测试结果



19.2 Shop测试1

19.2.1 测试逻辑

单例模式是为了保证在整个程序运行过程中某个类只能拥有一个实例化的对象，从而保证程序的正确运行。

我们将仓库代理设计为单例，将仓库代理的构造函数设计为私有方法。获取唯一的仓库代理实例只能通过静态公有的Instance()方法。

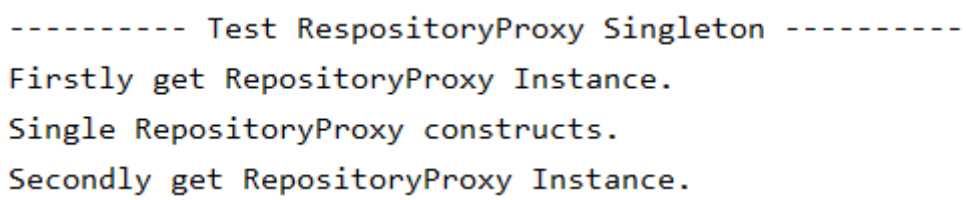
在RepositoryProxy类的构造函数中输出信息，确保当类被构造时能够被知晓。连续两次调用RepositoryProxy.Instance()来展示单例对象的构造过程

19.2.2 测试用例

1.输出并第一次调用Instance()方法

2.输出并第二次调用Instance()方法

19.2.3 测试结果



19.3 Shop测试2

19.3.1 测试逻辑

单例模式是为了保证在整个程序运行过程中某个类只能拥有一个实例化的对象，从而保证程序的正确运行。在我们的程序中商店这个类只能拥有一个instance，在程序中直接new一个商店会报错，我们将singlestore的构造函数设为private，防止外部重新创建新的对象，要想获得实例，只能功能调用getInstance()方法来实现。

19.3.2 测试用例

调用SingleStore.getSingleStore()方法获得一个singlestore的实例，根据输出判断获取实例成功。

19.3.3 测试结果

```
---------- SingleStore Singleton Test ----------
You have got a SingleStore!
```

20 State Pattern

20.1 Factory测试

20.1.1 逻辑测试

State设计模式是指对有状态的对象，把复杂的"判断逻辑"提取到不同的状态对象中，允许状态对象在其内部状态发生改变时改变其行为。在我们的测试用例中，加工厂中持有一个Environment的对象(该对象持有humidity和temperature属性)，当外界环境发生变化时就会调用加工厂中的handle()方法来调整工厂中的环境。

20.1.2 测试用例

1. 首先初始化农产品加工厂与牧产品加工厂
2. 通过调用加工厂的setEnvironment()方法来初始化加工厂的环境
3. 用户通过调用工厂中的setEnvironment()方法来模拟外界环境发生改变的情况，之后在调用加工厂类的handle()方法来根据环境调整加工厂中的环境。

20.1.3 测试结果

```
-------------State Test-------------
Check FarmProcessingFactory's Environment:
current humidity:32%
current temperature:29degree
humidity is too high
humidity decreased to 15%
temperature is too high
temperature decreased to 15degree
Check PastureProcessingFactory's Environment:
current humidity:15%
current temperature:15degree
```

20.2 Shop测试

20.2.1 测试逻辑

State模式允许对象在内部状态发生改变时改变它自身的行为，提供了方便的解决复杂对象状态转换的方法、解决了不同状态下行为的封装问题。在我们的场景中，我们对订单的状态变化使用状态模式。

在测试中，我们将通过创建订单来展示过程中的订单状态变化。从未支付阶段（UnpaidState）到已支付阶段（PaidState）到商品已交付阶段（DeliveredState）。通过状态模式的运用，减少订单错误的可能。

三种状态的区别在于支付和商品交付动作 pay() 和 deliverGoods() 方法，同一订单在不同的阶段会有不同的方法，如在PaidState下该方法将会在收到商品交付命令后更新库存，并且重置当前订单的状态。

20.2.2 测试用例

1. 新建购买订单
2. 新建出售订单

20.2.3 测试结果

```
---------- Order Status Test ----------
Buy two kilogram corn.
Payment received
RepositoryProxy: Corn in repository add: 2, now: 2
Repository notifies all observers.
Inventory updated
Order success

Sale two kilogram corn.
Payment received
RepositoryProxy: Corn in repository consume: 2, now: 0
Repository notifies all observers.
Inventory updated
Summary updated
Order success
```

21 Strategy Pattern

21.1 Pasture测试

21.1.1 测试逻辑

策略模式，对不同的场景调用不同的已经被封装好的算法，以保持代码的规范和整洁。这里作者希望用策略模式实现动物与动物之间的互动。

首先作者在com.stratery包中设置了一个strategy接口，代表互动的可能模式。之后三个实体的互动演出，Fight，Joke，Sing。最后用Interaction作为一个提供给外界的接口。

21.1.2 测试用例

测试用例分为三组。首先都新建一个互动模式，并指明是什么互动模式，以Interaction来承接。然后输入要互动的两只动物，让他们执行Interaction，并返回互动结果。

21.1.3 测试结果

```
-------------Strategy Test-------------
Cat(id: 1) sing to Sheep(id:2)!
Sheep(id: 2) joke on Chicken(id:3)!
Chicken(id: 3) fight with Cat(id:1)!
```

21.2 Shop测试

21.2.1 测试逻辑

策略模式是通过封装一系列策略，根据不同的情况采用不同测的策略应对不同的需求。

在商店的情境下，会根据库存量的多少来对订单采取不同的销售策略。分别有normalVIP、goldVIP和superVIP对应不同的优惠力度，从而计算出订单的优惠后的价格。

在该模式的测试中，我们采用SingleStore调用ExecuteStrategy类的calculatePrice方法来调用SaleStrategy接口的三个实体类进行价格计算。

21.2.2 测试用例

1.首先让仓库与商店进行连接，实时更新策略。

2.添加Corn的库存让其库存量先处于superVIP的范围内，然后进行购买。

3.购买后Corn库存量减少到goldVIP的范围内，再次进行购买。

4.之后Corn库存量减少到normalVIP的范围内进行购买。

5.从输出的结果发现三次购买，商店采用了不同的销售策略，导致同一种商品的单价不同。

21.2.3 测试结果

```
---------- SingleStore strategy and observer Test ----------
Single RepositoryProxy constructs.
RepositoryProxy gets a new observer.
The repository has attached the SingleStore!
RepositoryProxy: Egg in repository add: 9100, now: 9100
Repository notifies all observers.

The strategy is normalVIP, whose discount is 7.0
The SingleStore has sold 1000 Eggs,whose initial price is 5000.0
The Eggs cost the customer 3500.0
Payment received
RepositoryProxy: Egg in repository consume: 1000, now: 8100
Repository notifies all observers.
Inventory updated
Summary updated
Order success

The strategy is goldVIP, whose discount is 8.0
The SingleStore has sold 4000 Eggs,whose price is 20000.0
The Eggs cost the customer 16000.0
Payment received
RepositoryProxy: Egg in repository consume: 4000, now: 4100
Repository notifies all observers.
Inventory updated
Summary updated
Order success

The strategy is superVIP, whose discount is 9.0
The SingleStore has sold 3000 Eggs,whose price is 5.0
The Eggs cost the customer 4500.0
Payment received
RepositoryProxy: Egg in repository consume: 1000, now: 3100
Repository notifies all observers.
Inventory updated
Summary updated
Order success
```

22 Template Pattern

22.1 测试逻辑

在模板模式中，一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。
在测试中，农场加工厂和牧场加工厂共同继承了加工产品的模板，但各自重写了加工产品的各个步骤。分别调用农场加工厂和牧场加工厂的生产商品的方法，验证农场加工厂和牧场加工厂实现了加工不同产品的功能。

22.2 测试用例

1. 首先初始化农场加工厂和牧场加工厂。
2. 分别调用农场加工厂和牧场加工厂的生产产品的方法。

22.3 测试结果

```
-------------Template Test-------------
Successfully access the ingredient: Wheat
Single RepositoryProxy constructs.
RepositoryProxy: Flour in repository add: 1, now: 1
Successfully produce and store the product: Flour
Successfully access the ingredient: Egg
RepositoryProxy: EggCake in repository add: 1, now: 1
Successfully produce and store the product: EggCake
```

23 Visitor Pattern

23.1 测试逻辑

在访问者模式中，元素的执行算法可以随着访问者改变而改变。根据模式，元素对象已接受访问者对象，这样访问者对象就可以处理元素对象上的操作。
在测试中，我们让维修人员作为访问者访问被损坏的机器，被损坏的机器在接受维修人员的访问后得到修复，重新恢复为正常工作的状态。

23.2 测试用例

1. 实例化一个代表维修人员的访问者类。
2. 将一台机器的状态设置为损坏状态。
3. 通过访问者访问被损坏的机器，将机器修复为正常工作状态。

23.3 测试结果

```
-------------Visitor Text-------------
No.0 Machine in FarmProcessingFactory is damaged
Successfully repair the machine through the visitor.
No.13 Machine in PastureProcessingFactory is damaged
Successfully repair the machine through the visitor.
```

24 Null Object Pattern

24.1 测试逻辑

在空对象模式（Null Object Pattern）中，一个空对象取代 Null 对象实例的检查。Null 对象不是检查空值，而是反应一个不做任何动作的关系。在我们的场景中，我们对创建订单时的错误输入运用空对象模式。

在测试中，我们通过输入一个错误的订单类型字串来测试空对象模式的作用。

24.2 测试用例

新建错误订单（orderType不是BUY或SALE，而是SAL）

24.3 测试结果

```
---------- Null Object Test ----------
Create wrong order
Order is not correct!
```