开心农场

Document for Design Pattern Project

一个简单的农场经营类游戏的框架，包含农场、牧场、加工厂、商店
等多个模块，应用到多种设计模式

DESIGN PATTERN, AUTUMN 2019

BY

1650350　乔　宇
1653632　尹汇锋
1654007　何淳伟
1751130　朱　宽
1751174　叶　琪
1751576　徐　扬
1751650　蒋伟博
1753499　潘小逸
1753603　林寅嘉
1753783　缪禕天

同濟大學
TONGJI UNIVERSITY

Tongji University
School of Software Engineering

1 题材综述

《开心农场》是一个模拟经营类游戏。玩家一共有4个模块，包括农场，牧场，加工厂，商场。玩家可以在这四个模块内自由穿梭，体验来自大自然农业生态的收成之乐。

在农场里，测试者将购买土豆、玉米、小麦种子，种植相应植株，施肥，收获植株成为相应的产品并添加至仓库等操作。还可以记录某一时间的田地的种植情况，在需要的时候将在那之后多种的植物拔除。

牧场里有，有羊鸡等各种家畜，也有狗猫等宠物。在这里可以体验到收获动物幼崽以及给饥饿的动物喂食。可以剪羊毛以及给羊毛染色。当有动物发生逃跑的时候，还会有警报来提醒，捉回动物。

加工厂从仓库获取原材料并对其进行加工，之后将加工好的产品放到仓库.此外加工厂还可以定期检查机器的状态和工厂的环境，若机器损坏或者环境不达标，则会修理、添加机器并调整工厂环境.

在商店里，将体验出售农场，牧场及加工厂产出的产品，并且还可以从外界购买农场需要的种子和化肥，牧场需要的动物幼崽等生产资料的过程。

2 Design Pattern 汇总表

编号	Design pattern name	实现套数	样例套数	Factory	Farm	Pasture	Shop
1	Abstract Factory	1	1	1			
2	Adapter	1	1			1	
3	Bridge	1	1			1	
4	Builder	1	1				1
5	Chain of Responsibility	1	1	1			
6	Command	1	1		1		
7	Composite	1	1		1		
8	Decorator	1	1			1	
9	Facade	1	1			1	
10	Factory Method	3	3		1	1	1
11	Flyweight	1	1		1		
12	Interpreter	1	1			1	
13	Iterator	2	1				2
14	Mediator	1	1		1		
15	Memento	1	1		1		
16	Observer	3	3	1		1	1
17	Prototype	1	1	1			
18	Proxy	2	2		1		1
19	Singleton	7	2		4		3
20	State	2	2	1			1
21	Strategy	2	2			1	1
22	Template Method	1	1	1			
23	Visitor	1	1	1			
24	Null Object	1	1				1

编号	Design pattern name	实现套数	样例套数	Factory	Farm	Pasture	Shop
共计	24	38	28	7	11	8	12

3 Design Pattern 详述

3.1 Abstract Factory Pattern

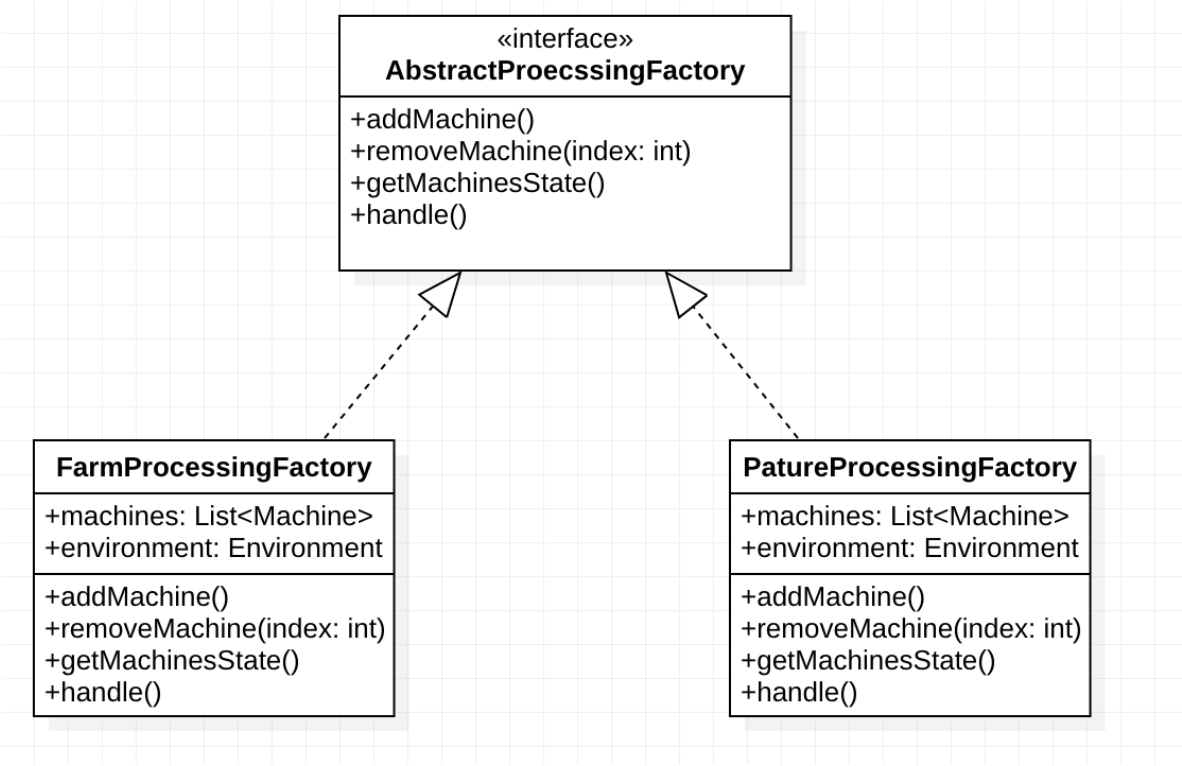
设计模式简述

Abstract 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。

3.1.1 API描述

函数名	作用
void addMachine()	接口定义方法，增加机器
void removeMachine（...）	接口定义方法，移除机器
void getMachinesState()	接口定义方法，获取所有机器状态
void setEnvironment（...）	接口定义方法，设置工厂的环境
void handle()	接口定义方法，根据外界环境变化调正工厂环境

3.1.2 类图



3.2 Adapter Pattern

设计模式简述

在软件开发中采用类似于电源适配器的设计和编码技巧被称为适配器模式。

通常情况下，客户端可以通过目标类的接口访问它所提供的服务。有时，现有的类可以满足客户类的功能需要，但是它所提供的接口不一定是客户类所期望的，这可能是因为现有类中方法名与目标类中定义的方法名不一致等原因所导致的。

在这种情况下，现有的接口需要转化为客户类期望的接口，这样保证了对现有类的重用。如果不进行这样的转化，客户类就不能利用现有类所提供的功能，适配器模式可以完成这样的转化。
在适配器模式中可以定义一个包装类，包装不兼容接口的对象，这个包装类指的就是适配器(Adapter)，它所包装的对象就是适配者(Adaptee)，即被适配的类。

适配器提供客户类需要的接口，适配器的实现就是把客户类的请求转化为对适配者的相应接口的调用。也就是说：当客户类调用适配器的方法时，在适配器类的内部将调用适配者类的方法，而这个过程对客户类是透明的，客户类并不直接访问适配者类。因此，适配器可以使由于接口不兼容而不能交互的类可以一起工作。这就是适配器模式的模式动机。

3.2.1 API 描述

我们有一个 `AnimalEat` 接口和一个实现了 `AnimalEat` 接口的实体类 `grassEat`。因为草是无限的不需要购买, 所以， `grassEat` 可以执行羊的吃的方法。

我们还有另一个接口 `AdvancedAnimalEat` 和实现了 `AdvancedAnimalEat` 接口的实体类 `NormalEat`。该类可以执行其他动物的吃的方法(因为除草之外, 其他动物的事物都需要购买而来)。

我们想要让 `grassEat` 也可以执行其他动物吃的动作。为了实现这个功能，我们需要创建一个实现了 `AnimalEat` 接口的适配器类 `EatingAdapter`，并使用 `AdvancedAnimalEat` 对象来执行所需的动作。

`grassEat` 使用适配器类 `EatingAdapter` 传递所需的eat的类型，不需要知道执行吃的方法的实际类。

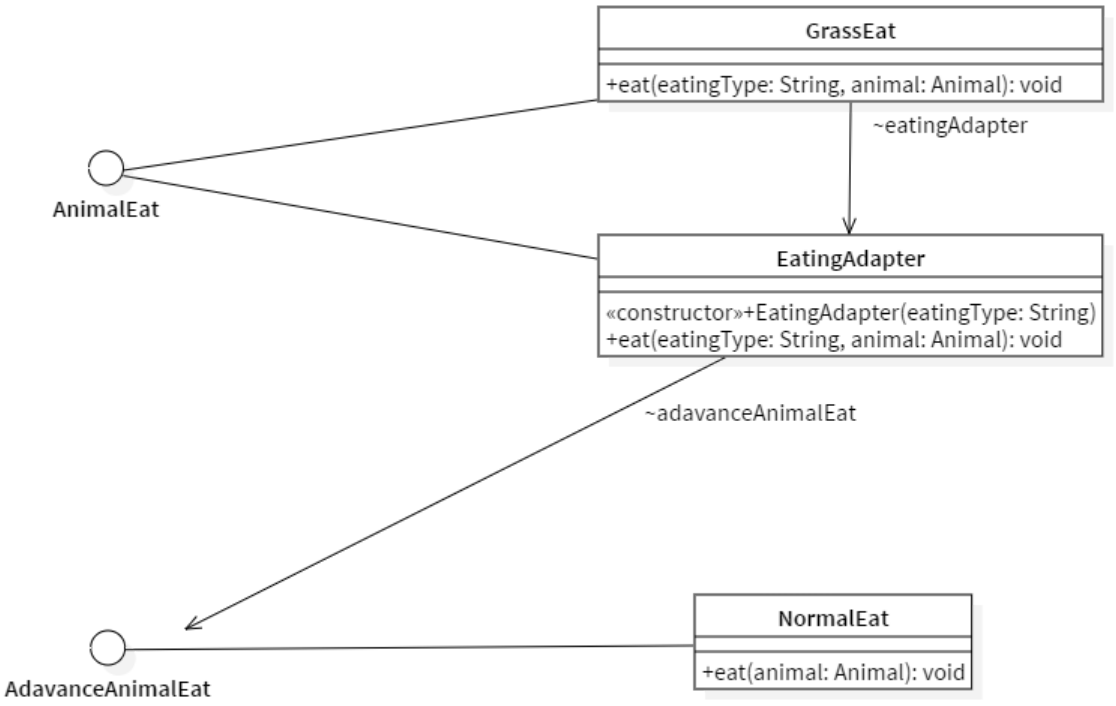
函数名	作用
`eat()`	进行吃的动作

3.2.2 使用例

```
grassEat.eat("Grass", sheep);
grassEat.eat("Normal", cat);
grassEat.eat("Normal", chicken);
```

```
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!
```

3.2.3 类图



3.3 Bridge Pattern

设计模式简述

桥接模式即将抽象部分与它的实现部分分离开来，使他们都可以独立变化。桥接模式将继承关系转化成关联关系，它降低了类与类之间的耦合度，减少了系统中类的数量，也减少了代码量。

3.3.1 API 描述

动物会发叫. 不同的动物会发出不同次数的叫声和不同类型的叫声. 这样就有了两个维度: 猫的叫的方式有叫两次和叫三次的不同, 同样, 羊叫的方式也有叫两声和叫三声之分. 于是在`AnimalYell`的`yell()`中, 并没有具体的实现`yell`的方法, 而是调用接口`yellTimes`来进行实现.

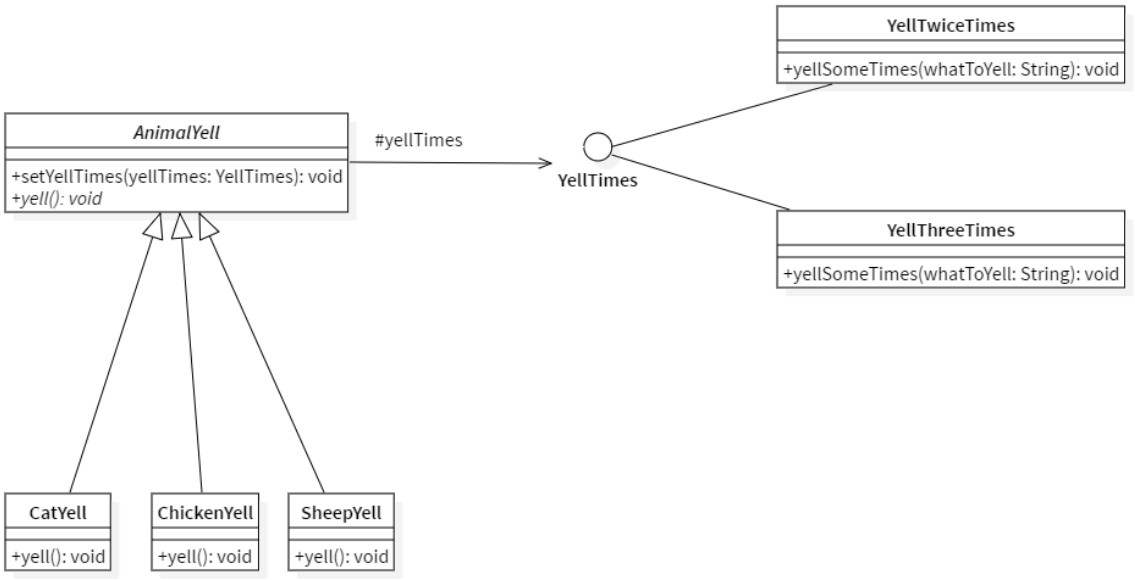
函数名	作用
`yell()`	进行叫的动作
`setYellTimes()`	设定叫的次数

3.3.2 使用例

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

```
Let the cat to yell 3 times..
miaomiaomiaomiao
miaomiaomiaomiao
miaomiaomiaomiao
```

3.3.3 类图



3.4 Builder Pattern

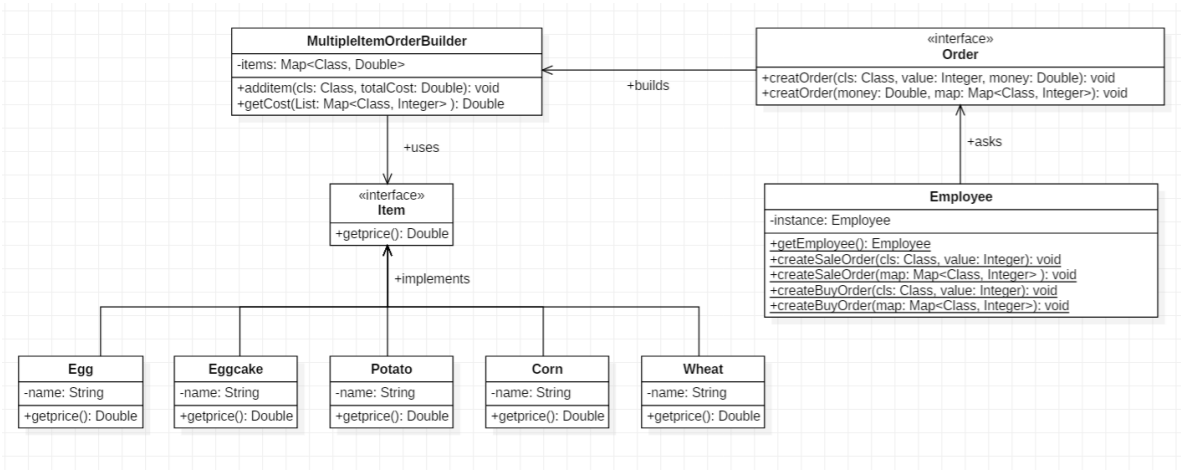
设计模式简述

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，我们使用这种模式的目的在于将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示，这对于设计者来说是一个解耦的过程。其优点在于用户使用简单，并且可以在不需要知道内部构建细节的情况下构建复杂的对象模型，建造者独立而易扩展，便于控制细节风险。缺点在于产品需要有共同点，范围有限制，且如果内部变化复杂，建造类会比较多。

3.4.1 API 描述

我们通过Employee类来生成订单，一个订单可以由很多个不同种类的商品组合，要创建这种订单我们就要通过Order接口MultipleItemOrderBuilder调用Item接口来组成每一种商品的总价，从而计算出整个订单的价格。最终传给Order接口创建新的订单。

3.4.2 类图



3.5 Chain of Responsibilty Pattern

设计模式简述

在Chain of Responsibilty Pattern中，为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。主要的意图是使多个对象都有机会处理同一个请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。并且对于职责链上的处理者负责处理请求，客户只需要将请求发送到职责链上即可，无须关心请求的处理细节和请求的传递，所以职责链将请求的发送者和请求的处理者解耦。

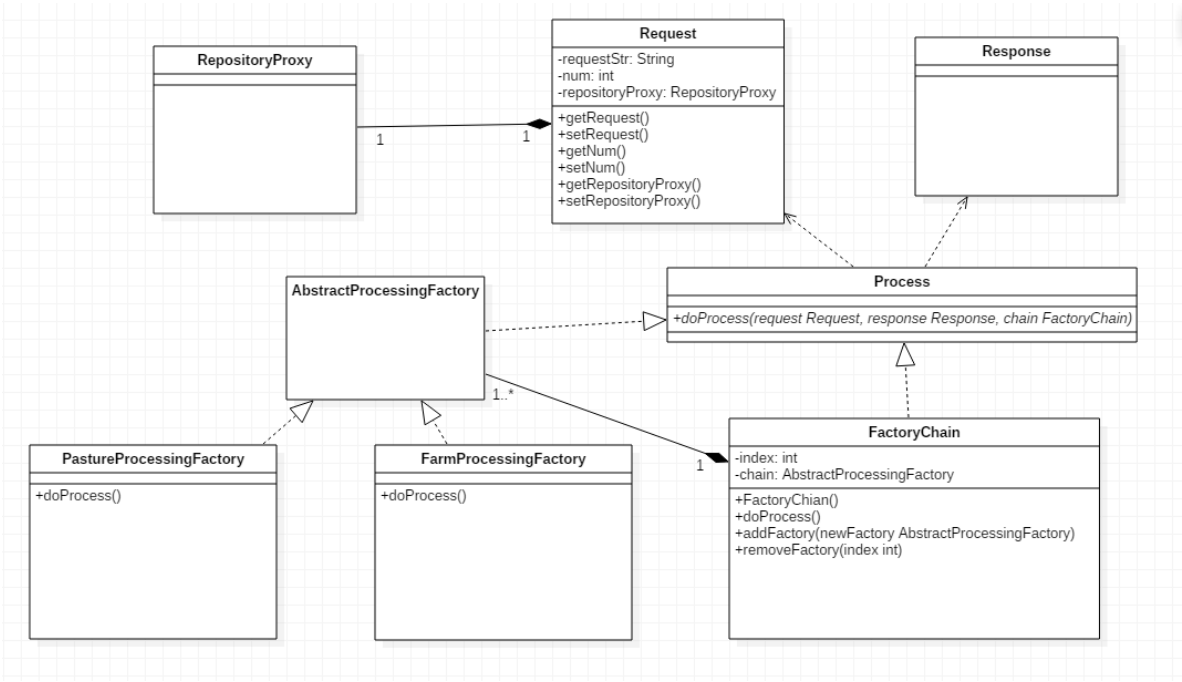
3.5.1 API 描述

对于具有多样性的加工厂的加工行为，定义了两个PastureProcessingFactory、FarmProcessingFactory实现自AbstractProcessingFactory的方法，两个类分别表示牧场产品加工工厂和农产品加工工厂，每个类中有不同的doProcess用来与不同的产品配对。同时有FactoryChain类用来存储Factory并顺序执行其中的doProcess方法。在doProcess方法中，Request是请求，需要输入所生产的产品（String）、数量（int）、仓库（RepositoryProxy）。Respond是回复。FactoryChain是工厂链，需要事先把工厂添加进入。

主要函数如下：

函数名	作用
void doProcess(Request request,Respond respond,FactoryChain chain)	生产，执行时调用工厂链的方法，会顺序执行里面所有工厂的方法。
boolean addFactory(AbstractPorcessingFactory new Factory)	往工厂链中添加工厂
boolean removeFactory(int index)	删除工厂链的工厂

3.5.2 类图



3.6 Command Pattern

设计模式简述

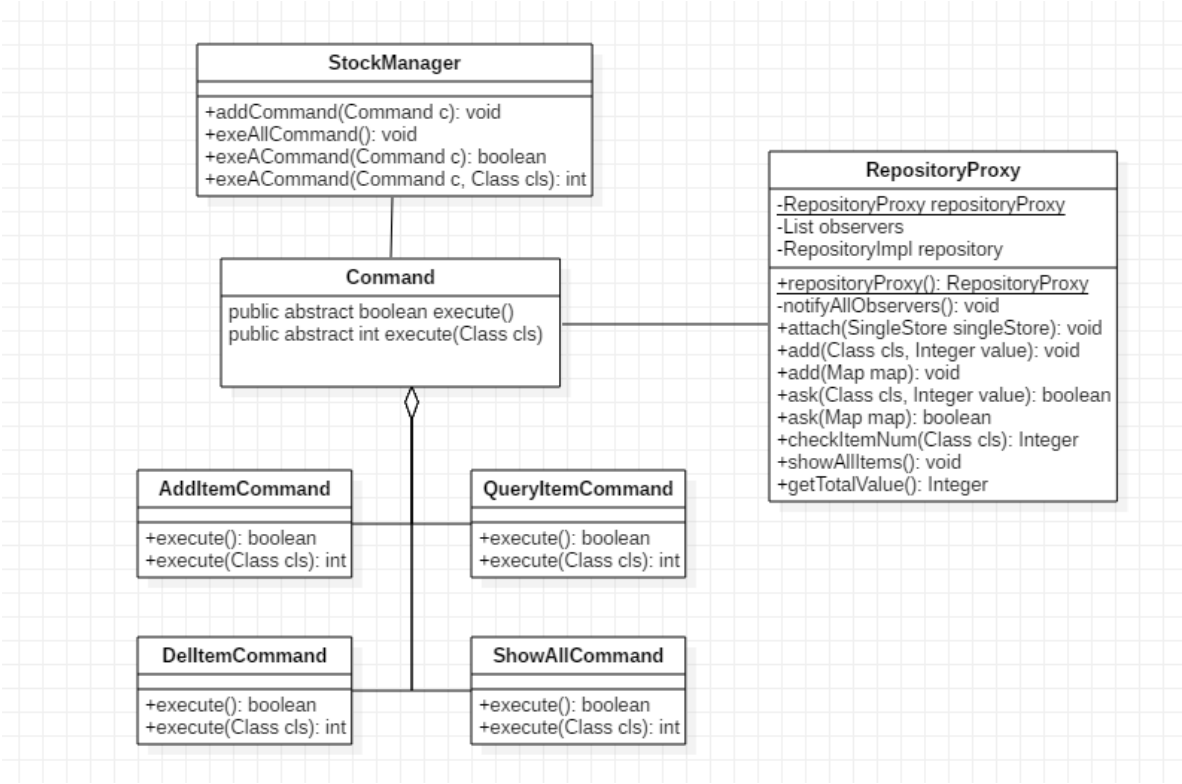
命令模式（Command Pattern）将一个请求封装成一个对象，从而使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。这是一种数据驱动的设计模式，它属于行为型模式。请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。

通过使用此设计模式降低了系统耦合度，容易添加新命令，同时也会使得系统中有过多具体命令类，过于繁杂。

3.6.1 API描述

在种植植株的时候，采用Command设计模式，StockManager执行FarmManager的命令，执行对象为另一个包的仓库。

3.6.2 类图



3.7 Composite

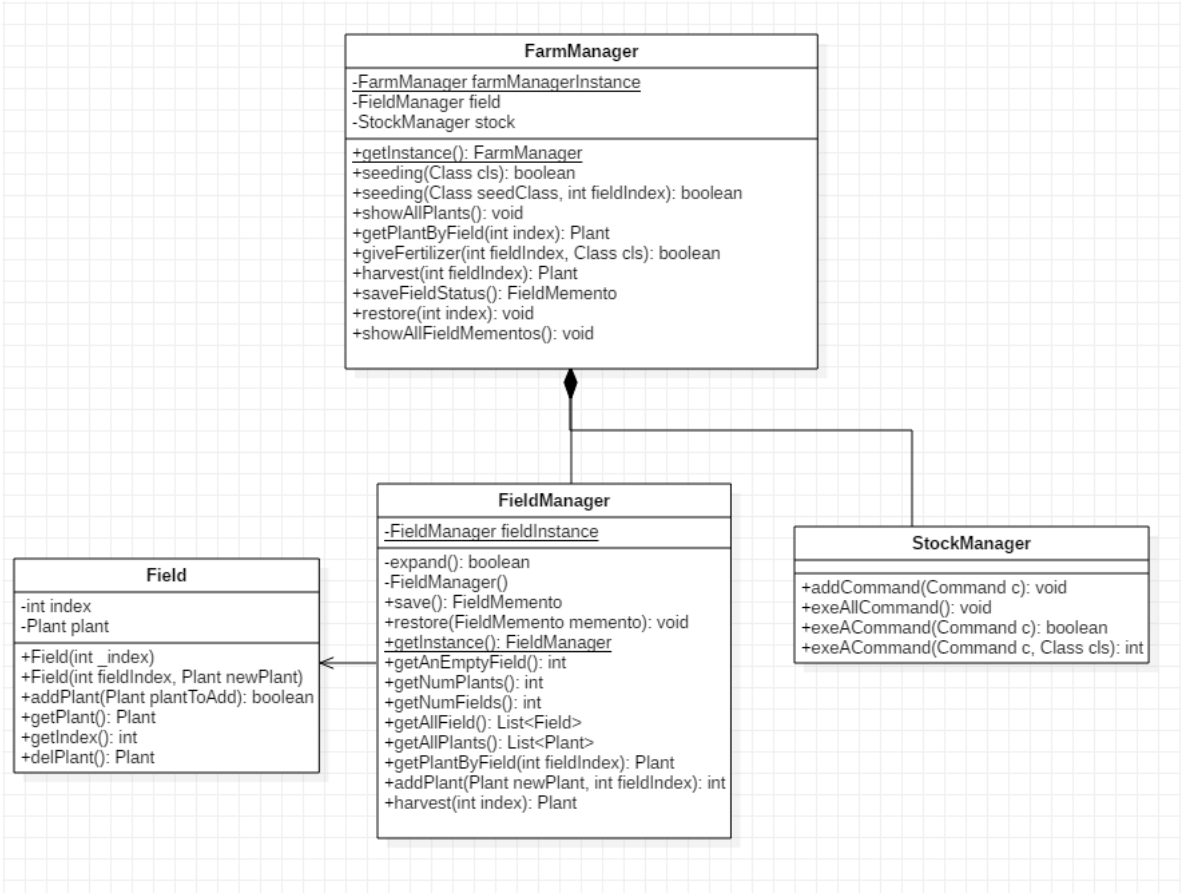
设计模式简述

Composite模式属于结构型模式的一种，该模式将对象组合成树形结构，以此来表示对象的"部分-整体"的层次结构。优点是提升了代码的复用性；在逻辑上将树结构中的根节点和叶子节点做同化处理，使高层模块的调用更加简单；经过重写基类Component中的行为函数runAction()，可以使单个对象和复杂对象的使用具有一致性；可以自由的增加节点。但是也会使逻辑会变得更加复杂；叶子节点和根节点都是实现类，而不是接口，违反了依赖倒置原则。

3.7.1 API描述

我们在种植这一过程中使用Composite设计模式，Filed类中包含Plant，FieldManager类中包含Field。FieldManager和StockManager组成FarmManager。

3.7.2 类图



3.8 Decorator Pattern

设计模式简述

装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

装饰器模式相比生成子类更为灵活，能够扩展一个类的功能，并且可以代替继承。

3.8.1 API 描述

羊可以产出羊毛. 而羊毛可以被染成其他的颜色. 这里认为是涂料与羊毛本身是无关的, 两者更像是附着在一起的关系, 而不是认为颜色是羊毛自己的属性.

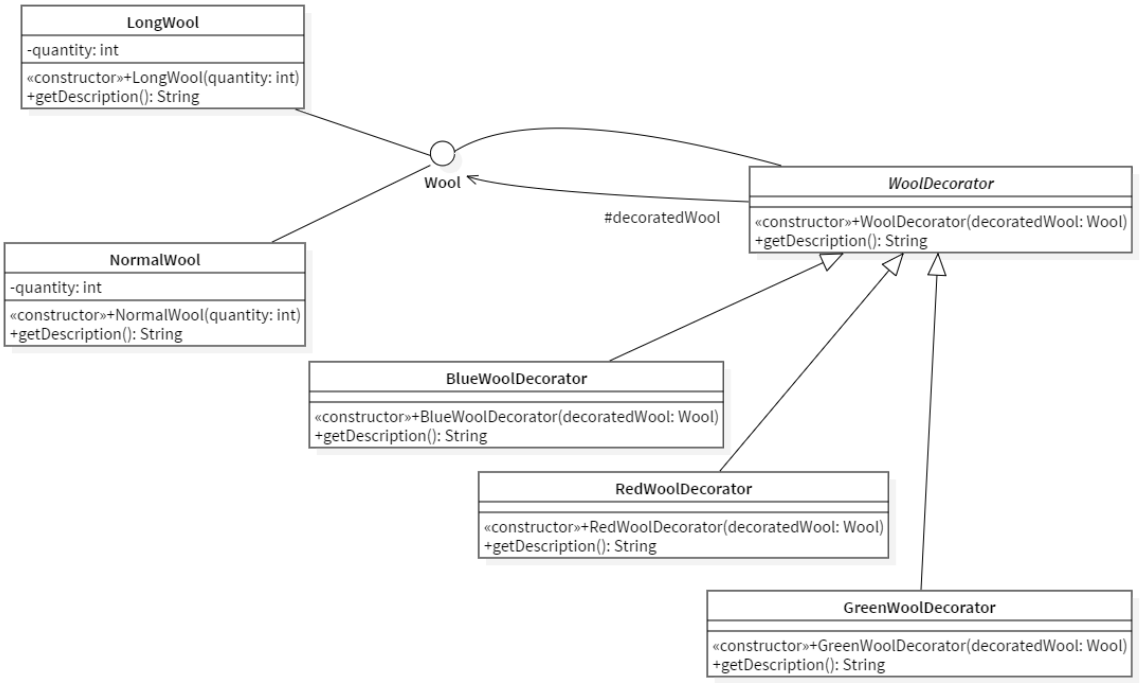
函数名	作用
`getDescription()`	获取羊毛(或者装饰后的)的信息

3.8.2 使用例

```
System.out.println("Now start dyeing...");
System.out.println(
        new RedWoolDecorator(
                new BlueWoolDecorator(
                        new GreenWoolDecorator(longWool)
                )
        ).getDescription()
);
```

```
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.
```

3.8.3 类图



3.9 Facade Pattern

设计模式简述

外观模式（Facade Pattern）隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性。这种模式涉及到一个单一的类，该类提供了客户端请求的简化方法和对现有系统类方法的委托调用。为子系统中的一组接口提供一个一致的界面，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使用。在层次化结构中，可以使用外观模式定义系统中每一层的入口。

3.9.1 API 描述

动物要进行排泄. 不同动物排泄的方式不同. 这里实现了一个`AnimalPoo`接口, 以及继承自此接口的`CatPoo`类和`ChickenPoo`以及`SheepPoo`. 同时实现了一个外观类`AnimalPooMaker`, 所有的调用均通过其进行.

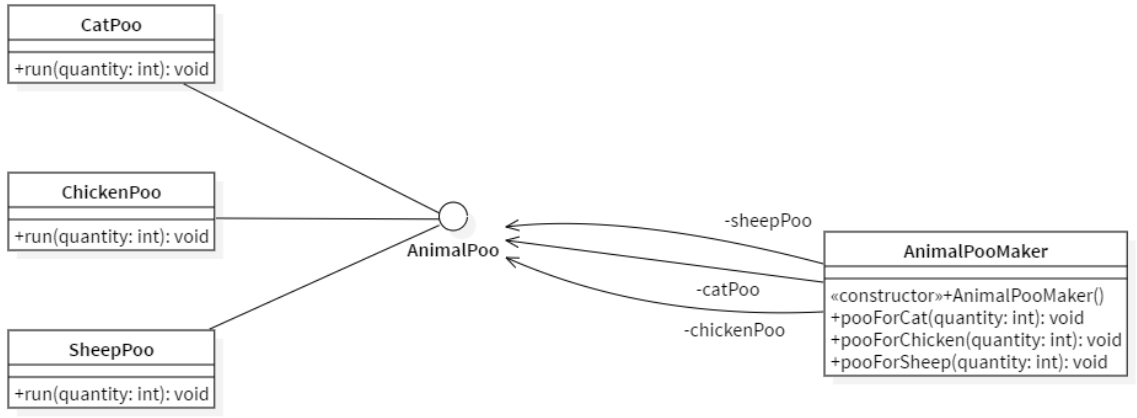
函数名	作用
`run()`	进行排泄
`pooForCat()`	使猫进行排泄
`pooForChicken()`	使鸡进行排泄
`pooForSheep()`	使羊进行排泄

3.9.2 使用例

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

```
The cat poo 2 cat stool.
The chicken poo 1 chicken stool.
The sheep poo 1 sheep stool.
```

3.9.3 类图



3.10 Factory Method Pattern

设计模式简述

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。

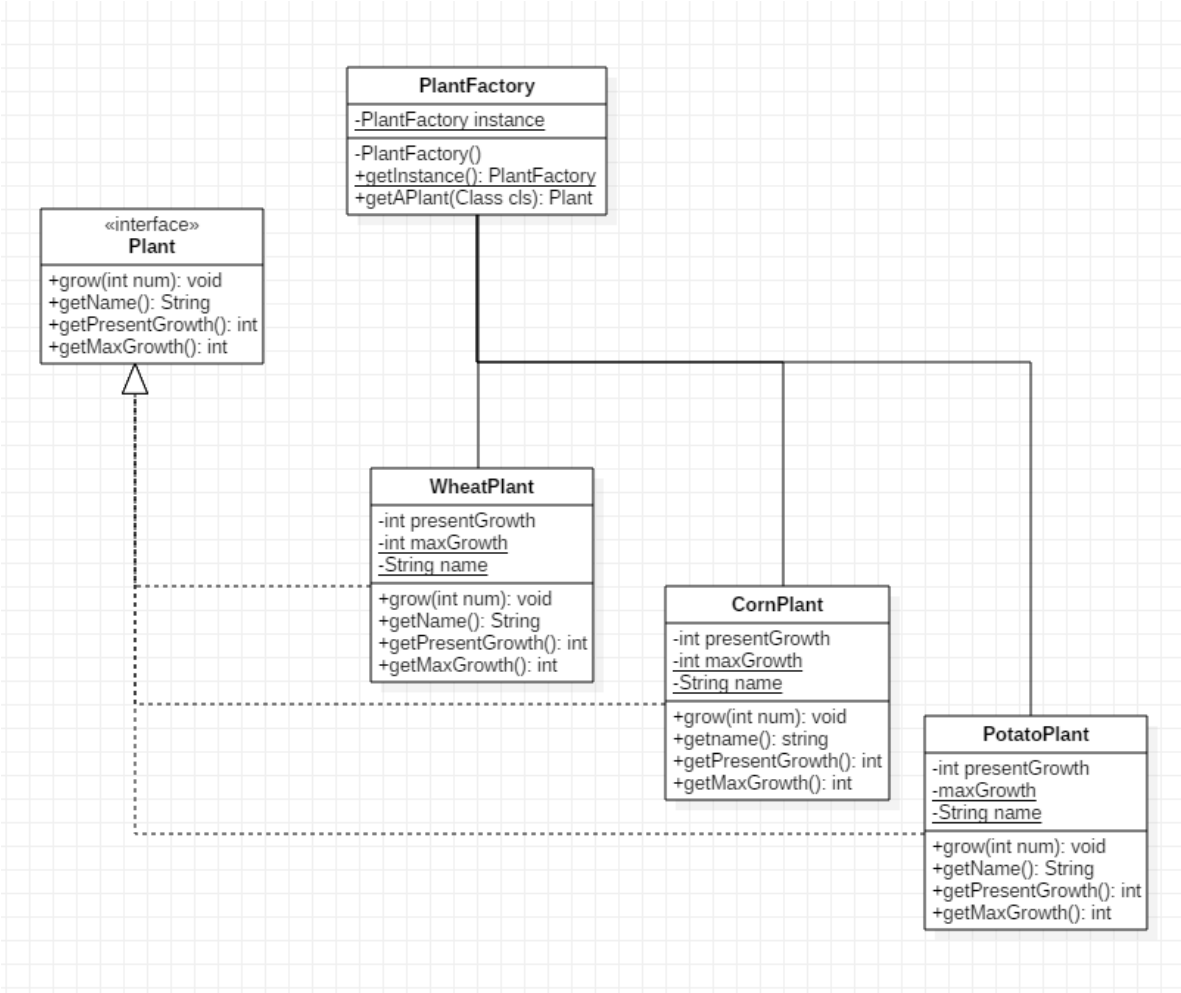
在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

3.10.1 Farm 实现API

3.10.1.1 API描述

我们将该模式运用于植株的种植过程，PlantFactory实现了 Factory Method模式，判断输入后可以在grow();函数中生产3种不同的plant：Corn、Potato、Wheat。

3.11.1.2 类图



3.10.2 Pasture 实现API

3.10.2.1 API 描述

动物工厂根据参数的不同, 创建不同的动物.

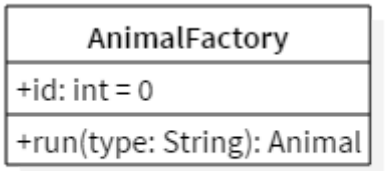
函数名	作用
`run()`	返回一个动物

3.10.2.2 使用例

```
AnimalFactory factoryDemo = new AnimalFactory();
Animal cat = factoryDemo.run("Cat");
Animal sheep = factoryDemo.run("Sheep");
Animal chicken = factoryDemo.run("Chicken");
```

```
Create a new cat!
Create a new sheep!
Create a new chicken!
```

3.10.2.3 类图

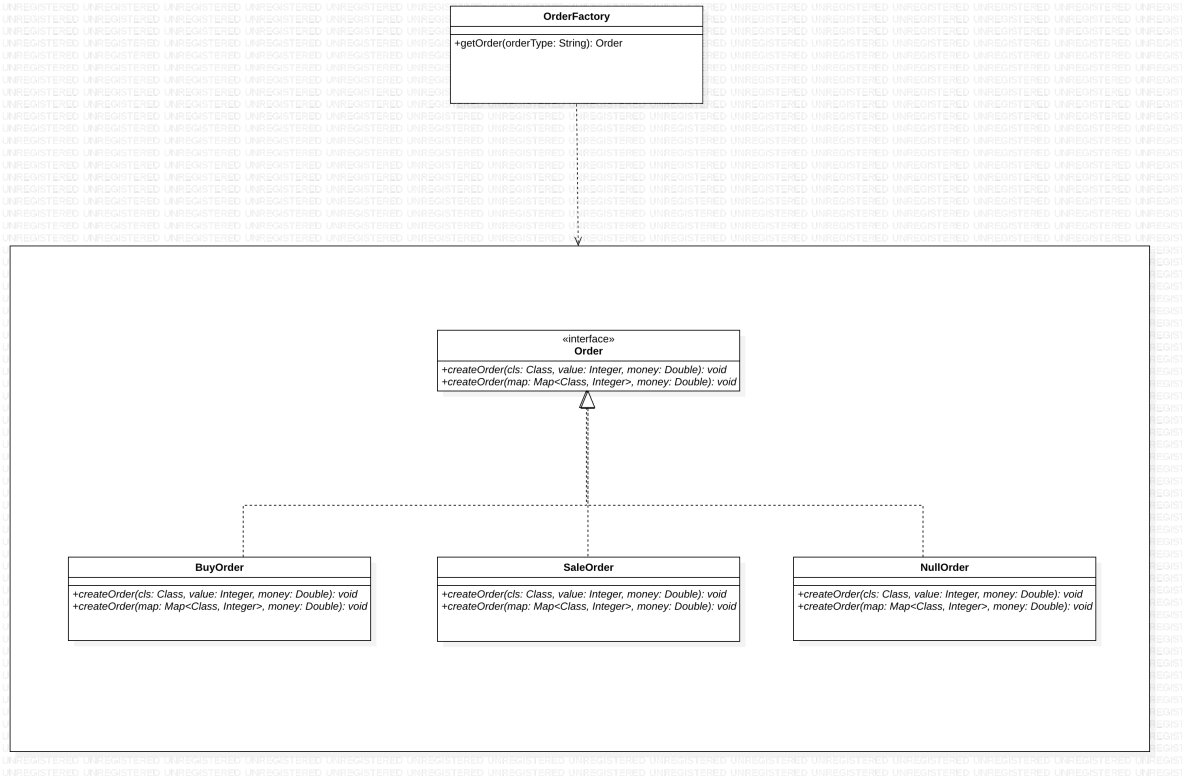


3.10.3 Shop 实现API

3.10.3.1 API 描述

我们将该模式运用于订单的创建过程中，判断输入后可以在getOrder()中通过参数“Buy”或“Sale”创建BuyOrder或SaleOrder; NullOrder用于处理输入错误的情况。

3.10.3.2 类图



3.11 Flyweight Pattern

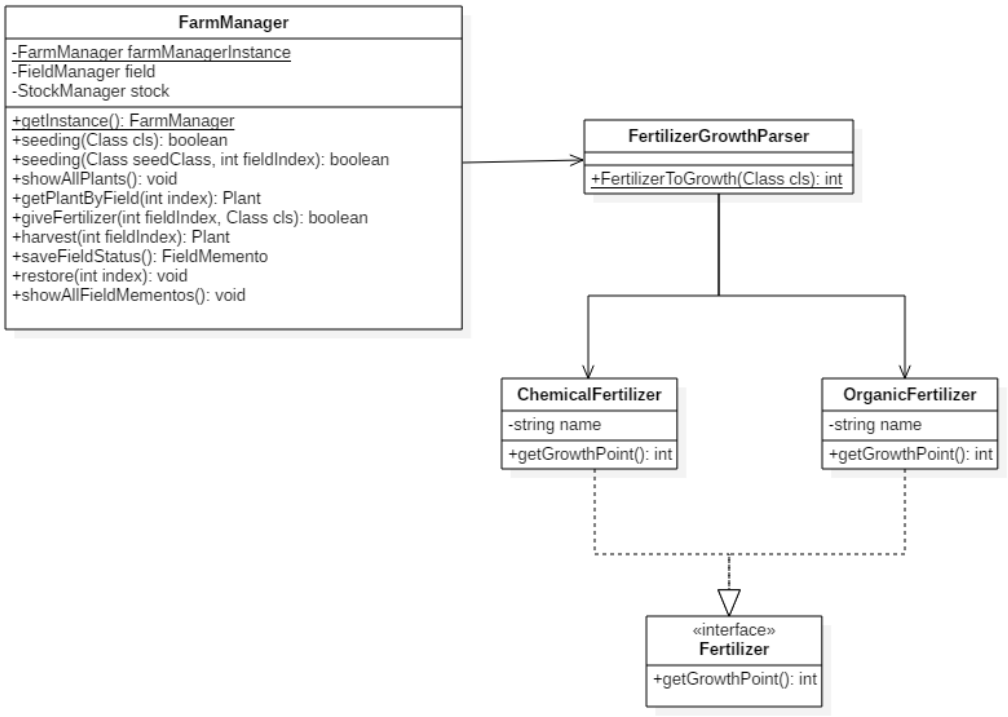
设计模式简述

享元模式（Flyweight Pattern）是说在有大量对象时，有可能会造成内存溢出，我们把其中共同的部分抽象出来，如果有相同的业务请求，直接返回在内存中已有的对象，避免重新创建。使用这个设计模式时，大大减少了对象的创建，降低系统的内存，使效率提高。但是提高了系统的复杂度，容易造成系统的混乱。

3.11.1 API描述

通过FertilizerGrowthParser类保存OrganicFertilizer的一个实例和ChemicalFertilizer的一个实例，在每次通过FarmManager进行施肥操作时创建对这两个实例的引用。

3.11.2 类图



3.12 Interpreter Pattern

设计模式简述

解释器模式（Interpreter Pattern）提供了评估语言的语法或表达式的方式，它属于行为型模式。这种模式实现了一个表达式接口，该接口解释一个特定的上下文。这种模式被用在 SQL 解析、符号处理引擎等。给定一个语言，定义它的文法表示，并定义一个解释器，这个解释器使用该标识来解释语言中的句子。

3.12.1 API描述

在这个项目中，有很多重复的加法功能，如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。

函数名	作用
void interpret()	实现解释器的具体操作

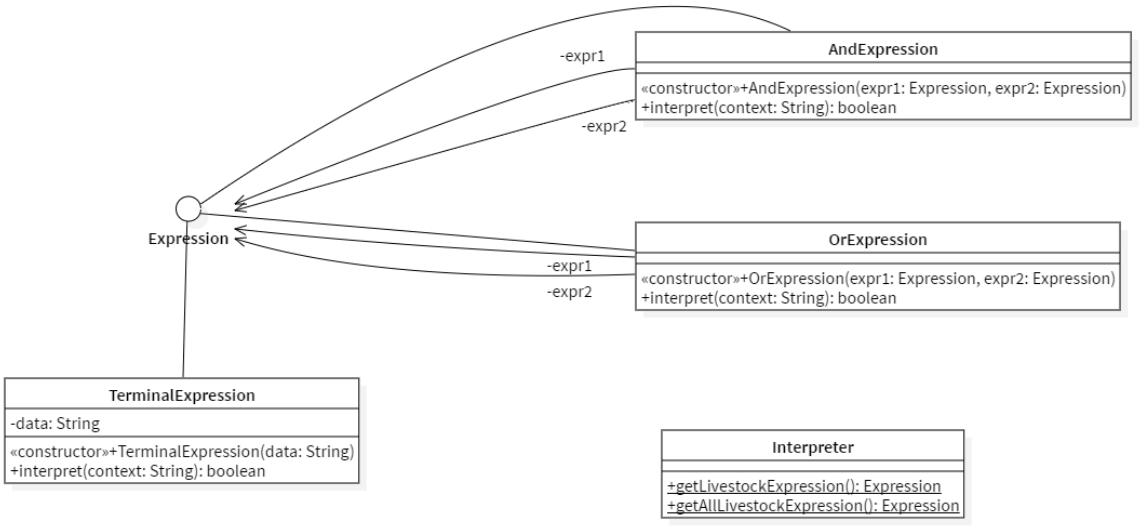
3.12.2 使用例

```
Expression oneLivestock = Interpreter.getLivestockExpression();
Expression allLivestock = Interpreter.getAllLivestockExpression();

System.out.println("Chicken is livestock? " +
oneLivestock.interpret("Chicken"));
System.out.println("Sheep and chicken are all the livestock in pasture? "
        + allLivestock.interpret("Sheep Chicken"));
```

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

3.12.3 类图



3.13 Iterator Pattern

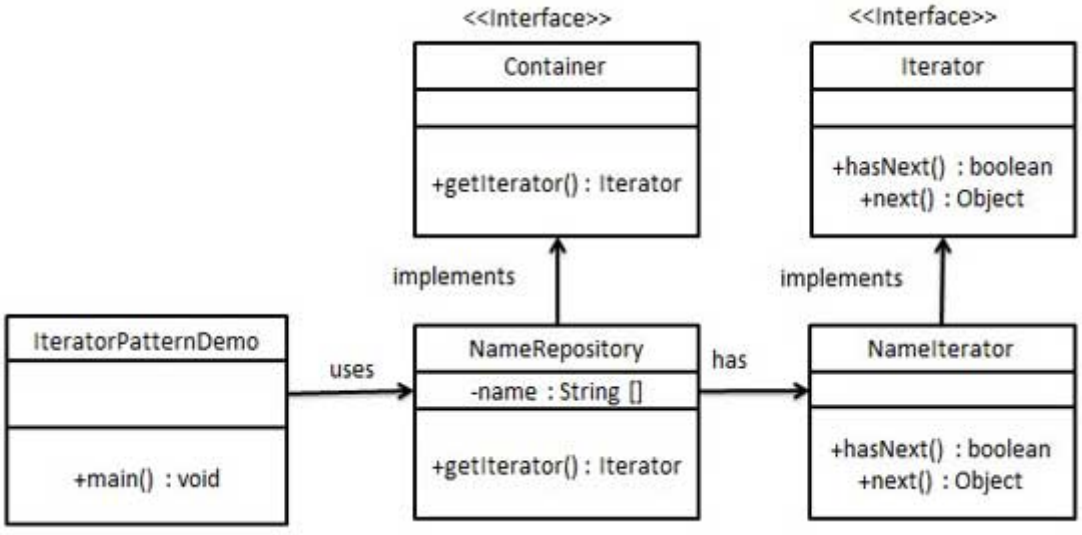
设计模式简述

迭代器模式（Iterator Pattern）是 Java 和 .Net 编程环境中非常常用的设计模式。这种模式用于顺序访问集合对象的元素，不需要知道集合对象的底层表示。迭代器模式属于行为型模式。提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。

3.13.1 API描述

在RepositoryImpl类中，传入参数为Map类型的add、ask方法，处理Map使用的是迭代器模式，通过迭代器遍历整个字典，对于每一个迭代对象作为参数分别调用一次参数为<Class, Integer>的add/ask方法。

3.13.2 类图



3.14 Mediator Pattern

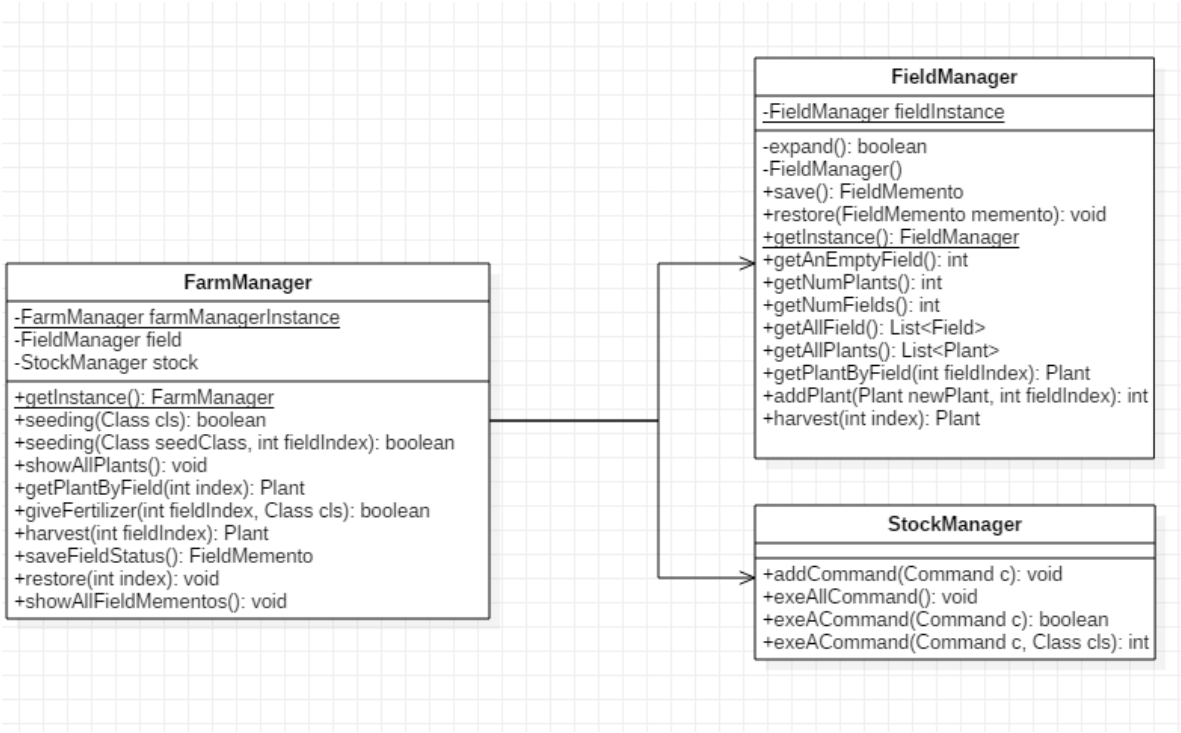
设计模式简述

Mediator解耦多个同事对象之间的复杂交互关系。创建中介者，每个同时对象都用与中介者的交互来替代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。

3.14.1 API描述

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

3.14.2 类图



3.15 Memento Pattern

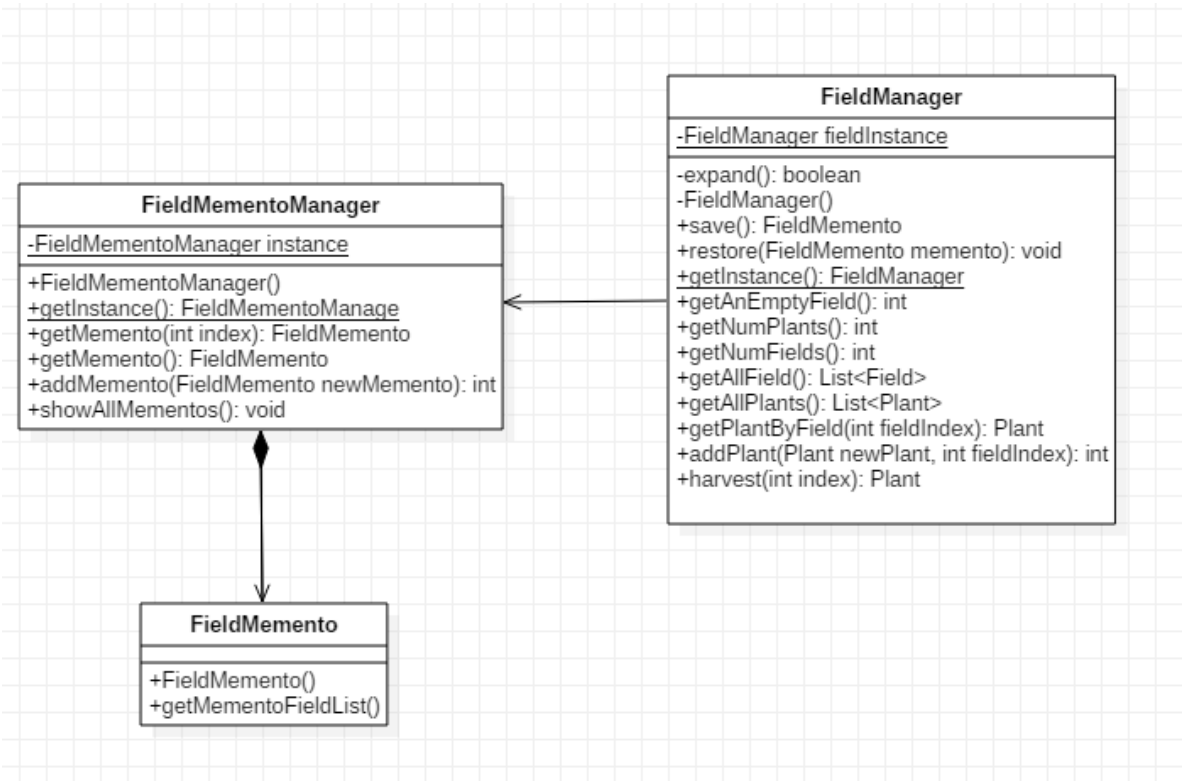
设计模式简述

备忘录模式就是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以在以后将对象恢复到原先保存的状态。客户不与备忘录类耦合，与备忘录管理类耦合。

3.15.1 API 描述

备忘录模式就FieldMemento类，用于存储植物的当前状态，每一个植物都对应一个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。

3.15.2 类图



3.16 Observer Pattern

设计模式简述

当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，当一个对象被修改时，则会自动通知它的依赖对象。观察者模式属于行为型模式。定义对象间的一种一对多的依赖关系，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。

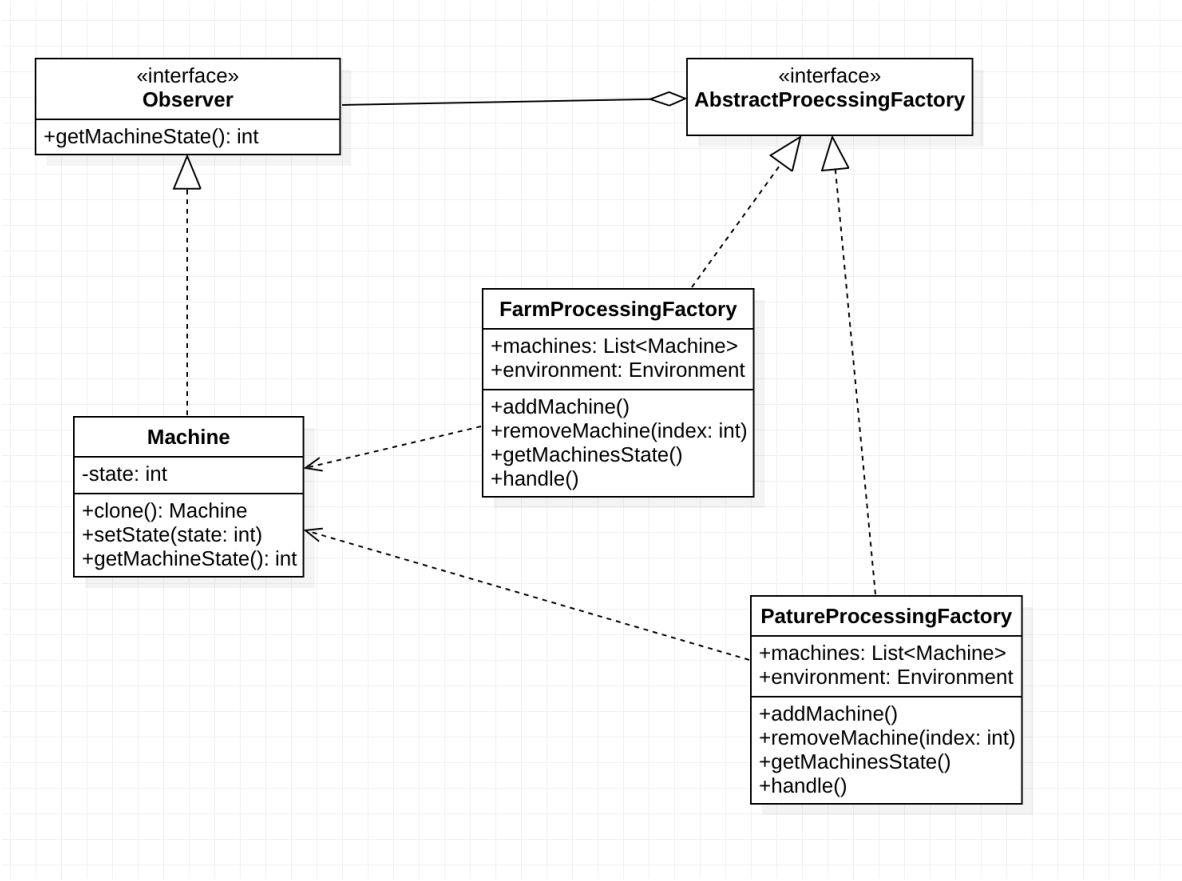
3.16.1 Factory实现API

3.16.1.1 API 描述

每隔一段时间工厂需要检查工厂中机器的运行情况，机器为观察者，当工厂调用getMachinesState()方法时，每台机器都会向工厂报告运行情况。

函数名	作用
int getMahcineState()	接口定义方法，获取机器的状态
int getMahcineState()	实现接口定义方法，获取机器的状态
Void setState()	返回机器的状态

3.16.1.2 类图



3.16.2 Pasture实现API

3.16.2.1 API 描述

当有动物逃跑时, 会有电报和电话通知.

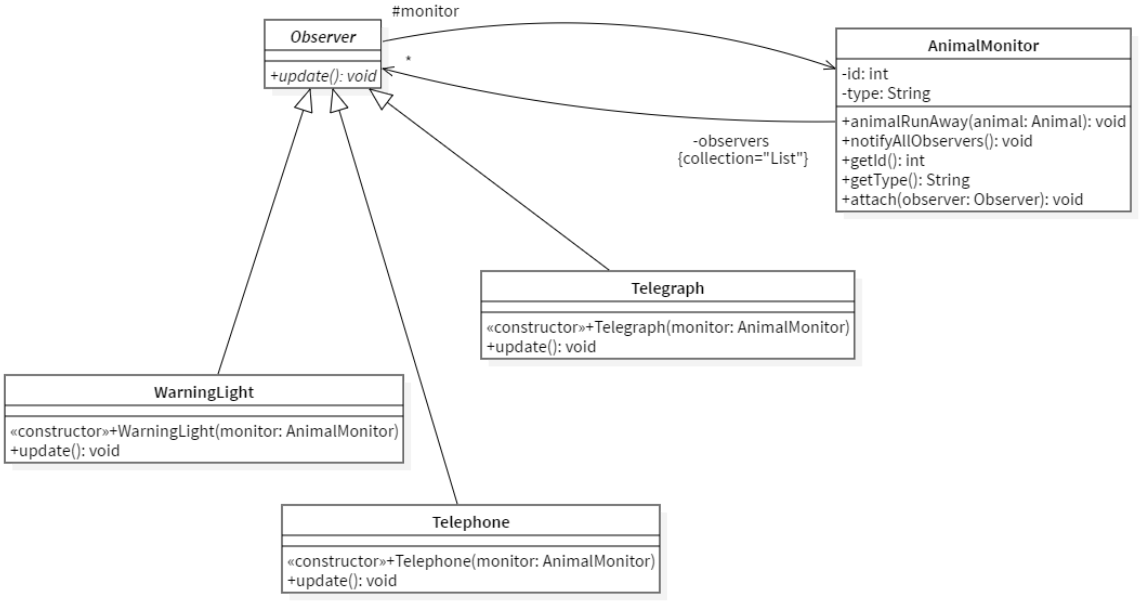
函数名	作用
`animalRunAway()`	使一个动物逃跑

3.16.2.2 使用例

```
AnimalMonitor monitor = new AnimalMonitor();
new WarningLight(monitor);
new Telegraph(monitor);
new Telephone(monitor);
monitor.animalRunAway(chicken);
```

```
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!
```

3.16.2.3 类图



3.16.3 Shop实现API

3.16.3.1 API 描述

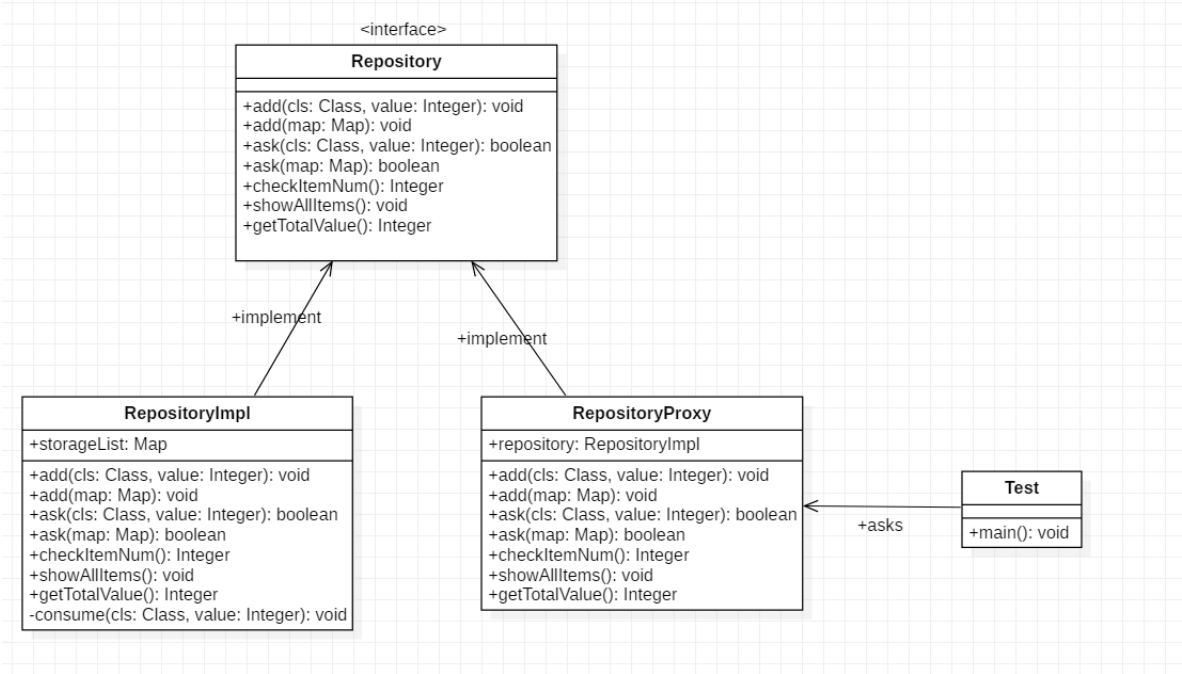
Store类拥有自身所观察的被观察对象，当自身的update方法被调用时根据被观察对象的状态作出相应变更

方法名	描述
update	根据被观察对象的状态作出相应变更

RepositoryProxy类拥有一个观察者列表，当自身状态变化时调用Notify方法，来通知每一个观察者

方法名	描述
attach	以观察者为参数，将观察者加入观察者列表
NotifyAllObservers	调用每一个观察者的update方法

3.16.3.2 类图



3.17 Prototype Pattern

设计模式简述

原型模式（Prototype Pattern）是用于创建重复的对象，同时又能保证性能。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。

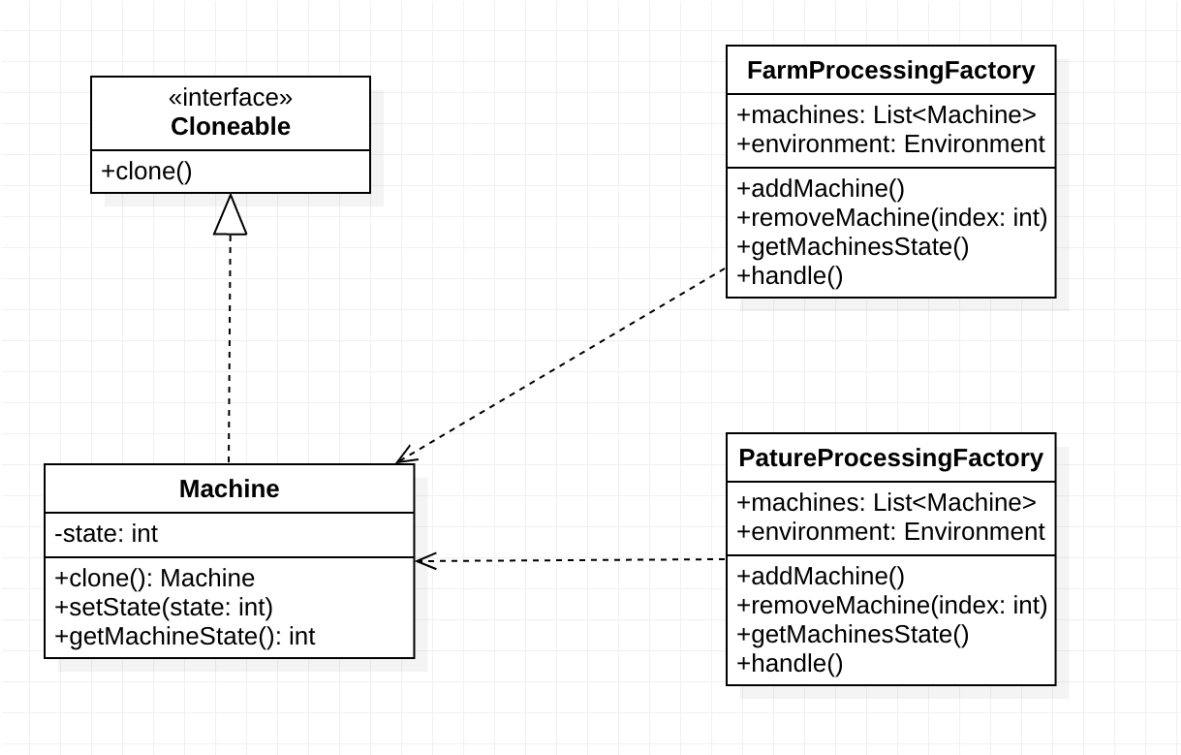
这种模式是实现了一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。例如，一个对象需要在一个高代价的数据库操作之后被创建。我们可以缓存该对象，在下一个请求时返回它的克隆，在需要的时候更新数据库，以此来减少数据库调用。模式即将抽象部分与它的实现部分分离开来，使他们都可以独立变化。桥接模式将继承关系转化成关联关系，它降低了类与类之间的耦合度，减少了系统中类的数量，也减少了代码量。

3.17.1 API 描述

在工厂调用getMachinesState()时，系统会产生一个随机数来修改Machine的state属性来模拟机器损坏的情况，当机器损坏时，工厂会移除该机器，然后调用Machine类的clone()方法来产生一个新的机器。

函数名	作用
Machine clone()	重写接口方法，返回机器的一个复制对象

3.17.2 类图



3.18 Proxy Pattern

设计模式简述

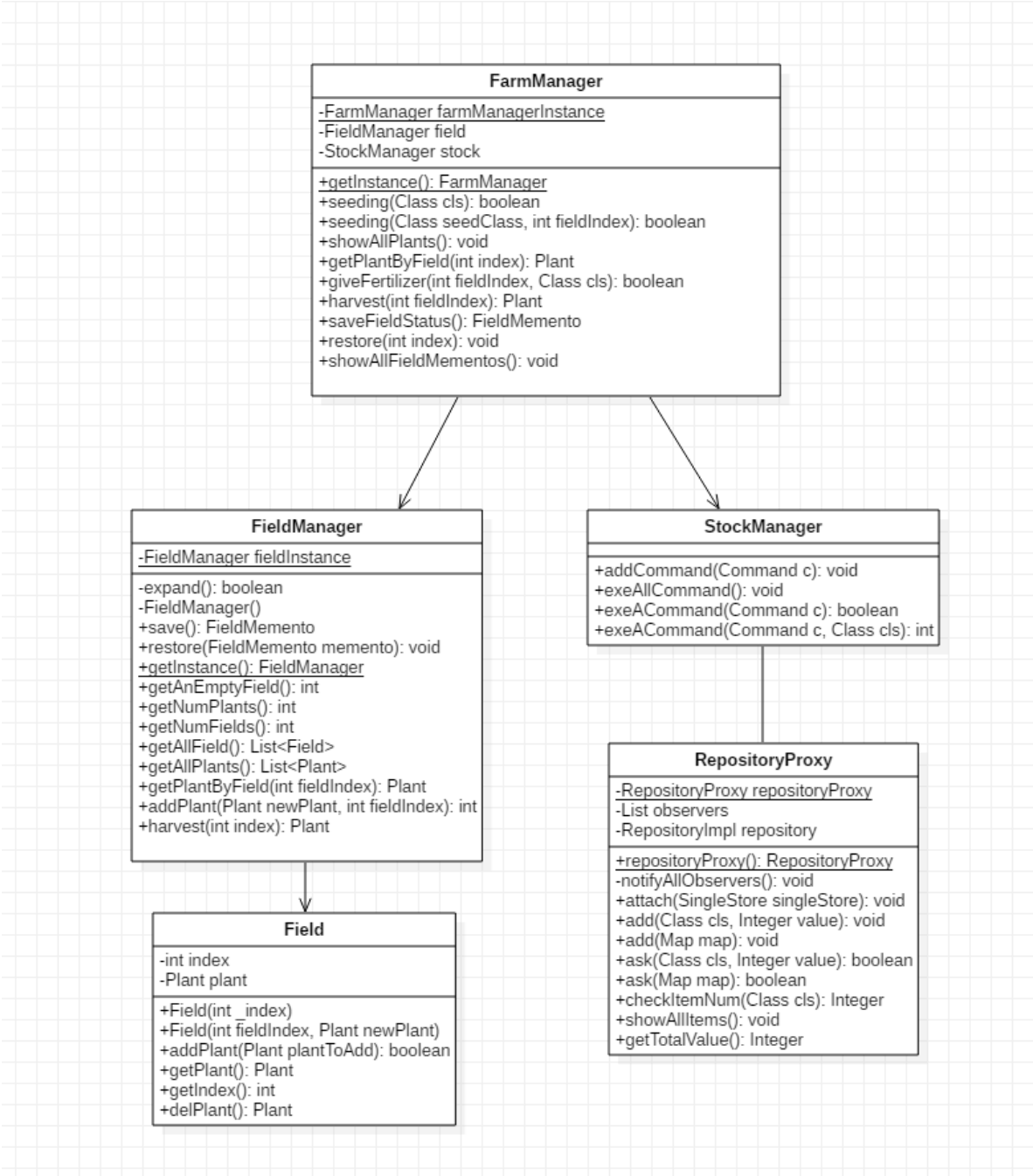
代理模式（Proxy Pattern）中，一个类代表另一个类的功能。这种类型的设计模式属于结构型模式。在代理模式中，我们创建具有现有对象的对象，以便向外界提供功能接口。为其他对象提供一种代理以控制对这个对象的访问。

3.18.1 Farm实现API

3.18.1.1 API 描述

在农场的设计中，我指派PlantFactory和Field进行工作，而FarmManager本身无需工作，此时FarmManager代理所有PlantFactory和Field的所有操作，且无法在farm包外访问。

3.18.1.2 类图



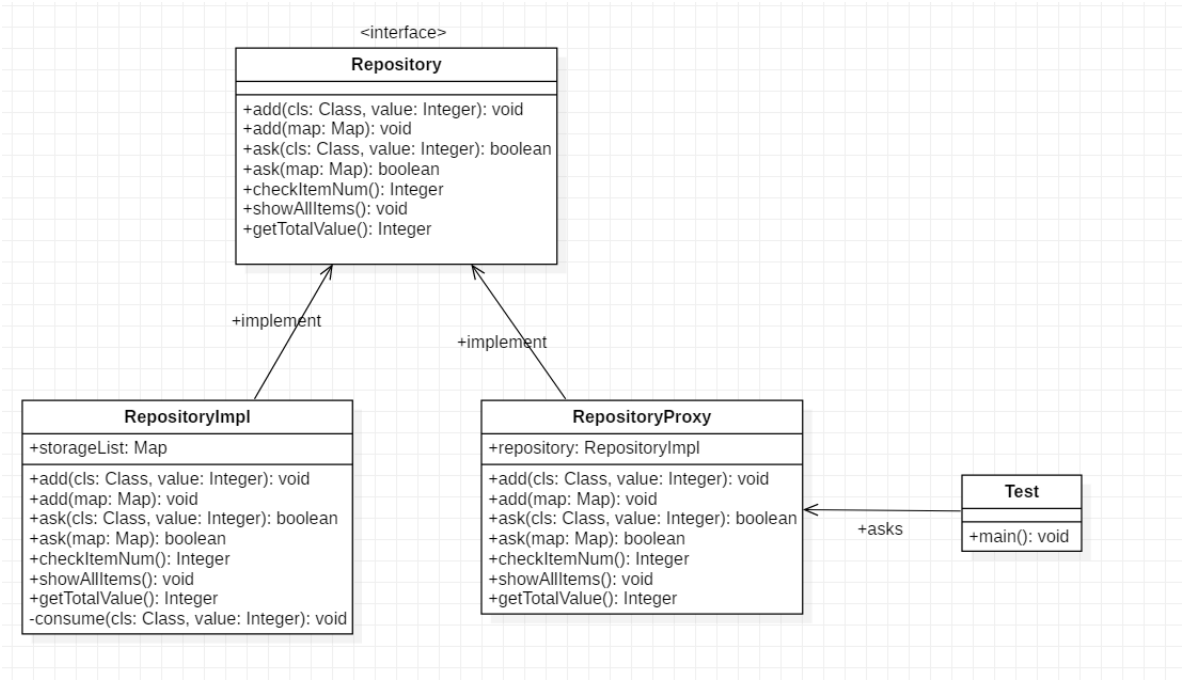
3.18.2 Shop实现API

3.18.2.1 API 描述

以RepositoryProxy代理类来代理RepositoryImpl类，两者都是Repository接口的实现类。

方法名	描述
Instance	返回代理类的单例
add	以类名与整数值，或者一个字典为参数，向仓库中添加对应项
ask	以类名与整数值，或者一个字典为参数，向仓库请求使用对应项，如果剩余库存充足则直接消耗，返回true，否则返回false
showAllItems	在控制台输出仓库中储存的所有项与对应存量
checkItemNum	以类名为参数，返回仓库中该类所存储的存量。
getTotalValue	返回仓库中所有项的总存量

3.18.2.2 类图



3.19 Singleton Pattern

设计模式简述

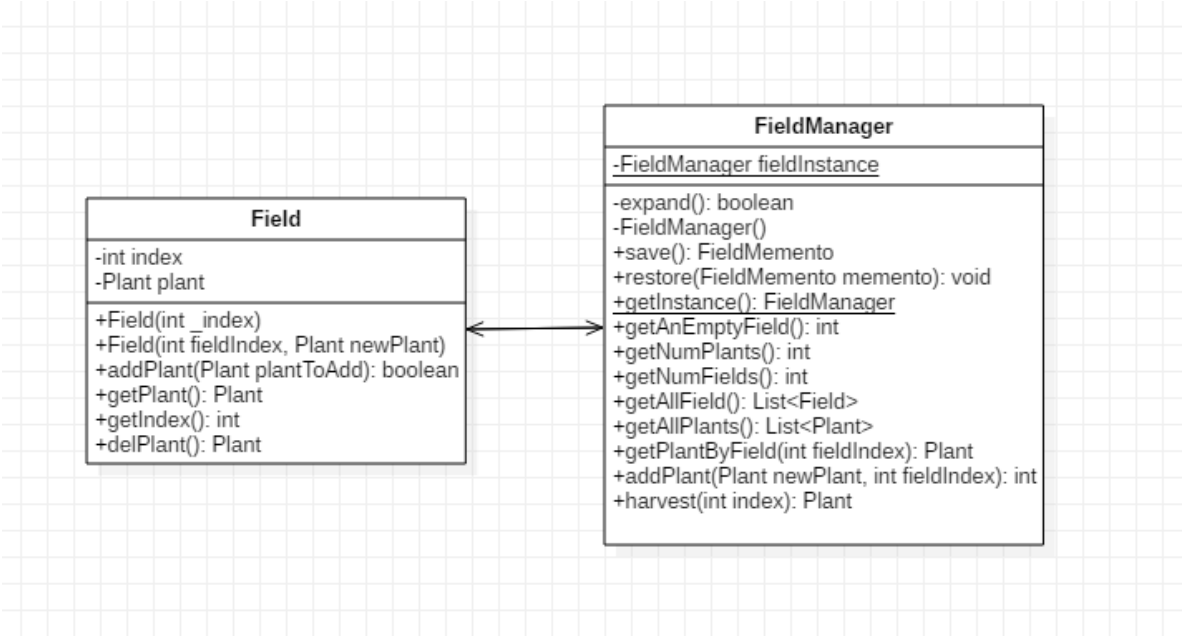
Singleton Pattern涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。主要意图是保证一个类仅有一个实例，并提供一个访问它的全局访问点。并且解决一个全局使用的类频繁地创建与销毁的问题。

3.19.1 Farm实现API

3.19.1.1 API 描述

实现一个Field类，该田地在全局只有一个。主要函数是addPlant()：添加植物、getPlant()：取得植物、getIndex()：取得这块田的索引和delPlant()：删除植物。

3.19.1.2 类图



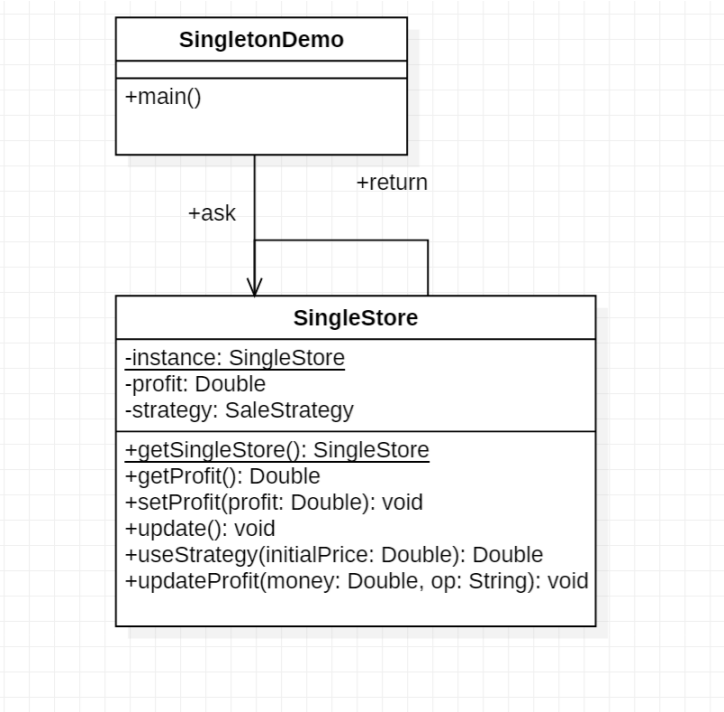
3.19.2 Shop实现API

3.19.2.1 API 描述

我们将SingleStore定义为一个单例。可以通过getSingleStore()方法来获取SingleStore单例。并且构造函数为私有，不可以通过外部创建。

函数名	作用
SingleStore getSingleStore()	获取商店类的唯一实例instance并返回。

3.19.2.2 类图



3.20 State Pattern

设计模式简述

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。优点是在逻辑上确定并枚举可能的状态与该状态下对应的行为方法，将状态与行为做逻辑关联，使得高层模块可以直接通过更改状态实现不同行为的选择；多个环境对象可以共享同一个状态模块，实现了代码的复用。但是在原有的结构基础上增加了不同的状态类与对象类，但在实现方法上得到了精简；对OCP原则不友好，增加新的状态时必不可少的要更改切换状态的源代码。

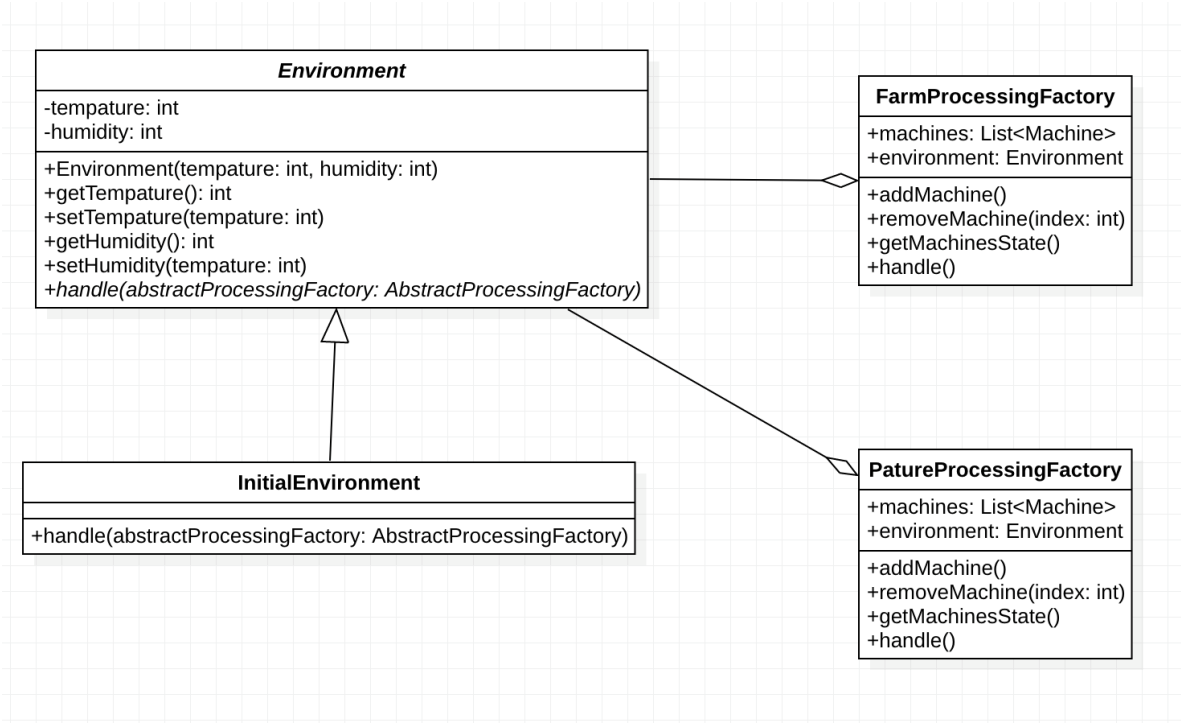
3.20.1 Factory实现API

3.20.1.1 API 描述

我们将外界环境定义为温度和湿度，工厂持有一个外界环境类，当外界环境发生变化时，工厂会调用外界环境类的handle()方法来调整工厂内的温度，湿度。

方法	作用
abstract void handle(..)	抽象方法，在实现类中实现该方法来处理外界环境的变化
Int getTemerature()	返回环境的温度
Void setTemperature()	设置环境的温度
Int getHumidity()	返回环境的湿度
Void setHumidity()	设置环境的湿度

3.20.1.2 类图



3.20.2 Shop实现API

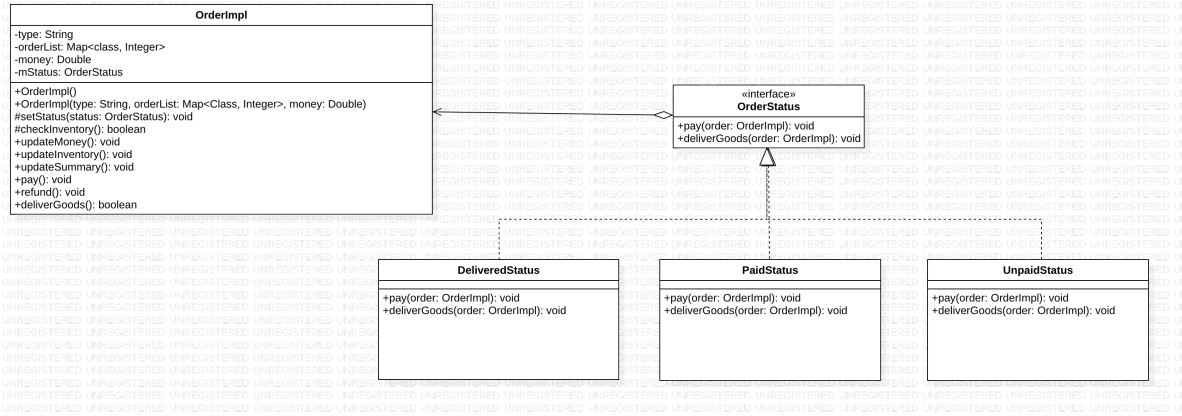
3.20.2.1 API 描述

我们以OrderState为基本状态接口，为植物类（OrderImpl）设立了三种状态：

类名	描述
DeliveredState	商品已交付阶段
PaidState	已支付阶段
UnpaidState	未支付阶段

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。

3.20.2.2 类图



3.21 Strategy Pattern

设计模式简述

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。

此设计模式分离具体的算法和客户端，使得客户端可以自由切换算法，算法也可以独立于客户端自由进行更改；避免在同一算法中出现大量的条件判断，而是将原本逻辑复杂的算法拆分成多个结构相对简单的独立算法；算法可扩展性良好。但是在结构框架中需要实例化每一个新的策略类，且需要对外暴露所有的策略，复杂化了结构。

3.21.1 Pasture 实现API

3.21.1.1 API描述

不同的动物会有不同的动作

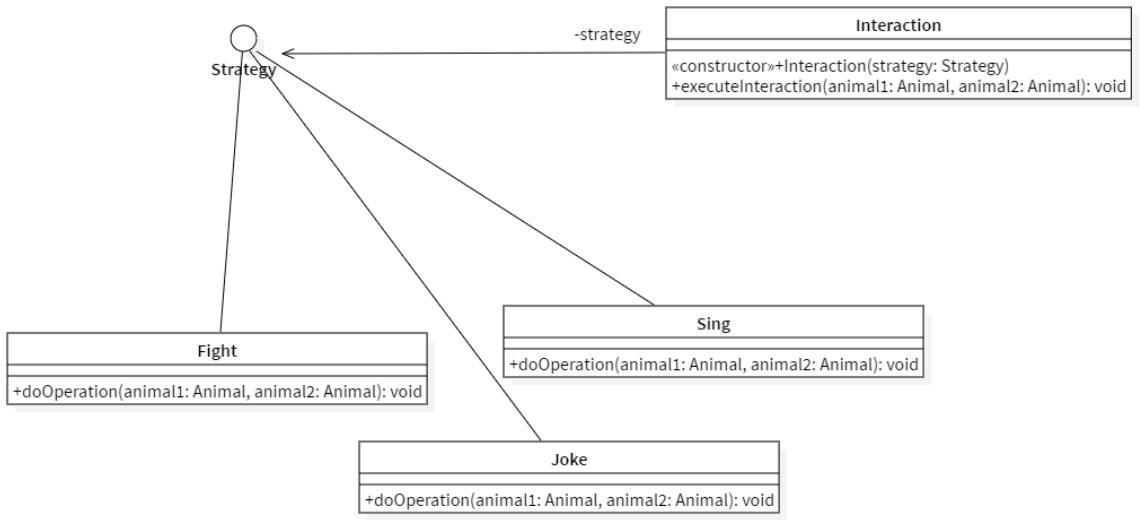
类名	描述
`executeInteraction()`	执行动作

3.21.1.2 使用例

```
Interaction sing = new Interaction(new Sing());
sing.executeInteraction(cat, sheep);
Interaction joke = new Interaction(new Joke());
joke.executeInteraction(sheep, chicken);
Interaction fight = new Interaction(new Fight());
fight.executeInteraction(chicken, cat);
```

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

3.21.1.3 类图



3.21.2 Shop 实现API

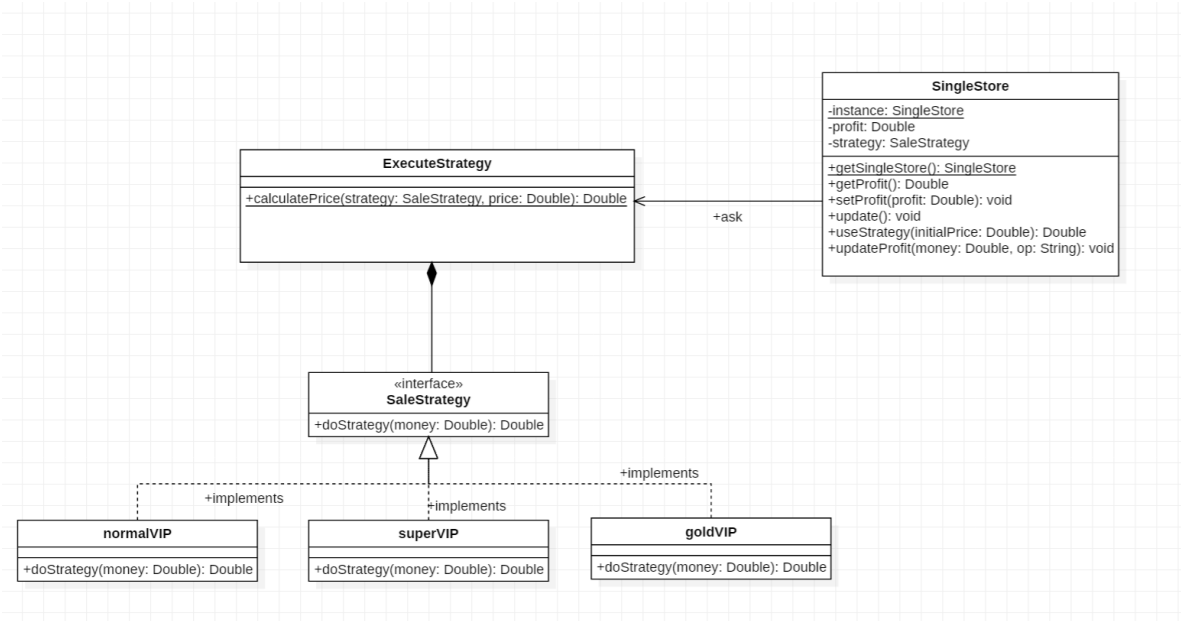
3.21.2.1 API 描述

商店出售商品时选择了三种优惠方式：normalVIP，goldVIP，superVIP分别对订单总价采用不同的折扣来进行销售，订单的选择根据仓库的反馈来进行更新，当仓库库存接近溢出时采取较大的折扣，当仓库库存不足时，采用较小的折扣。

函数名	作用
Double doStrategy(Double money)	具体计算使用优惠策略后所需要支付的金额
Double calculatePrice(SaleStretagy stretagy,Double price)	调用strategy的doStretagy（）方法来执行策略
Double useStretagy(Double initialPrice)	商店实例通过输入原价来获取使用优惠策略后的价格

类名	描述
normalVIP	普通会员打九折，在库存接近空时采用
goldVIP	黄金会员打八折，在库存处于正常范围时使用
superVIP	超级会员打七折，在库存接近饱和时使用

3.21.2.2 类图



3.22 Template Method

设计模式简述

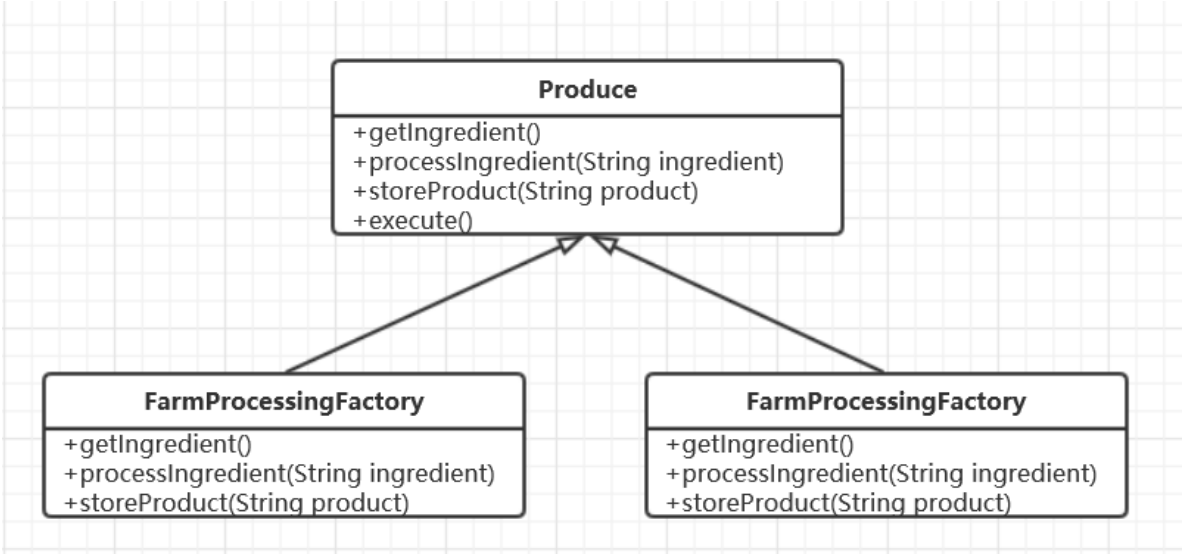
Template Method提供了一种在父类中定义处理流程，在子类中具体实现的处理方式，同时在具体实现时Template Method又允许子类重新定义流程的具体步骤。优点是实现了反向控制和OCP原则，既提高了代码的复用性，又可以便捷的扩展子类群（扩展性），实现无限的可能性。这个设计模式虽然提高了代码的复用性，但是每一个不同的实现都需要一个新的子类实现，从而提升了系统的复杂度。

3.22.1 API描述

通过模板模式在工厂中加工商品，使用抽象类Produce定义了加工商品的三个基本步骤，取出原材料，加工产品，将产品储存到仓库，以及进行加工操作的方法。在FarmProcessingFactory和PastureProcessingFactory类中重写三个基本步骤的方法实现不同工厂加工产品的方法。

函数名	作用
String getIngredient()	获取原材料的抽象方法
String processIngredient(String ingredient)	加工产品的抽象方法
void storeProduct(String product)	储存产品的抽象方法
void excecute()	通用的模板方法，执行上面三个步骤

3.22.2 类图



3.23 Visitor Pattern

设计模式简述

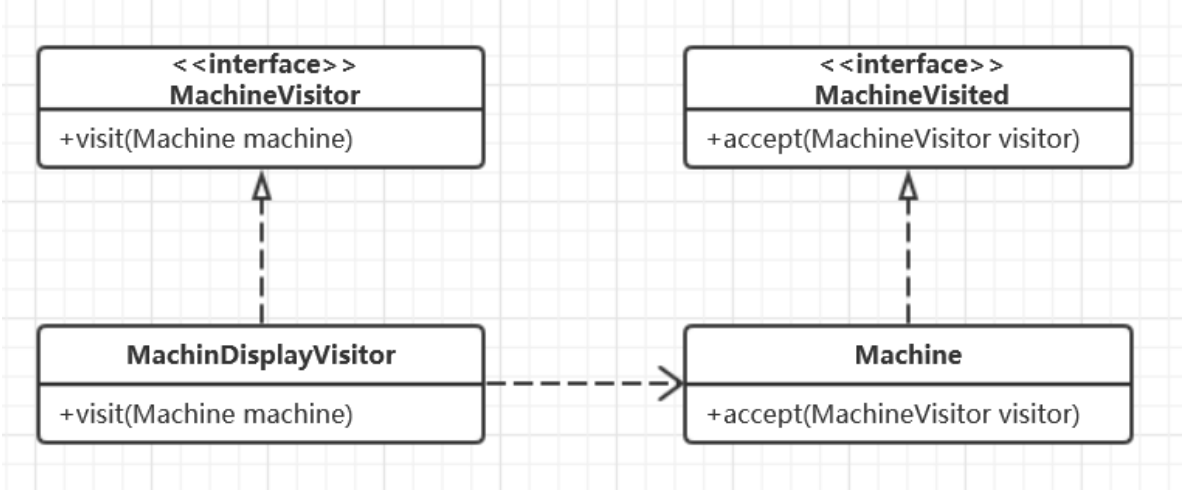
在访问者模式（Visitor Pattern）中，我们使用了一个访问者类，它改变了元素类的执行算法。通过这种方式，元素的执行算法可以随着访问者改变而改变。这种类型的设计模式属于行为型模式。根据模式，元素对象已接受访问者对象，这样访问者对象就可以处理元素对象上的操作。

3.23.1 API描述

定义MachineVisitor接口，定义访问者的操作，定义接受操作的接口MachineVisited。使用MachineDisplayVisitor类实现MachineVisitor接口，定义访问者的操作细节；在Machine类中实现MachineVisited接口执行相应的操作。

函数名	作用
void visit(Machine machine)	定义访问者操作的接口
void accept(MachineVisitor visitor)	定义接受操作的接口

3.23.2 类图



3.24 Null Object Pattern

设计模式简述

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。这样的 Null 对象也可以在数据不可用的时候提供默认的行为。在空对象模式中，我们创建一个指定各种要执行的操作的抽象类和扩展该类的实体类，还创建一个未对该类做任何实现的空对象类，该空对象类将无缝地使用在需要检查空值的地方。

3.24.1 API描述

我们将该模式运用于订单的创建过程中，NullOrder作为Order接口的实现，判断输入后可以在getOrder()中通过参数“Buy”或“Sale”创建BuyOrder或SaleOrder; NullOrder用于处理输入错误的情况。

3.24.2 类图