设计模式项目文档——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

1 项目简介

《天天农场》是由我们小组设计的一套农场题材的体验类农场API，用户可以扮演一个农场的农场主，在自己的农场里进行游戏。

农场主可以雇佣工人，包括采购员、种植员、警卫、维修工、宿舍管理员和建筑工等。农场主可以给各个工人下命令、分配工作：

宿管可以叫醒员工、整理房间内务和运走垃圾；

建筑工可以修建工人居所、接待中心和豪宅。你还可以布置这些建筑的内部，例如添加桌子板凳、安装红外报警器和监控器等；

种植员可以种植、生产各类蔬菜和水果(如苹果、樱桃、土豆、西红柿和豌豆)，养殖各类动物(如鸡、狗和鱼)，田地被分成了若干小块，使你可以细粒度地经营自己的田地。每个小块都可以是荒地、普通土地和黑土地中的任意一个。你还可以将田地的俯视图打印在控制台中，用以观察田地的整体布局；

采购员可以购入、售卖各类农产品；

农场提供各类工具和消耗品，用于方便农夫们和工人们进行各种各样的生产活动和建设活动，同时管理员也能管理这些物品，对它们进行日常的检查、维修、换新等操作；

农场中有看门犬负责整个农场的安全。

在经营自己农场的过程中，用户除了可以体验种植、养殖及收获的过程与乐趣，还可以体验多种农场经典活动，如使用各类工具，植物人工授粉，动物繁殖，加工农产品并售卖等，更有布置工人居所、招待访客、记录日常生活等一系列有趣操作。

总而言之，《天天农场》具有很强的可操作性与参与度，是一套经典的经营策略类游戏API。

2 Design Pattern 汇总表

编号	Design Pattern Name	实现个（套）数	sample programs 个数	备注
1	Abstract Factory	1	1	
2	Adapter	1	1	
3	Bridge	1	1	
4	Builder	1	1	
5	Chain of Responsibility	1	1	
6	Command	1	1	
7	Composite	1	1	
8	Decorator	1	1	
9	Facade	1	1	
10	Factory Method	2	2	
11	Flyweight	1	1	
12	Interpreter	1	1	
13	Iterator	1	1	
14	Mediator	1	1	
15	Memento	1	1	
16	Observer	1	1	
17	Prototype	1	1	
18	Proxy	1	1	
19	Singleton	1	1	
20	State	2	2	
21	Strategy	1	1	
22	Template Method	3	3	
23	Visitor	1	1	
24	AOP	1	1	
25	IOC	1	1	
26	COW	1	1	
总计		31	31	

3 设计模式详述

3.1 Abstract Factory

设计模式简述

桥接模式即以同一界面创建一整族相关或相依的objects，不需点名各对象真正所属的具体类。为访问类提供一个创建一组相关或相互依赖对象的接口，且访问类无须指定所要产品的具体类就能得到同族的不同等级的产品的模式结构。

使用抽象工厂的条件：

- 系统中有多个产品族，每个具体工厂创建同一族但属于不同等级结构的产品。
- 系统一次只可能消费其中某一族产品，即同族的产品一起使用。

优点：

- 可以在类的内部对产品族中相关联的多等级产品共同管理，而不必专门引入多个新的类来进行管理。
- 当增加一个新的产品族时不需要修改原代码，满足开闭原则。

3.1.1 ProductFactory实现API

3.1.1.1 API 描述

从逻辑上来讲，春季工厂和夏季工厂都生产蔬菜和水果，只是二者种植的蔬菜和水果种类不同，将他们共同的特点——产品工厂抽象成父类，并在父类中将生产的蔬菜和水果也抽象化。如此一来，可以在具体的产品工厂中指定要生产的蔬菜及水果种类，同时用户可以方便的切换工厂，以生产不同种类的产品。

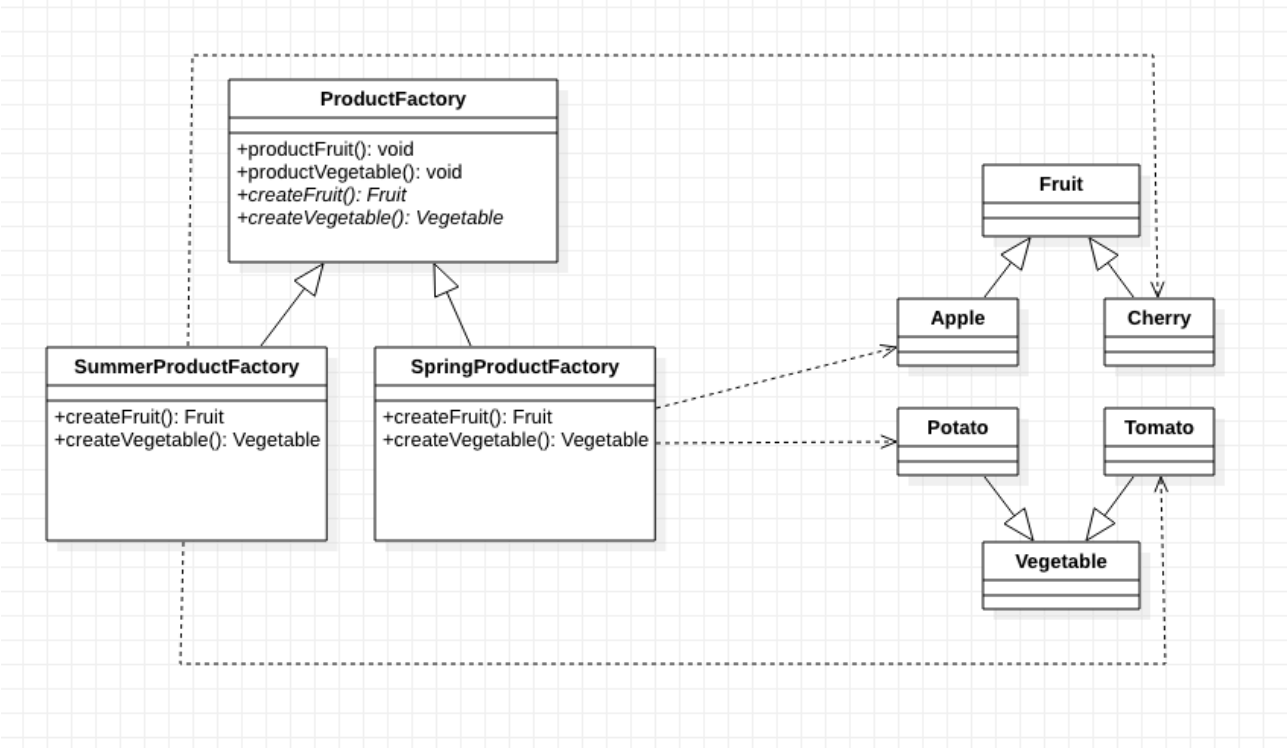
在该API中，为抽象工厂添加蔬菜仓库和水果仓库，将具体工厂生产的响应产品加入到仓库中，方便管理、并易于用户进行操作（如售卖、食用、种植等）。用户可以动态的更改所选的工厂，如果选择的工厂与之前相同，则继续生产该工厂的产品，并加入之前到仓库中；如果用户更换仓库，则生产不同的产品，并把原工厂生产的产品移动到现在的工厂中，从用户角度看，蔬菜仓库和水果仓库存储的是各个季节的蔬菜水果，既与真实场景相同，又便于管理和用户操作。

该API满足开放闭守原则，如果天天农场打算在秋天额外种植柚子和白菜，只需构建AutumnProductFactory，在工厂中种植Grapefruit和Cabbage，并由用户自己选择秋季工厂，不需要修改工厂的内部实现也不需要增加额外的接口，并且一切选择权交由用户自己决定。

函数名	作用
void productFruit()	作为抽象工厂的接口函数。ProductFactory基类中的方法，开放给用户的接口，内部首先创建水果，之后加入水果仓库中，具体的水果在具体的产品工厂中提供。

void productVegetable()	作为抽象工厂的接口函数。ProductFactory基类中的方法，开放给用户的接口，内部首先创建蔬菜，之后加入蔬菜仓库中，具体的蔬菜在具体的产品工厂中提供。
ProductFactory(Vector<Fruit> store_fruits_warehouse, Vector<Vegetable> store_vegetables_warehouse)	用户动态更改工厂时调用，将之前产品工厂中生产的产品移动到新工厂中。由于抽象工厂模式的特性，产品使用抽象类实现，因此不同种类的水果可以放到同一个水果仓库中，蔬菜同理。
void SwitchProductFactory(FactoryKind opcode)	用户动态更改产品工厂的选择 如果工厂之前是空的，则初始化一个新工厂 如果工厂之前生产的是其他东西，把之前的产品保存 如果工厂之前生产的就是当前的东西，则直接return

3.1.1.2 类图



3.2 Adapter

设计模式简述

在现实生活中，经常出现两个对象因接口不兼容而不能在一起工作的实例，这时需要第三者进行适配。在软件设计中也可能出现：需要开发的具有某种业务功能的组件在现有的组件库中已经存在，但它们与当前系统的接口规范不兼容，如果重新开发这些组件成本又很高，这时用适配器模式能很好地解决这些问题。

3.2.1 Watchdog实现API

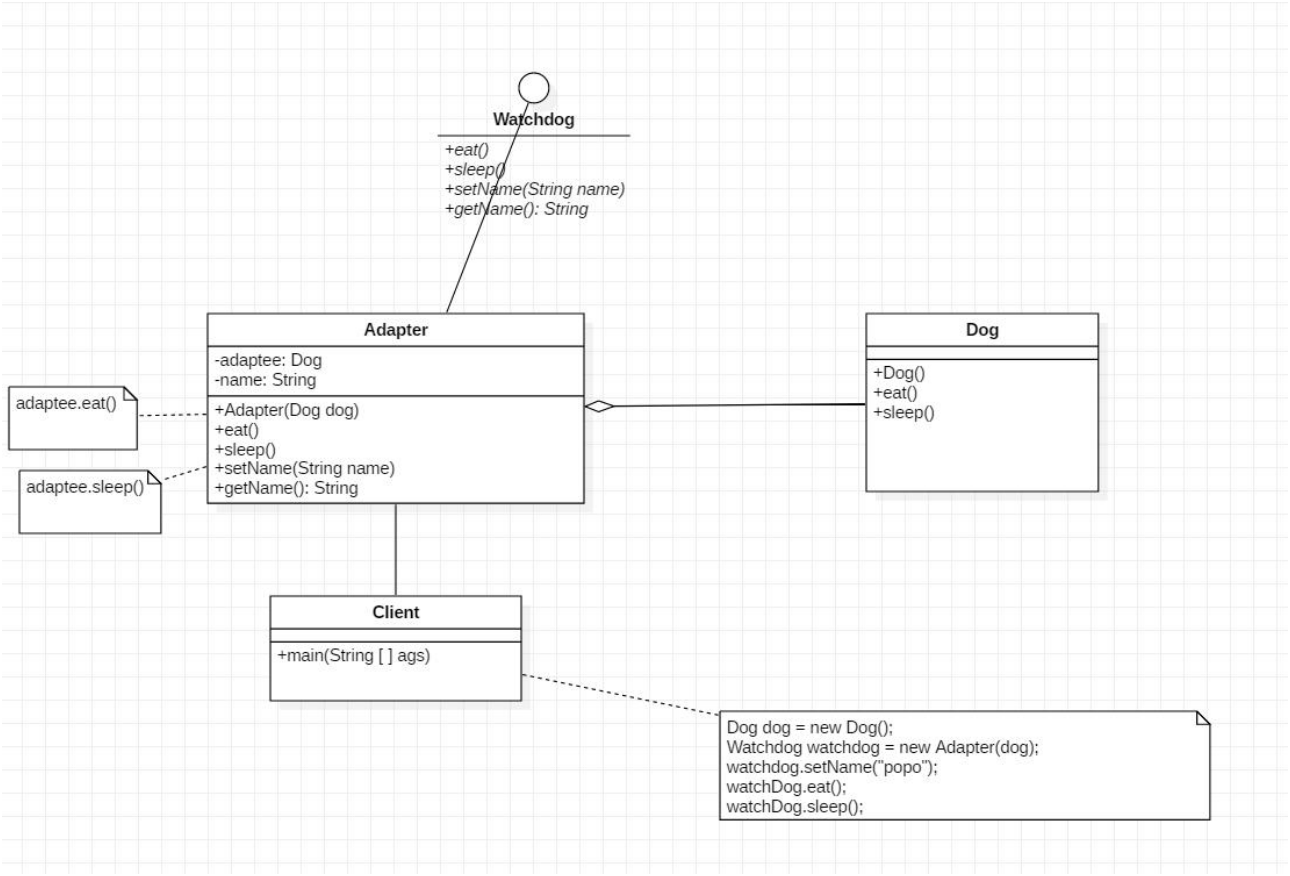
3.2.1.1 API描述

看门狗是一种狗，它有狗应该具有的生理需求，如吃（eat），睡（sleep）。但之前开发的Dog类继承自Product类，看门狗实际上不应该属于Product, 它又与Dog是有区别的，但重新开发eat和sleep功能又会产生新的成本（假设eat 和 sleep 开发成本很高），且如果Dog后续进行维护，Watchdog无法与其保持一致。

选择使用适配器模式，定义一个适配器类来实现当前看门狗的业务接口，同时又继承现有Dog类已有功能，这样减少了开发成本，还降低了Watchdog和Dog之间的耦合性。

函数名	作用
eat():void	Adapter类里的eat函数是对Watchdog里函数的实现，在函数中，调用适配者（即dog对象）的eat函数。
getName():String	获取Adapter适配的Watchdog的名字
setName(String name):void	设置Adapter适配的Watchdog的名字

3.2.1.2类图


Watchdog
+eat()
+sleep()
+setName(String name)
+getName(): String
Adapter
-adaptee: Dog
-name: String
+Adapter(Dog dog)
+eat()
+sleep()
+setName(String name)
+getName(): String
Dog
+Dog()
+eat()
+sleep()
adaptee.eat()
adaptee.sleep()
Client
+main(String [ ] ags)
Dog dog = new Dog();
Watchdog watchdog = new Adapter(dog);
watchdog.setName("popo");
watchDog.eat();
watchDog.sleep();

3.3 Bridge

设计模式简述

在现实生活中，某些类具有两个或多个维度的变化，如果用继承方式，不但对应的子类很多，而且扩展困难。选择桥接（Bridge）模式，将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度。

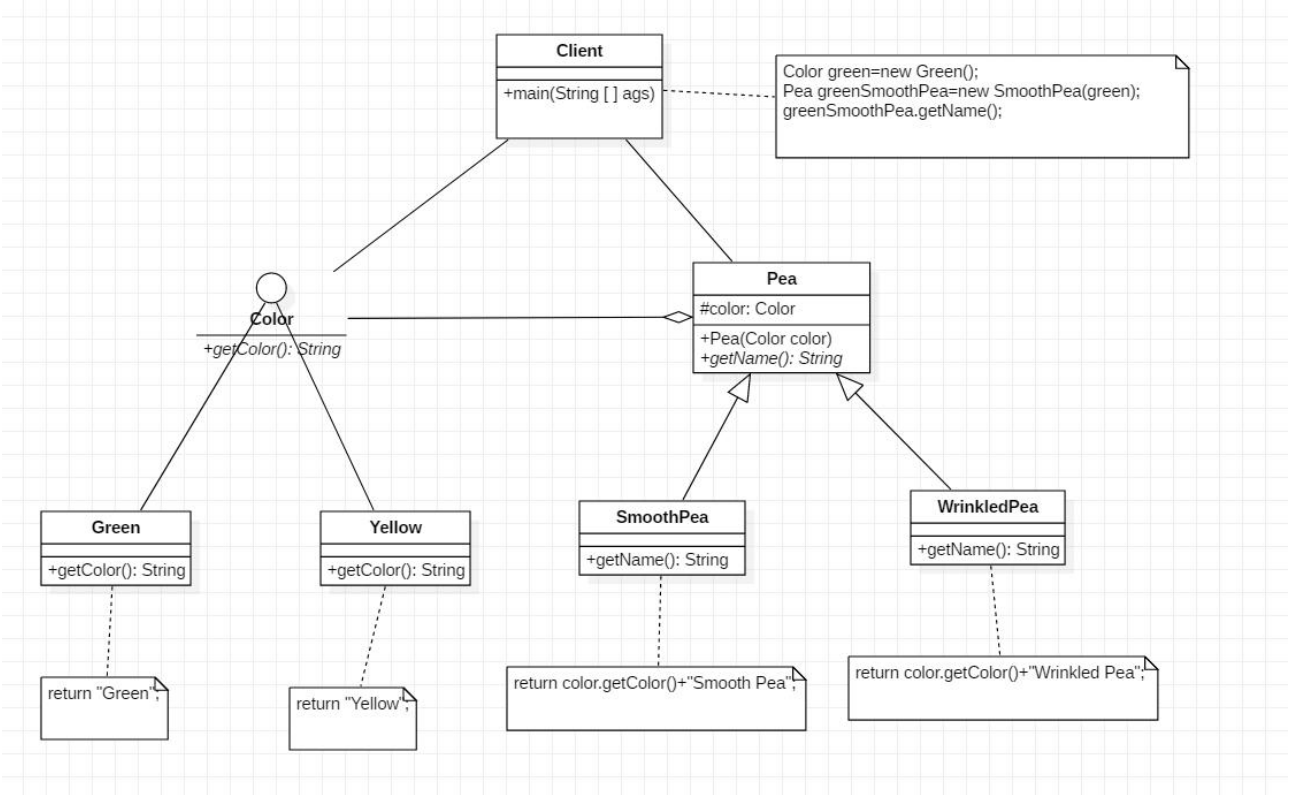
3.3.1 Bridge实现API

3.3.1.1 API描述

豌豆（Pea）有不同颜色，也有不同形状。当豌豆的颜色和形状互相组合的时候，将每个组合都开发成一个新的子类显然开发量很大，而且很难进行后续的开发。创建一个颜色（Color）类，用不同的子类与Pea的子类的组合关系来代替单纯开发Pea的子类，可以降低两种性状的偶尔性，后续也可以开发其他颜色和其他形状。

函数名	作用
getColor():String	继承自Color类，返回具体颜色的字符串
Pea(Color color)	用颜色对象作为参数来构造一个豌豆对象，实现颜色与豌豆本身的解耦
getName():String	继承自Pea类，返回Pea的颜色和形状

3.3.1.2类图



3.4 Builder

设计模式简述

建造者模式允许用户一步步构建复杂的对象，而避免处理多个类之间过度复杂的继承关系或者使用单个巨大的构造函数。与此同时，建造者模式允许用户使用同一组构造接口来创造多种不同的对象，从而方便用户的使用。

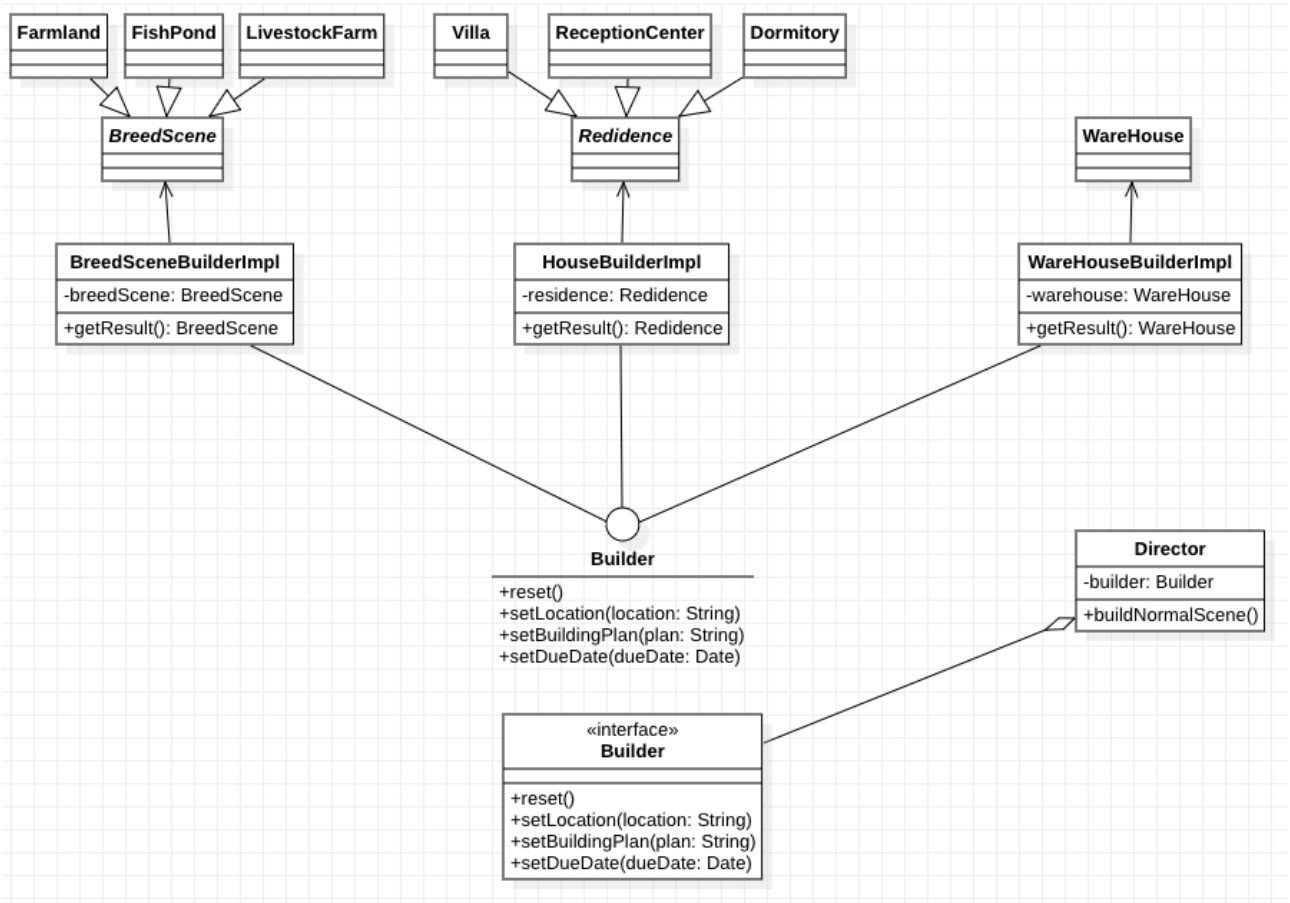
3.4.1 ShortTermWorker 实现 API

3.4.1.1 API 描述

在我们的类中有一类短期劳工，他们主要是为了模拟现实生活中农场主在需要新建各种建筑物的时候需要招聘短期工人的情景。他们能构建各种建筑，包括用于住宿的工人宿舍和农场主的别墅；用于养殖的农场、鱼塘和窝棚；用于储存物资的仓库。我们可以发现，这些建筑的构建过程都有类似的设置步骤，包括选定位置、确定建筑种类以及设置截至日期，将这些方法抽取出来，既能分解复杂的构造过程，也能允许一个 Director 来对一些常用的构建计划进行封装。

函数名	作用
void setLocation(String location)	如上说明，本函数定义于 Builder 接口中，分别在 BreedSceneBuilderImpl，HouseSceneBuilderImpl，WareHouseBuilderImpl 中进行不同的实现。
void setBuildingPlan(String plan)	同上所述。
void setDueDate(Date dueDate)	同上所述。
？ getResult()	本函数并不定义在 Builder 接口中，因为各个建造者的实现类中将存有的不同类的对象用于构建，所以他们的 getResult 函数将具有不同的返回类型。
void buildANormalScene()	本函数定义在 Director 类中。在构造函数获取了具体的 Builder 类之后，Director 能给根据一组事先定义的默认步骤来构造建筑，从而进一步简化使用。

3.4.1.2 类图

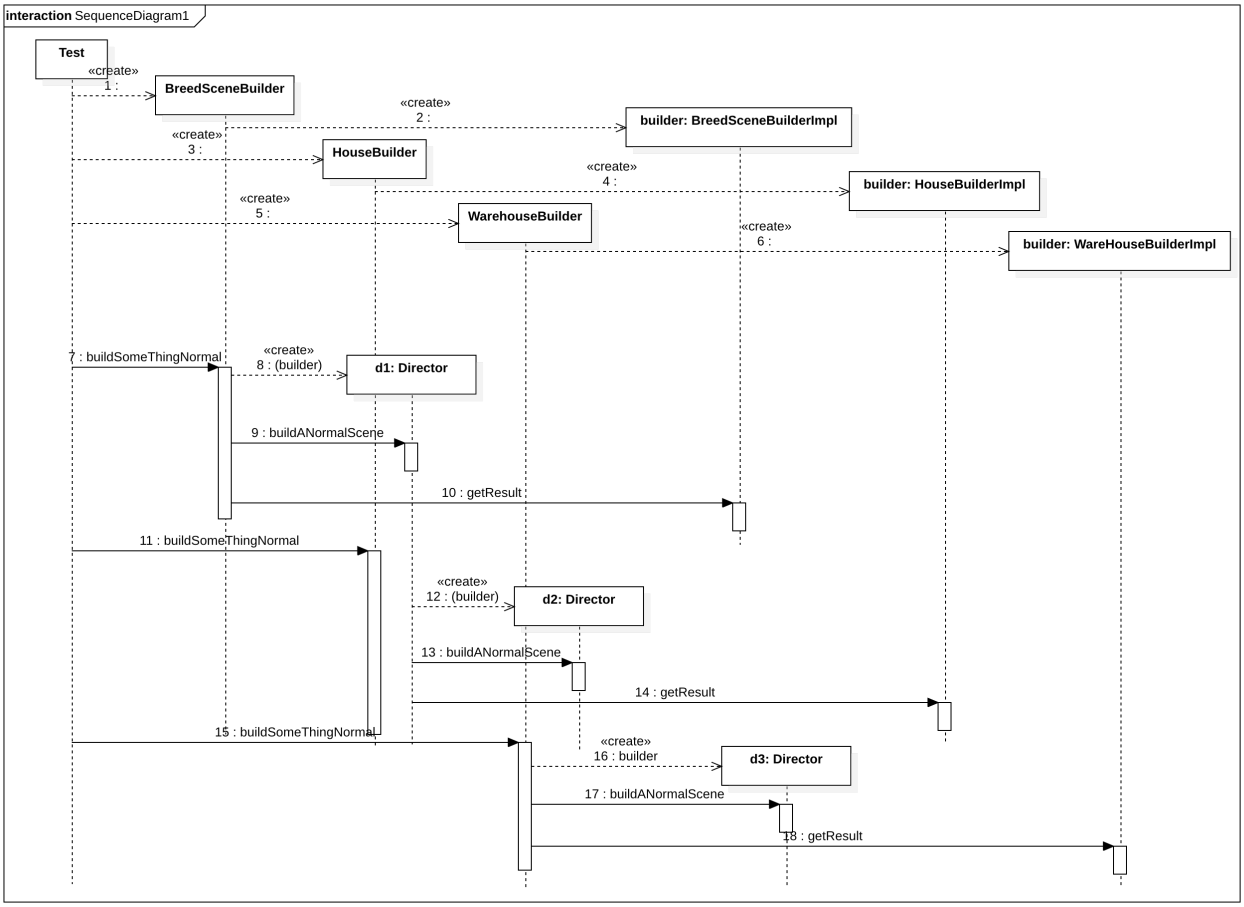


为了实现 Builder 设计模式，我们设计了如上的一些类来进行展示。首先，我们需要实现一个 Builder 的 interface，这个接口定义了一组公共的操作，其中包括了重置（将内部包含的对象清空，重新开始创建）、选定位置（为目标建筑确定位置）、设置建筑计划（正如上部分的继承系统可见，一大类的建筑可能有多个不同的小类，我们需要选择到底是要修建哪一个类型的建筑）、设置到期时间（一个工程计划应该根据到期的时间来确认如何进行规划）。在有了接口之后，我们就选择了两个不同种类的建筑物群体实现了两个实现类，分别是 BreedSceneBuilderImpl、HouseBuilderImpl 和 WarehouseBuilderImpl，分别实现了建造养殖类情景和建造住宿类情景。它们都基于相同的 Builder interface，只不过对于不同的函数做出了不同的实现。例如，在 BreeadSceneBuilderImpl 中，我们设置简直类型就只能在 Farmland、FishPond 和 LivestockFarm 中进行选择；而在 HouseBuilderImpl 中，我们所设置的类型就只能在 Villa、ReceptionCenter 和 Dormitory 中进行选择了。

除此之外，我们还实现了 Director 类，这个类按照设计模式的说明并不是必须的，但是我们仍然提供了这个类从而确保设计模式实现的完整性。这个类的意义在于封装一些较为常见的构建过程封装起来了。例如我们提供的buildNormalScene 方法就能够完成一系列自动的构建过程，从而为用户建造出一系列的建筑物而不需要用户来处理构建的细节。

这是我们具体的函数调用过程，我们可以看到，我们的 Test 类首先创建了一些建筑工人，因为它们持有的特定的 builder 实现类，从而进行来实现相应的功能。而后我们将创造的建筑工人传递到 Director 里面去，由 Director 来代理建筑工人的建造过程。在完成之后，再从各个实现中取出相应的成品类。

3.4.1.3 时序图



3.5 Chain of Responsibility Pattern

设计模式简述

使一个请求有机会被不止一个对象处理，从而避免请求的发送者和请求的接受者的耦合。把能够处理请求的对象串成一个链条，然后把请求沿着链条依次传递，直到一个对象处理它。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。

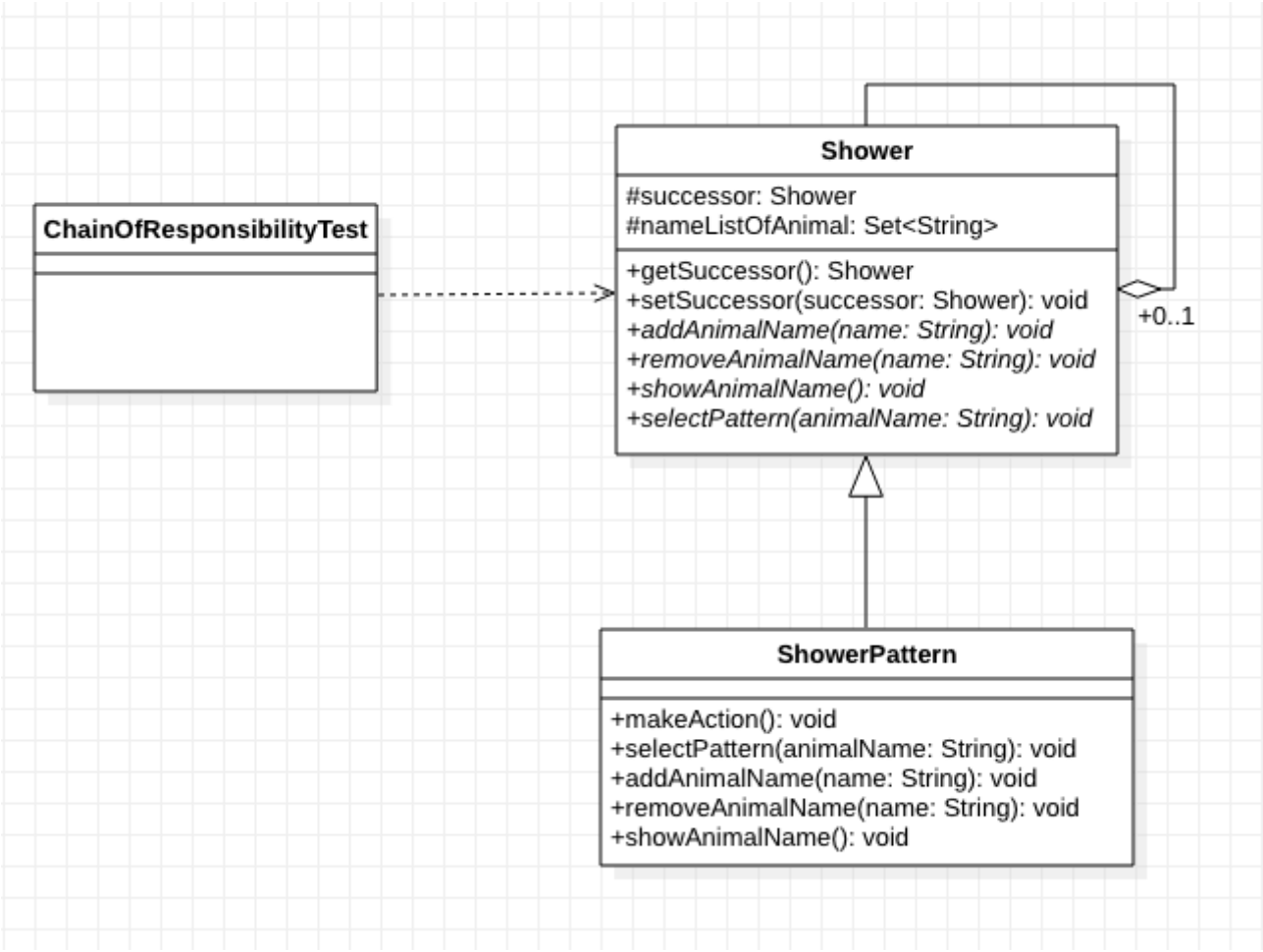
3.5.1 Shower 实现API

3.5.1.1 API描述

为动物设置一个行为Shower作为链条的基本父类，ShowerPattern作为具体的清洗模式。当任何一个动物需要进行这个行为，就依次遍历这个链条，直到有具体的行为模式给予这个清洗需求反馈为止。对于每个ShowerPattern有一个动物类别的set类型的数据，用来存储这个ShowerPattern适用的（可响应的）动物种类。当调用动物的清洗方法时，根据Shower链条的顺序逐个进行比对，直到找到具体适合的ShowerPattern返回响应。

函数名	作用
addAnimalName(in name:String): void	对模式适用的动物类进行增加操作
removeAnimalName(in name:String): void	对模式适用的动物类进行删减操作
showAnimalName(): void	查找对模式适用的动物类
selectPattern(in animalName:String): void	调用动物的清洗方法，扫描链条给予反馈

3.5.1.2 类图


ChainOfResponsibilityTest
Shower
#successor: Shower
#nameListOfAnimal: Set<String>
+getSuccessor(): Shower
+setSuccessor(successor: Shower): void
+addAnimalName(name: String): void
+removeAnimalName(name: String): void
+showAnimalName(): void
+selectPattern(animalName: String): void
+0..1
ShowerPattern
+makeAction(): void
+selectPattern(animalName: String): void
+addAnimalName(name: String): void
+removeAnimalName(name: String): void
+showAnimalName(): void

3.6 Command

设计模式简述

命令模式能将请求包装为一些对象，这些对象包含了有关于改请求的一切必要的信息。通过这样的处理，我们能够延迟对于一些方法的调用。与此同时，通过保存曾经被调用过的 Command 对象，用户能够较为方便的进行撤销 command 的操作。

3.6.1 Command 实现 API

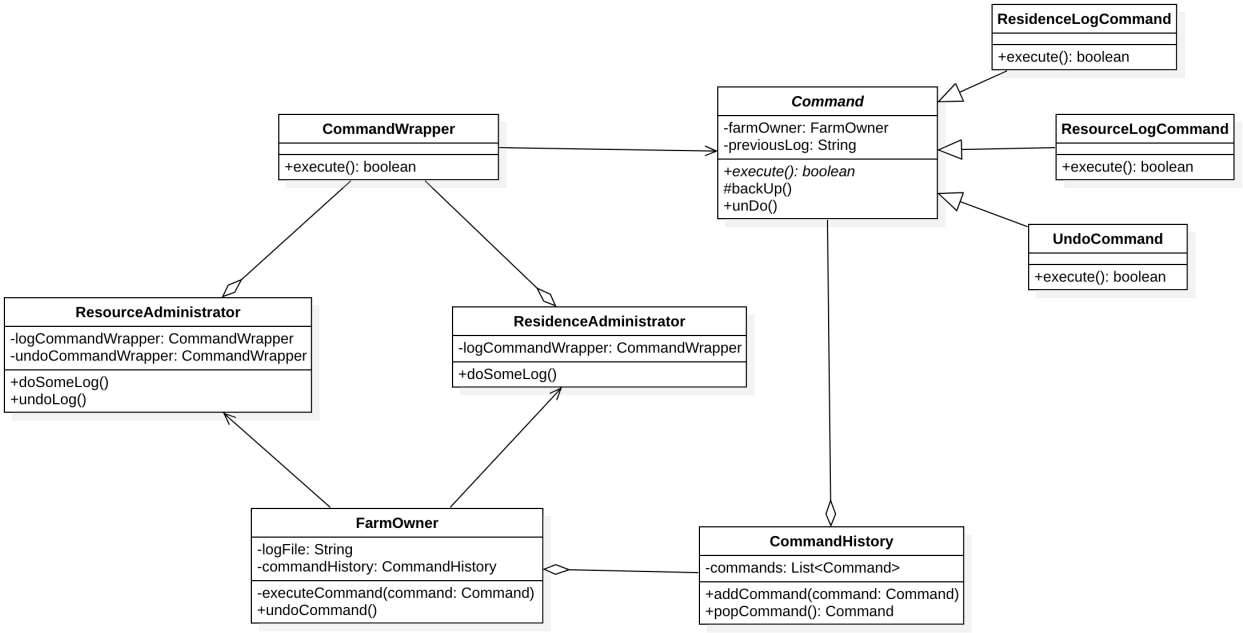
3.6.1.1 API 描述

农场主需要能够向农场中的物资管理员和宿舍管理员提供需要执行的命令，以便他们在合适的时候来执行这些任务（在本例中，我们假定了“记录日志”的任务，这两种管理员都需要在日志上写下自己需要执行的任务）。与此同时，我们同样需要支持”删除日志“的功能。综合以上两点，我们决定在农场主中采用了 Command 模式来实现这样的功能。

同时，我们需要在 FarmOwner 处保存一个栈来存储曾经执行过的 Command。除此之外，我们还使用了 Java 8 中的新特性，Lambda Expression 来实现函数作为对象的保存。

函数名	作用
boolean execute()	本函数就是经典的 Command 模式中所规定的 Command 对象必须具有的 execute 函数，其返回值 boolean 是为了判断该 Command 是否会对 log 做出改变，若做出改变，则需要保存该对象（undo 命令本身就不需要再被保存）。
void backUp()	用于记录此时 log 所处的状态，以便后续的 undo 操作。
void doSomeLog()	本函数处在 ResourceAdministrator 以及 ResidenceAdministrator 中，是为了在测试中能够通过这两个对象来调用存储在其中的 Command 而暴露的接口。
void undoLog()	同上所述。

3.6.1.2 类图

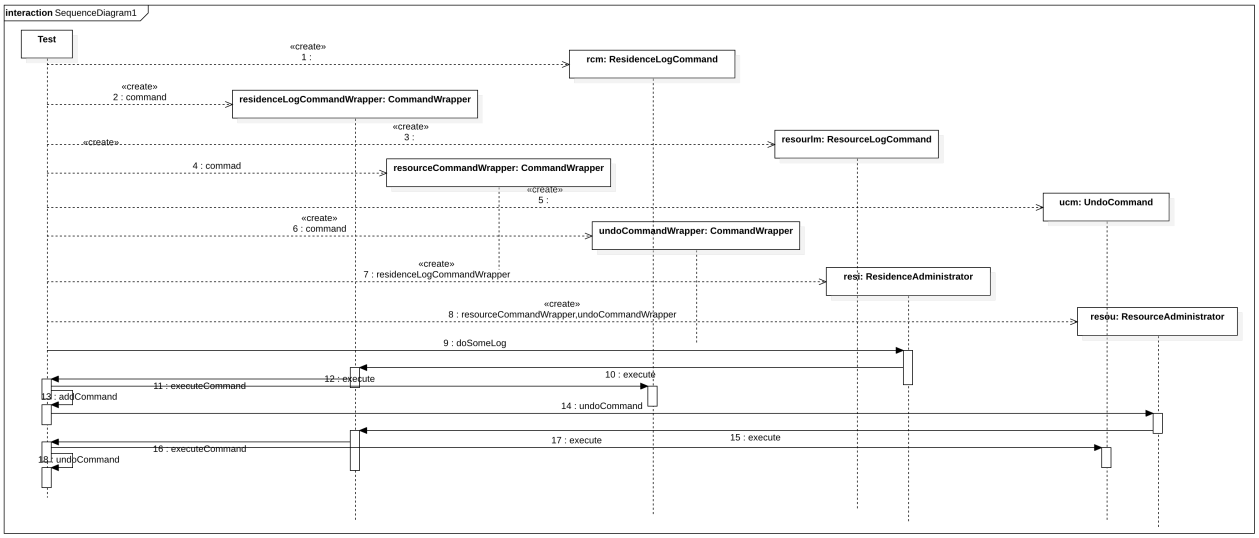


如上图所示，我们首先创造 Command 类型的虚类，这个类是 command 设计模式的关键。它拥有作为虚函数的 execute 函数，必须由它的子类进行实现。与此同时，由于 command 类需要能独自实现功能，因此它保存了 farmOwner 对象来调用实现方法。在本例中，我们一共为 Command 类实现了三个子类，分别代表了不同的命令。分别是 ResidenceLogCommand、ResourceLogCommand 和 UndoCommand，它们分别执行「住宅管理员记录日志」，「物资管理员记录日志」以及「撤销上次日志操作」这三个命令。

除此之外，我们还需要实现 CommandHistory 类，这个类保存着曾经执行过的命令。每当用户（在本例中是 ResourceAdministrator 以及 ResidenceAdministrator）调用它们所拥有的 command 对象时，就会在内部调用 FarmOwner 的 exectueCommand 命令，这就会在 history 中记录下已经执行过的 command。而对于特殊的 undocommand，则会从 commandHistory 中取出上一次执行的命令，并调用其 undo 方法，从而恢复 logFile 到上一个状态。

其中重要的实现在新建每一个 command 对象时，为了避免耦合以及将 executeCommand 函数作为公有函数的需要，我们加入了中间类 CommandWrapper，这个类是 Java 的 functional interface，它可以被一个符合特定函数签名的 lambda 表达式实例化，从而实现这个功能。

3.6.1.3 时序图



上图的序列图是对上述实现过程的详细描述。

3.7 Composite

设计模式简述

将对象组合成树状结构来体现“部分与整体“ 的层次关系。 组合模式使得客户端代码可以一致地对待 单体对象 和 由单体对象组合出的复合对象。

3.7.1 SucculentBonsai 实现API

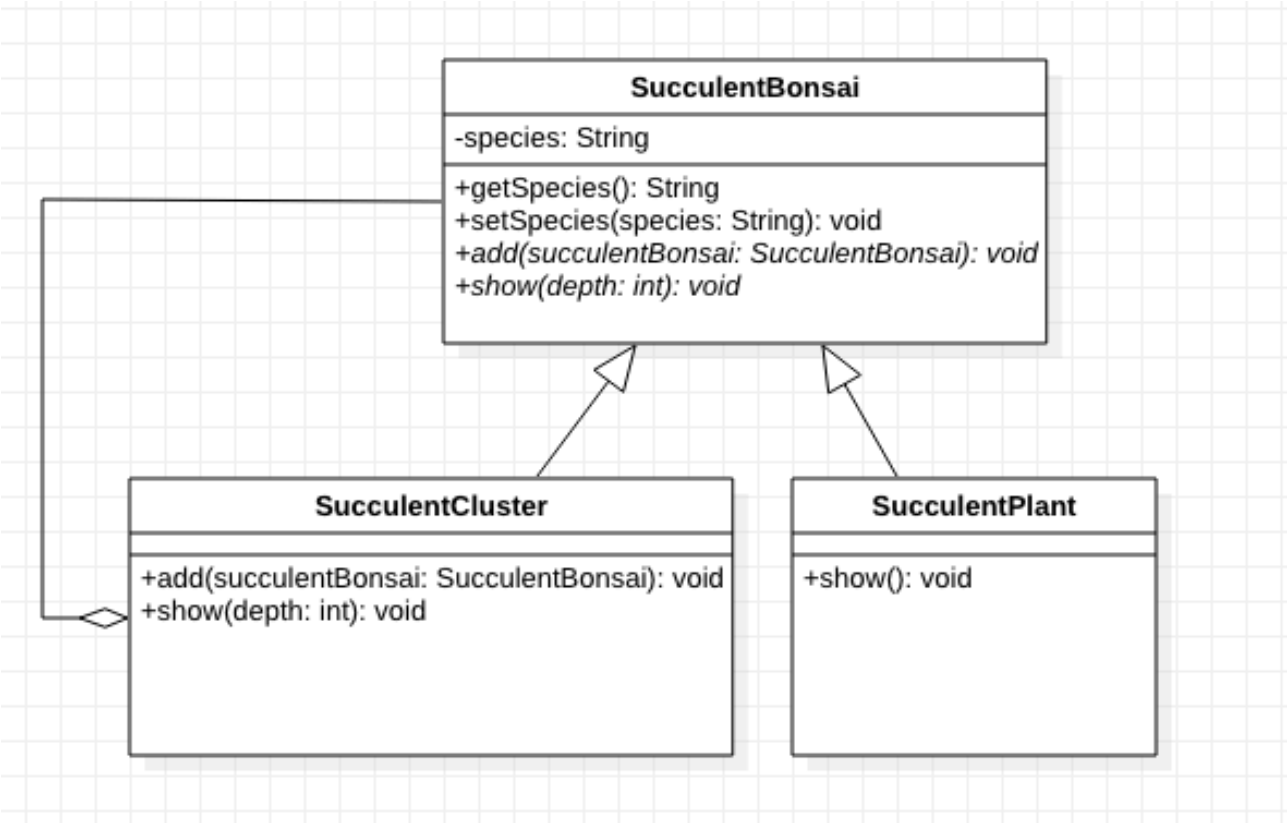
3.7.1.1 API描述

对于多肉植物而言，通常是以一个盆景为单位进行售卖，而一个盆景中既可以包括一个多肉植物的簇聚，也可以仅仅只是一棵单个的植株。将多肉盆景作为一个抽象父类SucculentBonsai，将多肉植物集和单株多肉植物分别作为相当于整体与部分的子类SucculentCluster和SucculentPlant。这样就能把多肉植物集和单株多肉植物同等的对待进行操作。

对于一个多肉盆景中，可以包含数个多肉植物簇聚和数个单株多肉植物，而任意一个多肉植物的簇聚中又能够包含数个多肉植物簇聚和单株多肉植物。作为整体的子类的多肉植物簇聚SucculentCluster具有add函数能添加内容，同时SucculentCluster和SucculentPlant都有show函数来查看其内部的元素名称。

函数名	作用
add(in succulentBonsai:SucculentBonsai): void	提供往作为整体集合的子类SucculentCluster中添加元素succulentBonsai的方法
show(in depth:int): void	显示以当前元素为根结点的全部叶子元素。depth默认值为0，作用是为了体现输出的规整性。

3.7.1.2 类图


SucculentBonsai
-species: String
+getSpecies(): String
+setSpecies(species: String): void
+add(succulentBonsai: SucculentBonsai): void
+show(depth: int): void
SucculentCluster
+add(succulentBonsai: SucculentBonsai): void
+show(depth: int): void
SucculentPlant
+show(): void

3.8 Decorator / Wrapper

设计模式简述

装饰模式又名包装模式，它以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案。运用该设计模式可以动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。

3.8.1 ResidenceDecorator 实现 API

3.8.1.1 API 描述

Residence 住宅抽象类有函数 getDescription() 和 getCost()，它们分别返回住宅的简介和造价。

在最初的设计中，我们默认每种住宅都有固定的价格：豪宅 50000 元 / 栋，接待中心 30000 元 / 栋，员工宿舍 20000 元 / 栋。但在后续开发过程中，我们想给住宅添加一些附加功能：红外报警器、监控器和中央空调，而增加这些附加功能会相应地增加住宅的造价。如何设计出一个造价动态改变的住宅类呢？这是个问题。

一种传统的做法是创建若干新的子类，例如带红外报警器的住宅、带监控器的接待中心、同时带中央空调和红外报警器的豪宅等等。但这样做的缺点非常明显。假设我们有三种住宅（豪宅、接待中心和员工宿舍）以及三种附加功能（红外报警器、监控器和中央空调），因住宅的种类和附加功能可以任意排列组合，我们需要创建大量的子类，共计 3 * P(3) = 18 种，这是十分繁琐的。

还有一种传统的做法是给住宅类新增3个布尔类的成员变量，用于表示是否安装红外报警器、是否安装监控器和是否安装中央空调。这种做法仍有缺陷：如果以后又有新的附加功能产生，例如太阳能电池板，那我们修改住宅类的设计，给它添加新的布尔类成员变量，用于表示是否安装太阳能电池板。如果我们以后不再支持某个附加功能，我们还得删掉住宅类的相关布尔变量。这违反了开放封闭原则。

为了解决上述缺陷，我们采用装饰模式。首先设计一个附加功能抽象类（ResidenceDecorator)，其继承自住宅类（Residence）。再设计三个附加功能实体类（Monitor, InfraredAlert 和 CentralAirCondition），它们继承自ResidenceDecorator。这样附加功能类和三种住宅实体类都同时继承自Residence类（是不是很神奇？）。每一个附加功能类内都有一个指向 Residence 类的指针，并且它们也同样具有 getDescription() 和 getCost() 函数。每当我们想要给Residence对象新增某个附加功能时，就创建这个附加功能类并让其成员指针指向Residence对象。这种添加附加功能的操作被称作“装饰”，而附加功能被称作装饰器decorator，原本的三种住宅被称作decorated。

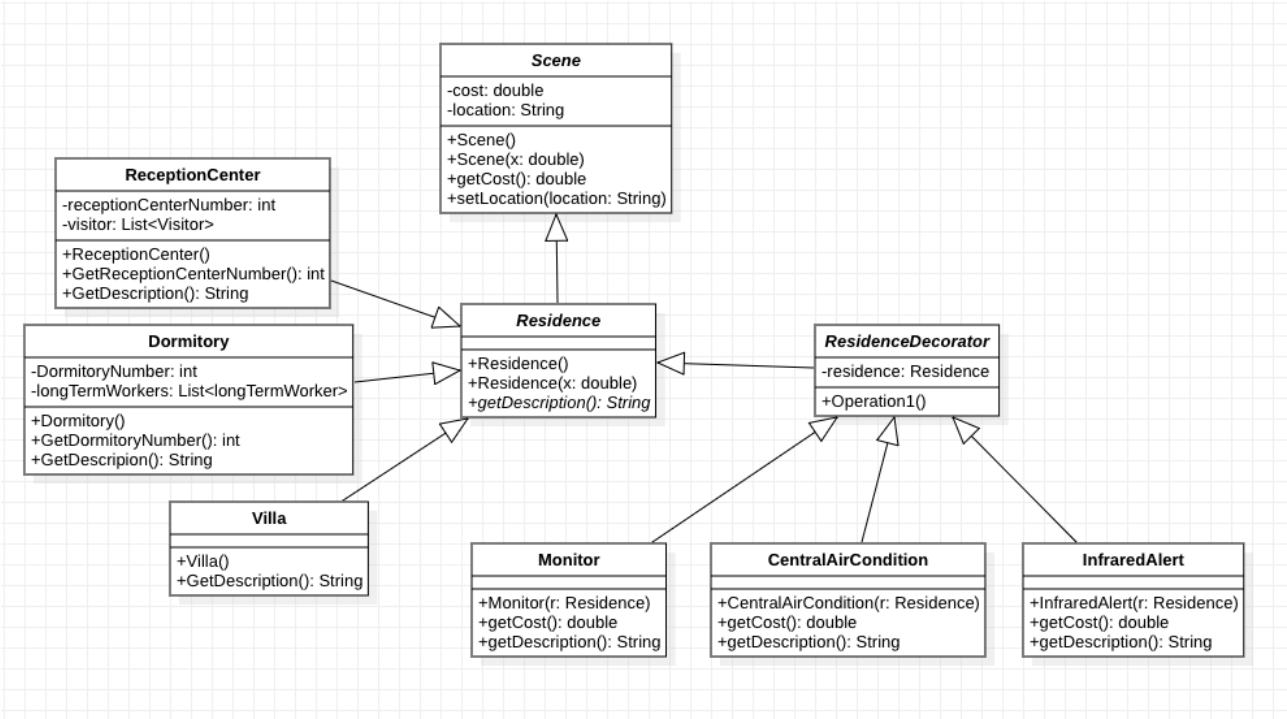
一个decorated在经过各种decorator的装饰后，它们会呈链表状。链表的尾节点为decorated，而其余节点都为decorator。前一个节点的 getDescription() 函数会打印出自己的描述并调用后一个节点的 getDescription() 函数，前一个节点的 getCost() 函数会调用后一个节点的

getCost() 函数并将其返回值与自己的价钱相加。故每当我们想知道被装饰的住宅的简介和总造价时，可以迭代地调用这个链表中每个节点地 getDescription() 和 getCost()。

运用装饰模式，我们的程序变得易于扩展，且满足开放封闭原则。每当我们需要给系统新增一种附加功能时，只需要新增一个附加功能实体类，而无需改变住宅实体类本身。

函数名	作用
Dormitory(), ReceptionCenter(), Villa()	三个住宅实体类的构造函数。
Monitor(Residence r), InfraredAlert(Residence r), CentralAirCondition(Residence r)	三个附加功能实体类的构造函数，在调用过程中会将被其装饰的 Residence 对象传给他的成员指针 residence。
double getCost()	返回住宅的总造价（它可能被装饰也可能未被装饰）
String getDescription()	返回住宅的描述

3.8.1.2 类图



3.9 Facade

设计模式简介

通过对外观的包装，使客户端只能看到外观对象，而不会看到具体的细节对象，这样降低了应用程序的复杂度，并且提高了程序的可维护性。换言之，在外观模式中，子系统的外部调用者必须通过一个统一的 Facade 对象才能和子系统内部通信。

3.9.1 ResidenceTask 实现 API

3.9.1.1 API描述

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。

为了实现以上功能，一种比较传统的做法是在住宅管理员的工作函数中依次执行以上操作，但是这样做会使得住宅管理员的工作函数非常庞杂，还会使住宅管理员与各个住宅实体类的耦合度增加，不利于扩展。

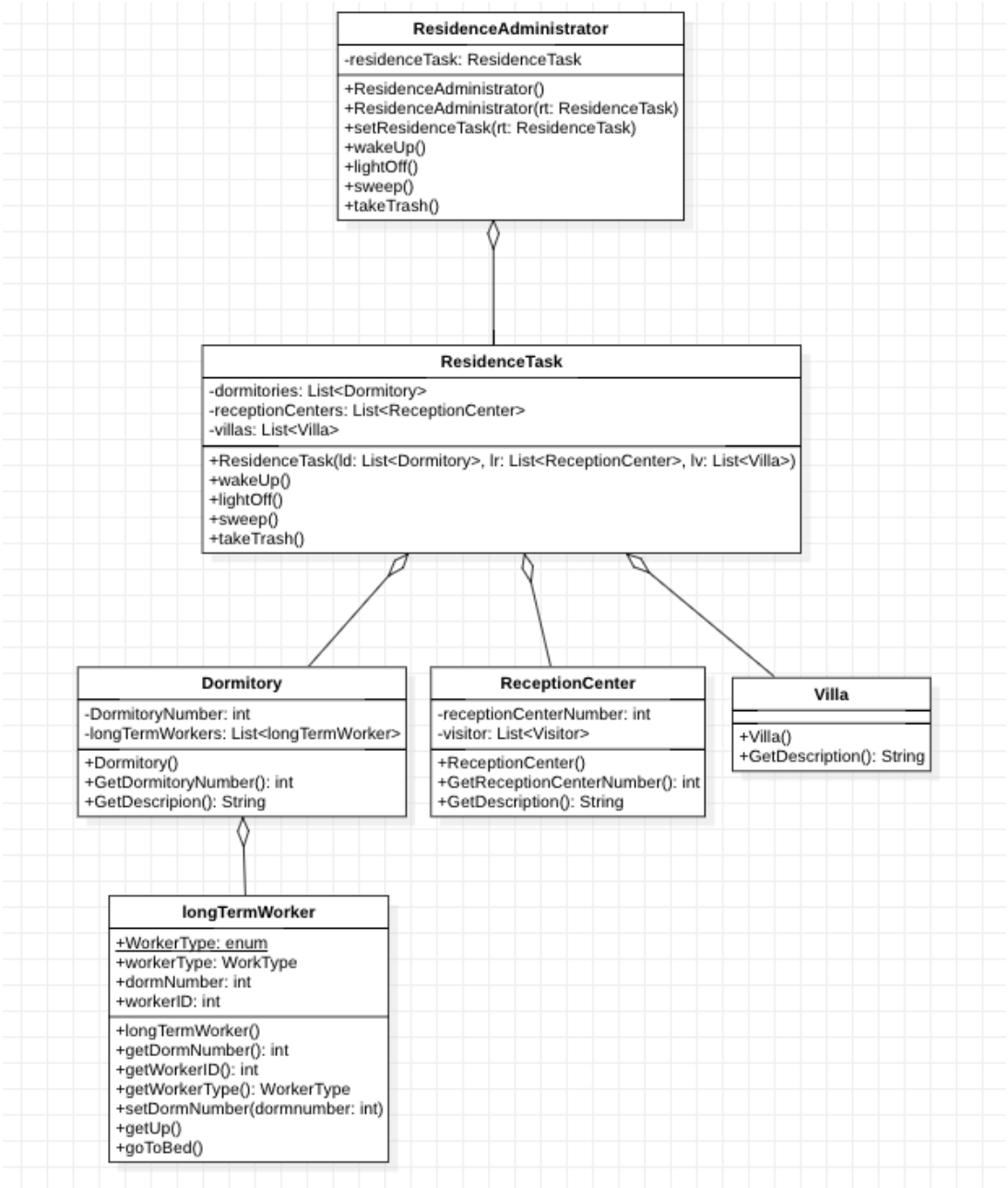
为了封装住宅管理员的工作细节，简化住宅管理员工作函数的设计，同时降低住宅管理员和各个住宅实体类的耦合度，我们新增 ResidenceTask 类，每一个 Residence 对象都会对应着某一个住宅管理员的工作。这个类将存储住宅管理员负责的住宅，即 dormitories、receptionCenters 和 villas 三个变量。它将住宅管理员的工作封装成三个函数，即 wakeUp(), lightOff(), sweep() 和 takeTrash()，让这些函数去执行庞杂的工作，例如 wakeUp() 函数会遍历住宅管理员负责的每一个员工宿舍中的每一个员工并将其唤醒，sweep() 函数会遍历负责的每一个员工宿舍和每一个接待中心并整理其内务。

在们住宅管理员工作时，他们只需调用自己的 ResidenceTask 对象中的那三个函数就行了，且它们并不知道这三个函数的实现细节。这个ResidenceTask 被称作住宅实体类的外观（Facade）。当需要修改住宅管理员的工作时，只需重点修改 ResidenceTask 类就可以了，而不需过多修改住宿管理员类本身。

函数名	作用
ResidenceTask(List<Dormitory> ld, List<ReceptionCenter> lr, List<Villa> lv)	ResidenceTask 构造函数，用于给住宅管理员分配任务。
ResidenceAdministrator. wakeUp()	叫醒员工宿舍中的员工
ResidenceAdministrator. lightOff()	催促员工回员工宿舍睡觉
ResidenceAdministrator. sweep()	整理员工宿舍和接待中心的内务

ResidenceAdministrator. takeTrash()	运走员工宿舍、接待中心和豪宅的垃圾

3.9.1.2 类图



3.10 Factory Method, Virtual Constructor

设计模式简述

工厂方法模式，又称工厂模式、多态工厂模式和虚拟构造器模式，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。

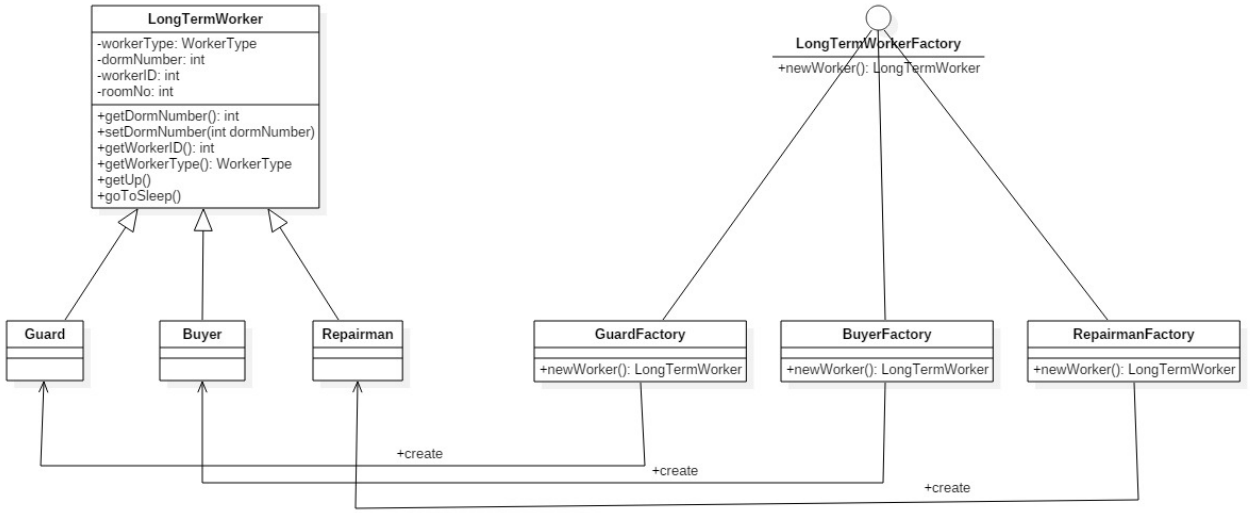
3.10.1 LongTermWorkerFactory实现API

3.10.1.1 API 描述

农场中有很多种类的长工，他们需要不同的工厂去创造示例。这里我们用LongTermWorkerFactory 工厂父类定义创建 LongTermWorker 的抽象方法，然后定义长工的基类LongTermWorker。这样，每当需要用工厂方法创建一个新的种类的长工，只需定义LongTermWorker的子类，然后定义工厂父类的子类去实例化新添加的长工对象。具体的长工工厂类和其能创建的长工类是一一对应的。如此一来便克服了简单工厂模式违背OCP原则的缺点，而保留了封装对象创建过程的优点,降低客户端和工厂的耦合性。

函数名	作用
LongTermWorker newWorker()	工厂父类的抽象方法，具体实现由创建具体LongTermWorker对象的具体工厂子类决定。

3.10.1.2 类图



3.11 Flyweight Pattern

设计模式简述

运用共享技术有效地支持大量细粒度的对象。

3.11.1 getCanvas实现API

3.11.1.1 API 描述

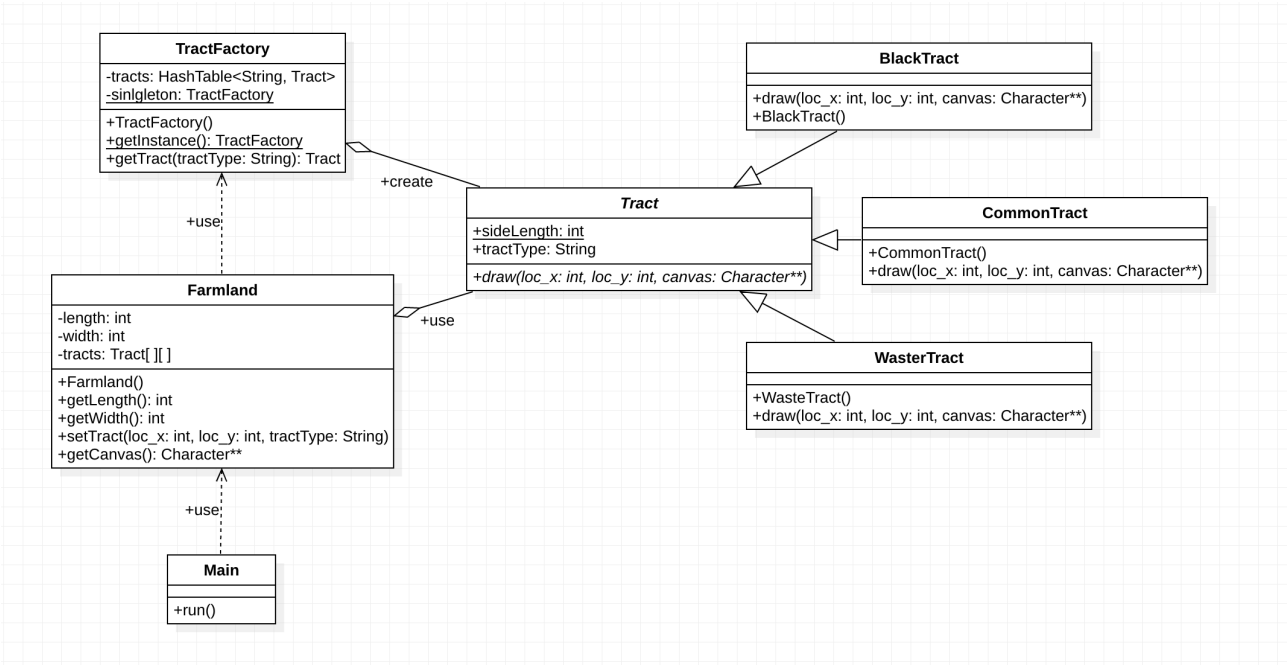
和QQ空间中的开心农场类似，每一片农田（Farmland）都由若干小块（Tract）组成，这些 Tract 具有相同的土质。我们的系统支持三种类型的 Tract，分别是未开垦的荒地（WasteTract），普通土地（CommonTract）和黑土地（BlackTract）。默认情况下，一个 Farmland 包含 12 个 Tract，这 12 个 Tract 排成一个 3*4 的矩阵。Farmland 类中会维护一个二维数组 tracts\[3][4]，它存储这 12 个 Tract 对象的引用。

Farmland 类有一个成员函数 getCanvas()，它返回 Farmland 的图像，这个图像其实就是一个存储 Character 的矩阵（由二维数组表示）。由于我们默认一个 Tract 的像素大小为 9*9（即一个 Tract 对应一个 9*9 的 Character 矩阵），故 Farmland 的图像矩阵大小为 (3*9)*(4*9)。getCanvas() 在执行过程中，会首先申明一个空的 (3*9)*(4*9) 的Character 矩阵，接着让每一个 Canvas 根据自己在 Farmland 中的位置依次渲染这个矩阵，渲染结束后返回这个矩阵。

如果这 12 个 Tract 每一个都代表一个新的对象，就会造成很大的空间浪费。因为相同类型的 Tract 图像也一样，它们渲染的方式也一样，所以我们没有必要对每一个 Tract 都新建一个对象。在 tracts\[3][4] 中，相同类型的 Tract 实际上都是同一个对象的引用。由于我们只有三种类型的 Tract，故 Tract 对象最多只有三个，它们被存储在 TractFactory 中。每当需要新的 Tract 时，TractFactory 便返回所需对象的引用。

函数名	作用
Farmland. setTract(int loc_x, int loc_y, String tractType)	将 Farmland 中坐标为（loc_x, loc_y）的土地块设置成类型为 tractType 的 Tract
Farmland. getCanvas()	将 Farmland 的图像输出到控制台
Farmland()	Farmland 的构造函数，默认创建 12 个 WasteTract。

3.11.1.2类图


TractFactory
-tracts: HashTable<String, Tract>
-sinlgleton: TractFactory
+TractFactory()
+getInstance(): TractFactory
+getTract(tractType: String): Tract
BlackTract
+draw(loc_x: int, loc_y: int, canvas: Character**)
+BlackTract()
+create
Tract
+sideLength: int
+tractType: String
+draw(loc_x: int, loc_y: int, canvas: Character**)
CommonTract
+CommonTract()
+draw(loc_x: int, loc_y: int, canvas: Character**)
+use
Farmland
-length: int
-width: int
-tracts: Tract[ ][ ]
+Farmland()
+getLength(): int
+getWidth(): int
+setTract(loc_x: int, loc_y: int, tractType: String)
+getCanvas(): Character**
+use
WasterTract
+WasteTract()
+draw(loc_x: int, loc_y: int, canvas: Character**)
+use
Main
+run()

3.12 Interpreter Pattern

设计模式简述

在软件开发中，会遇到有些问题多次重复出现，而且有一定的相似性和规律性。如果将它们归纳成一种简单的语言，那么这些问题实例将是该语言的一些句子，这样就可以用“编译原理”中的解释器模式来实现了。但对于满足以上特点，且对运行效率要求不是很高的应用实例，如果用解释器模式来实现，其效果是非常好的。

3.12.1 Person, Supply实现API

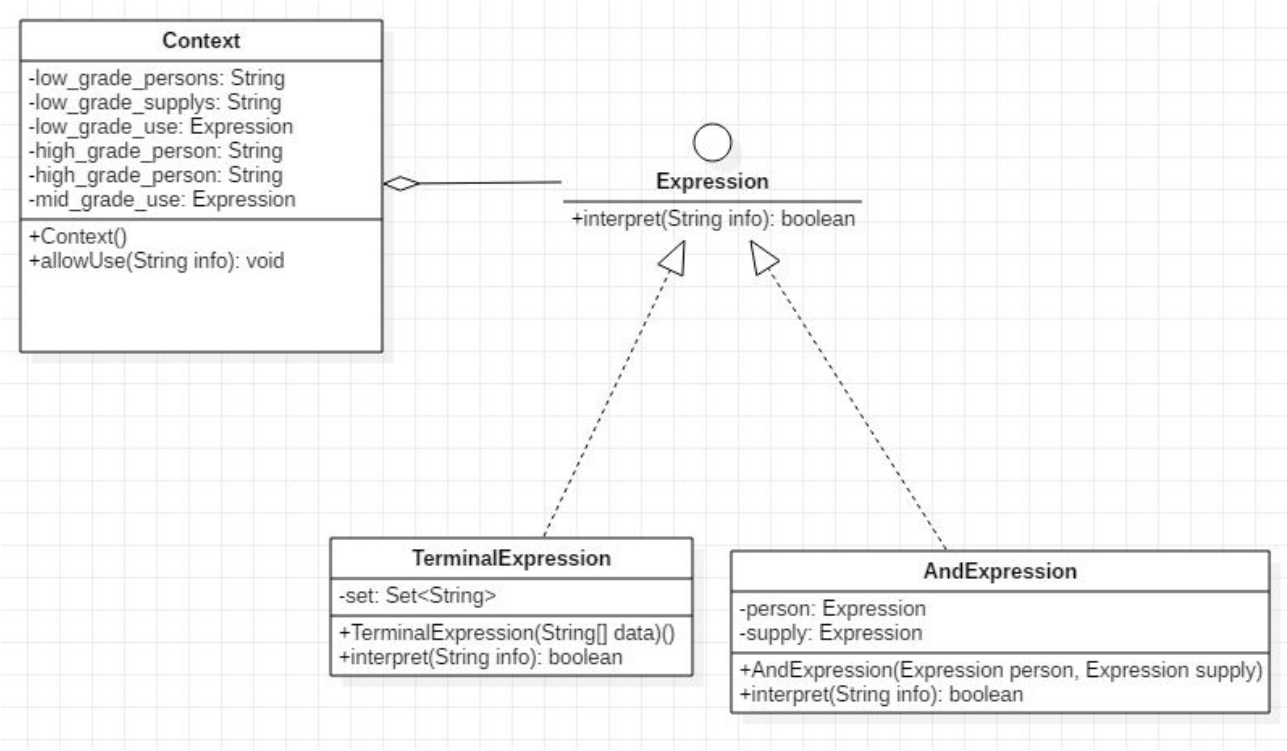
3.12.1.1 API 描述

一些工具的使用是需要权限的，并非所有人都可以使用所有工具，例如，“警卫使用警棍”就是合理的，而“建筑工使用警棍”的行为是不合理的。在农场中，存在着大量的“某人”使用“某物”的过程，对于这些行为，需要判断其合法性。

显然的，使用大量的if语句来进行这种判断是低效且扩展性差的，因此，选择使用解释器模式来完成这种判断，使得逻辑易于理解，且增强了可扩展性，降低了耦合度。

函数名	作用
TerminalExpression.interpret(String info): boolean	TerminalExpression类所实现的interpret函数，用于判断作为参数的String是否包含在该类的set中
AndExpression.interpret(String info): boolean	AndExpression类所实现的interpret函数，用于将传入的参数String根据语法树拆解并根据此来判断整个句子是否合法
allowUse(String info): void	用来判断一个句子是否合法，即：低等级使用权限和高等级使用权限至少应满足其中之一

3.12.1.2类图


Context
-low_grade_persons: String
-low_grade_supplys: String
-low_grade_use: Expression
-high_grade_person: String
-high_grade_person: String
-mid_grade_use: Expression
+Context()
+allowUse(String info): void
Expression
+interpret(String info): boolean
TerminalExpression
-set: Set<String>
+TerminalExpression(String[] data)()
+interpret(String info): boolean
AndExpression
-person: Expression
-supply: Expression
+AndExpression(Expression person, Expression supply)
+interpret(String info): boolean

3.13 Iterator Pattern

设计模式简述

在现实生活以及程序设计中，经常要访问一个聚合对象中的各个元素，如“数据结构”中的链表遍历，通常的做法是将链表的创建和遍历都放在同一个类中，但这种方式不利于程序的扩展，如果要更换遍历方法就必须修改程序源代码，这违背了 “开闭原则”。

“迭代器模式”能较好地克服以上缺点，它在客户访问类与聚合类之间插入一个迭代器，这分离了聚合对象与其遍历行为，对客户也隐藏了其内部细节，且满足“单一职责原则”和“开闭原则”。

3.13.1 Supply实现API

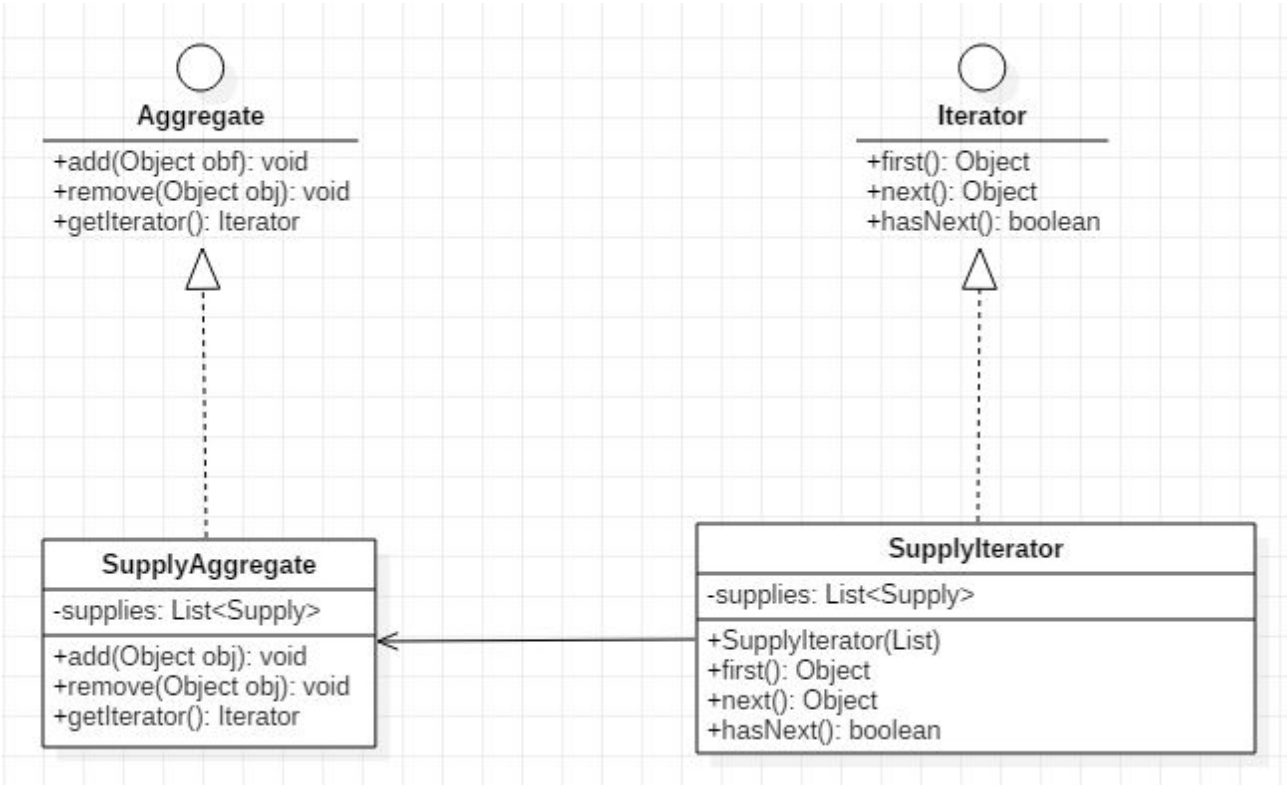
3.13.1.1 API 描述

我们经常需要去查看仓库中剩余的物品的数量以及使用情况，而且有时候我们会希望仅仅查看消耗品，有时候会希望仅仅查看工具，或者会希望去查看那些即将到达使用期限的物品或是即将被用完的消耗品……我们有如此之多的对于仓库中物品的查看方式，所以仅仅将遍历方式固定为某一种是不合理的。

显然的，将这么多遍历方式都实现一遍并写入仓库管理类中是不合理的，这严重违反了单一职责原则，引起代码的冗余和浪费。因此，我们使用迭代器模式，使得程序逻辑易于理解，可扩展性增强。

函数名	作用
add()	向物品的聚合类中添加某种物品
remove()	从物品的聚合类中移除某种物品
getIterator()	获取一个根据物品的聚合类创建出来的物品的迭代器
first()	获取第一个合理的元素
next()	获取下一个合理的元素
hasNext()	判断是否还有下一个合理的元素

3.13.1.2类图


Aggregate
+add(Object obf): void
+remove(Object obj): void
+getIterator(): Iterator
Iterator
+first(): Object
+next(): Object
+hasNext(): boolean
SupplyAggregate
-supplies: List<Supply>
+add(Object obj): void
+remove(Object obj): void
+getIterator(): Iterator
SupplyIterator
-supplies: List<Supply>
+SupplyIterator(List)
+first(): Object
+next(): Object
+hasNext(): boolean

3.14 Mediator Pattern

设计模式简述

中介者（Mediator）模式：定义一个中介对象来封装一系列对象之间的交互，使原有对象之间的耦合松散，且可以独立地改变它们之间的交互。中介者模式又叫调停模式，它是迪米特法则的典型应用。

- 降低了对象之间的耦合性，使得对象易于独立地被复用。
- 将对象间的一对多关联转变为一对一的关联，提高系统的灵活性，使得系统易于维护和扩展。

其主要缺点是：当同事类太多时，中介者的职责将很大，它会变得复杂而庞大，以至于系统难以维护。

3.14.1 植物自然传粉实现API

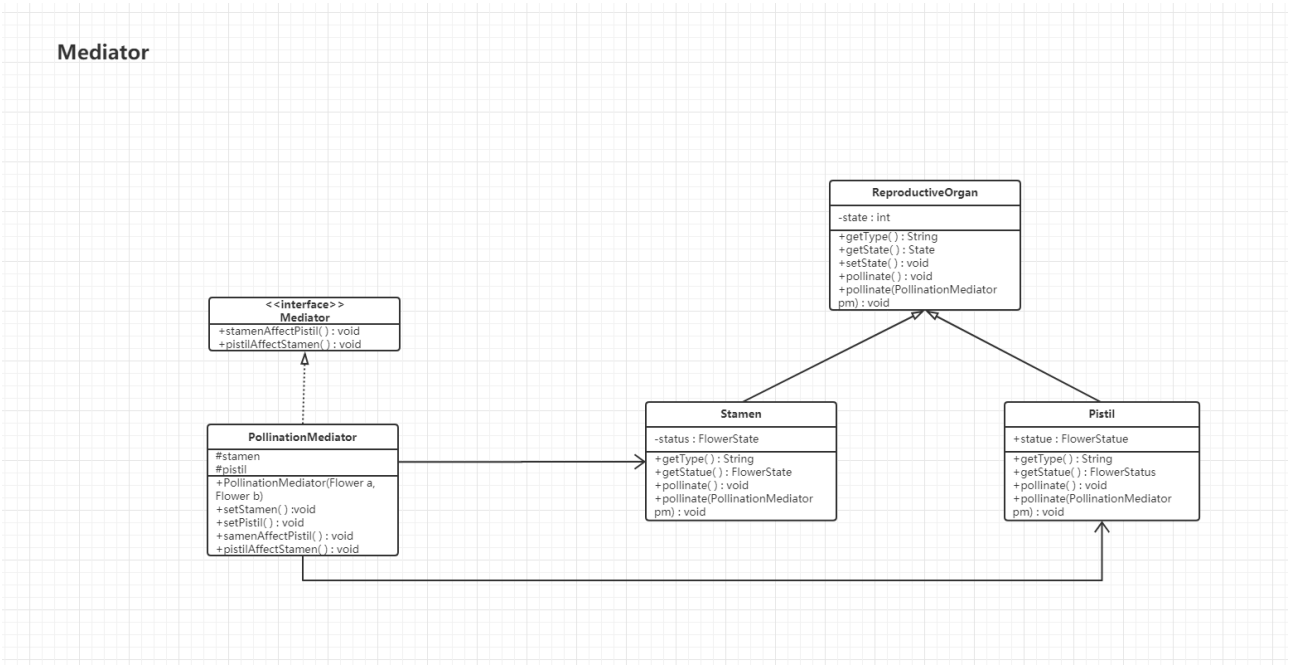
3.14.1.1 API 描述

植物分为异花授粉与自花受粉两种，自花授粉无需人工帮助，异花授粉需人工帮助，在目前拥有的植物中：苹果、樱桃为异花授粉植物，土豆、马铃薯为自花授粉植物。植物自花传粉时，多个pistil与stamen进行交互，如果两者直接进行交互会产生较为复杂的耦合关系，所以加入中介者PollinationMediator类进行帮助传粉。

设计如下：我们将中介者模式运用到了植物自然传粉这一行为体系中，在自然传粉的过程中，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。

类名	作用
Pistil	雌蕊类
Stamen	雄蕊类
PollinationMediator	自然传粉中介者，逻辑上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

3.14.1.2类图


Mediator
ReproductiveOrgan
-state : int
+getType( ) : String
+getState( ) : State
+setState( ) : void
+pollinate( ) : void
+pollinate(PollinationMediator pm) : void
<<interface>>
Mediator
+stamenAffectPistil( ) : void
+pistilAffectStamen( ) : void
PollinationMediator
#stamen
#pistil
+PollinationMediator(Flower a, Flower b)
+setStamen( ) :void
+setPistil( ) : void
+samenAffectPistil( ) : void
+pistilAffectStamen( ) : void
Stamen
-status : FlowerState
+getType( ) : String
+getStatue( ) : FlowerState
+pollinate( ) : void
+pollinate(PollinationMediator pm) : void
Pistil
+statue : FlowerStatue
+getType( ) : String
+getStatue( ) : FlowerStatus
+pollinate( ) : void
+pollinate(PollinationMediator pm) : void

3.15 Memento Pattern

设计模式简述

备忘录（Memento）模式：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 。
备忘录模式是一种对象行为型模式，其主要优点如下。

- 提供了一种可以恢复状态的机制。当用户需要时能够比较方便地将数据恢复到某个历史的状态。
- 实现了内部状态的封装。除了创建它的发起人之外，其他对象都不能够访问这些状态信息。
- 简化了发起人类。发起人不需要管理和保存其内部状态的各个备份，所有状态信息都保存在备忘录中，并由管理者进行管理，这符合单一职责原则。

其主要缺点是：资源消耗大。如果要保存的内部状态信息过多或者特别频繁，将会占用比较大的内存资源。

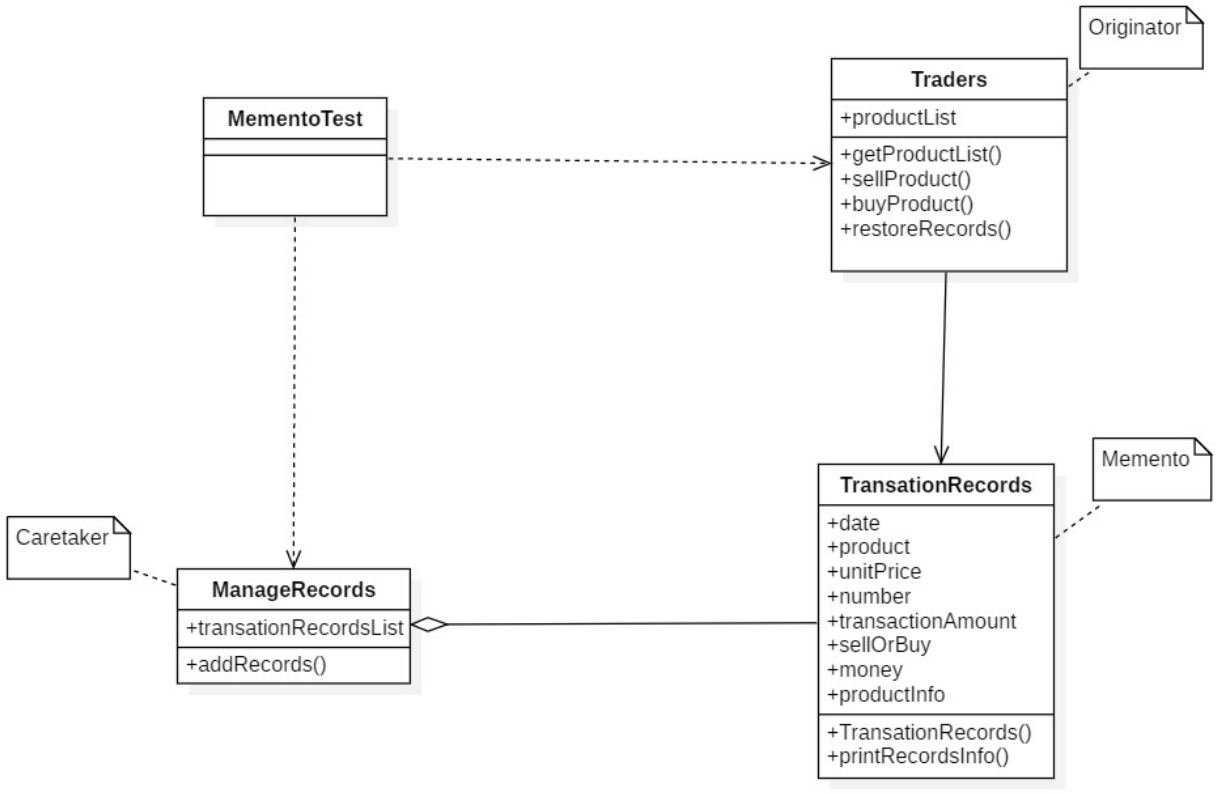
3.15.1 买卖产品实现API

3.15.1.1 API 描述

农场的贸易信息交由Trader管理，Trader可得知当前农场中的所有产品信息，并且可以购入产品以及卖出产品。每进行一次交易，都会产生一条交易记录，交易记录可以恢复。用TransationRecords类作为备忘录记录交易信息，所有备忘录由ManageRecords进行管理。

函数名	作用
buyProduct()	Trader购买产品，生成的订单为一条备忘录
sellProduct()	Trader卖出产品，生成的订单为一条备忘录
restoreRecords()	恢复到指定历史状态
addRecords	增加备忘录到链表中，以便管理

3.15.1.2类图


Originator
Traders
+productList
+getProductList()
+sellProduct()
+buyProduct()
+restoreRecords()
MementoTest
Memento
TransationRecords
+date
+product
+unitPrice
+number
+transactionAmount
+sellOrBuy
+money
+productInfo
+TransationRecords()
+printRecordsInfo()
Caretaker
ManageRecords
+transationRecordsList
+addRecords()

3.16 Observer Pattern

设计模式简述

在现实世界中，许多对象并不是独立存在的，其中一个对象的行为发生改变可能会导致一个或者多个其他对象的行为也发生改变。例如，某种商品的物价上涨时会导致部分商家高兴，而消费者伤心；还有，当我们开车到交叉路口时，遇到红灯会停，遇到绿灯会行。这样的例子还有很多，例如，股票价格与股民、微信公众号与微信用户、气象局的天气预报与听众、小偷与警察等。

在软件世界也是这样，例如，Excel 中的数据与折线图、饼状图、柱状图之间的关系；MVC 模式中的模型与视图的关系；事件模型中的事件源与事件处理者。所有这些，如果用观察者模式来实现就非常方便。

3.16.1 Supply实现API

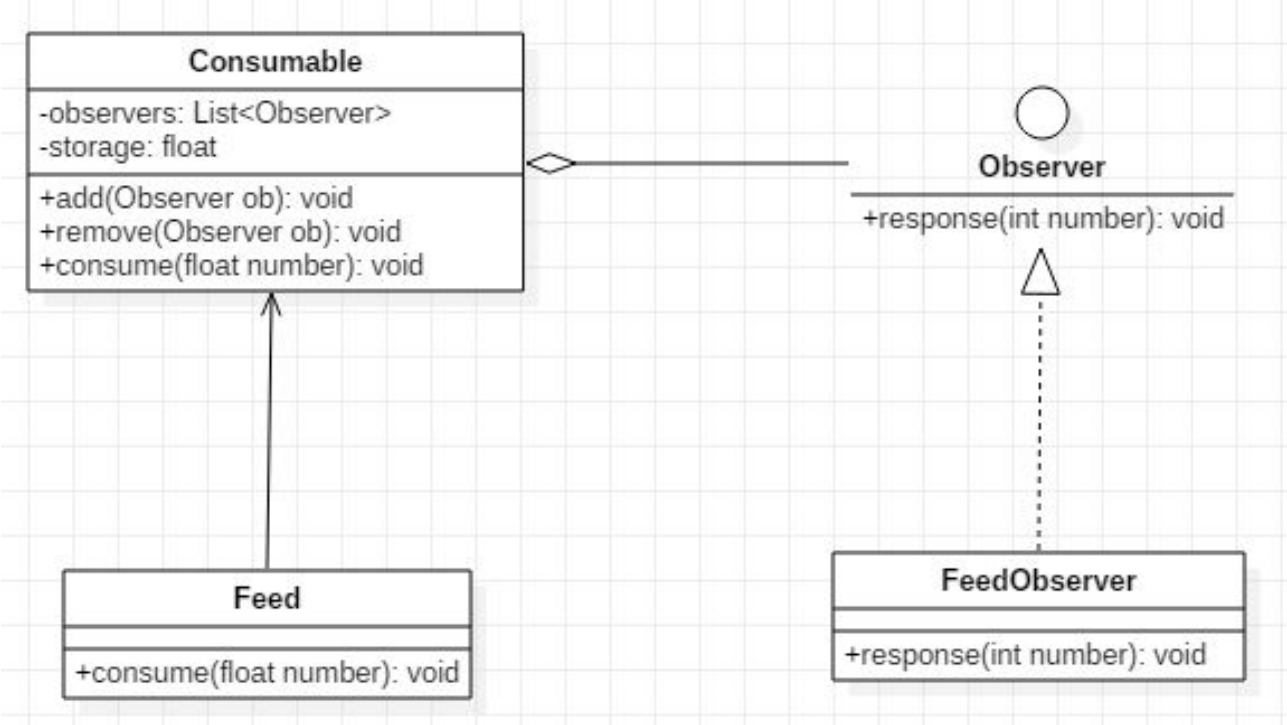
3.16.1.1 API 描述

仓库中的各类物品对于农场的正常工作是非常重要的，我们需要保证各类物资都有足够的储备来应对每天的消耗，但是，我们没有那么多空闲去不停地检查仓库中剩余的物资数量是否足够，因此，我们需要一种方法来在物资被消耗到较低水平时提醒我们。

使用Observer设计模式能够完美解决我们的问题，我们会为这些重要的物资分别设置观察者以在这些物资出现短缺时向我们发出警告。

函数名	作用
response()	作为Observer的响应函数，会在满足条件时发出警告
add()	为Storage添加观察者
remove()	从Storage移除观察者
consume()	消耗number数量的库存，并且会向观察者发出提醒

3.16.1.2类图


Consumable
-observers: List<Observer>
-storage: float
+add(Observer ob): void
+remove(Observer ob): void
+consume(float number): void
Observer
+response(int number): void
Feed
+consume(float number): void
FeedObserver
+response(int number): void

3.17 Prototype Pattern

设计模式简述

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。

3.17.1 Animal实现API

3.17.1.1 API 描述

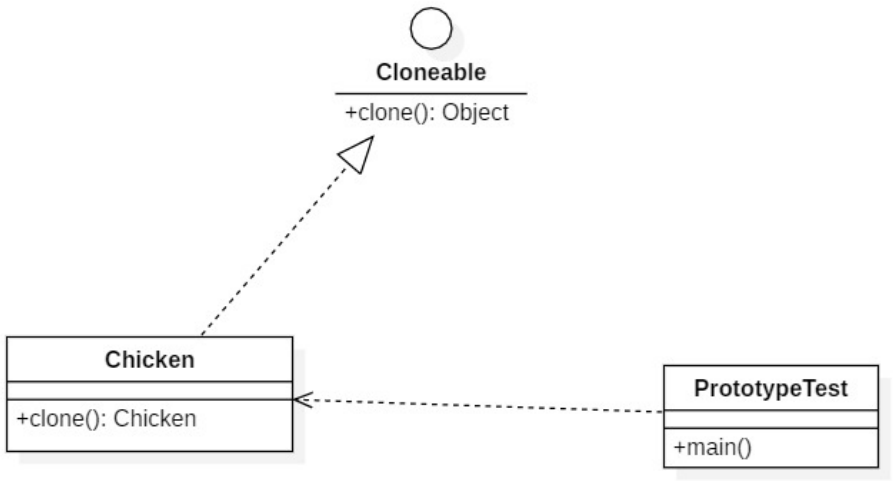
以Animal中的Chicken为例，Chicken在繁殖时会产生一定数量的新的Chicken对象，在刚被生下来时，这些新的Chicken对象除了性别之外，其他的属性都是一样的。如果每出生一个新的Chicken都要进行一次创建的话，其过程比较复杂且麻烦。

可以这样设计，当一只Chicken需要繁殖的时候，调用Chicken的breed方法创建一只新的Chicken，之后的新的Chicken都以这只Chicken为原型调用其clone方法进行复制即可。

由于Java的根类Object提供了clone的方法，并且提供了Cloneable的接口，所以只需要在Chicken类中实现Cloneable接口即可。

函数名	作用
Chicken clone（）	复制一个和当前对象除性别外（随机生成性别）一摸一样的对象

3.17.1.2类图



3.18 Proxy Pattern

设计模式简述

由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。 代理模式的结构比较简单，主要是通过定义一个继承抽象主题的代理来包含真实主题，从而实现对真实主题的访问。

3.18.1 Tool实现API

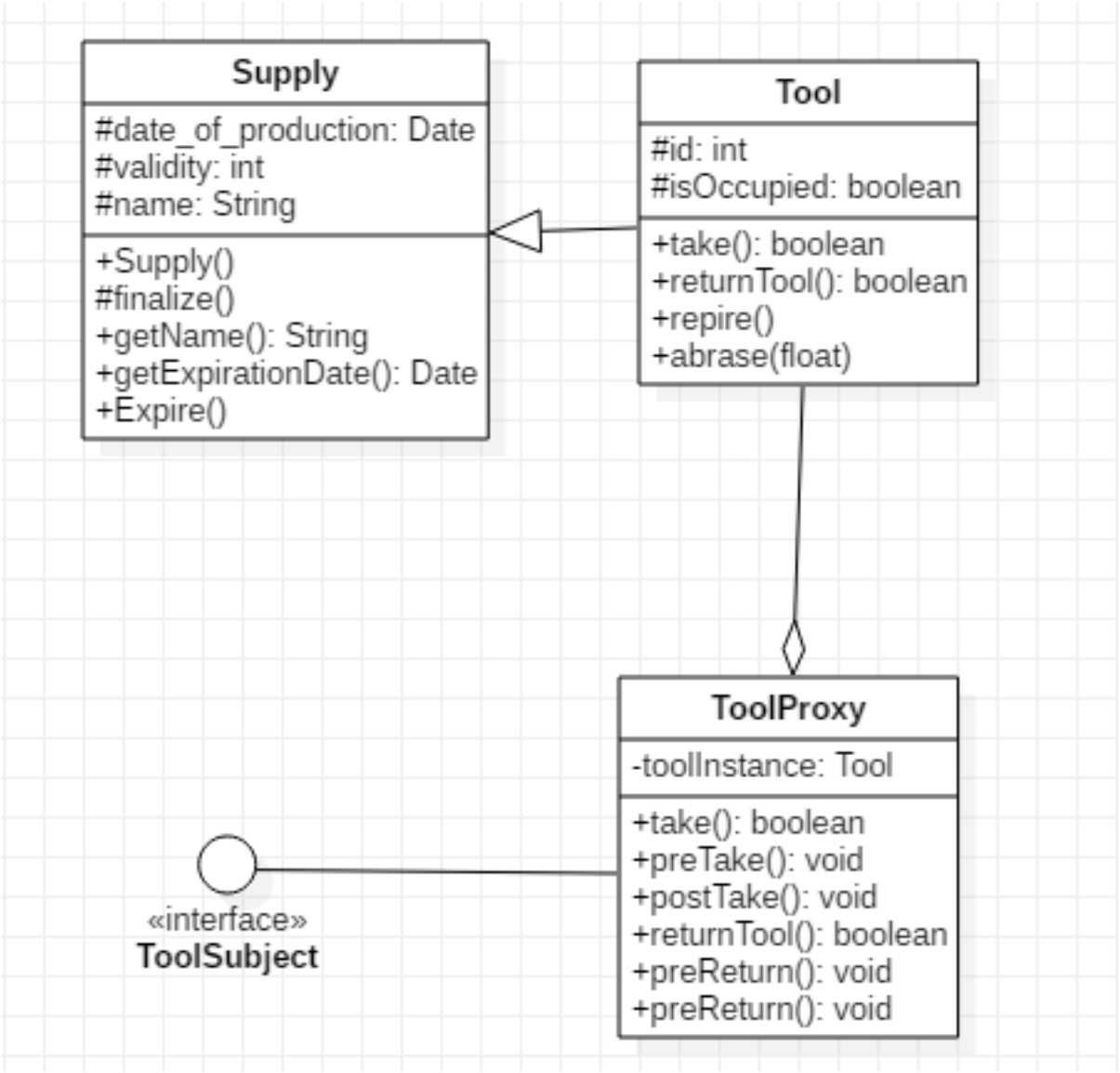
3.18.1.1 API 描述

为tool的借用做一个代理，工人们通过与代理的交互进行工具的获取与归还。

1. 抽象主题（Subject）类：通过接口实现tool的取用和归还两个功能
2. 真实主题（Real Subject）类：即tool类本身
3. 代理（Proxy）类：提供了和tool本身相同的方法，有所扩展

函数名	作用
boolean take()	取用该工具
boolean returnTool()	归还该工具
void preTake()	取用工具的前置处理
void preReturn()	归还工具的前置处理
void postTake()	取用工具的后置处理
void PostReturn()	归还工具的后置处理

3.18.1.2类图


Supply
#date_of_production: Date
#validity: int
#name: String
+Supply()
#finalize()
+getName(): String
+getExpirationDate(): Date
+Expire()
Tool
#id: int
#isOccupied: boolean
+take(): boolean
+returnTool(): boolean
+repire()
+abrase(float)
ToolProxy
-toolInstance: Tool
+take(): boolean
+preTake(): void
+postTake(): void
+returnTool(): boolean
+preReturn(): void
+preReturn(): void
«interface»
ToolSubject

3.19 Singleton Pattern

设计模式简述

单例模式的定义：指一个类只有一个实例，且该类能自行创建这个实例的一种模式。
单例模式有 3 个特点： 单例类只有一个实例对象；

该单例对象必须由单例类自行创建；

单例类对外提供一个访问该单例的全局访问点；

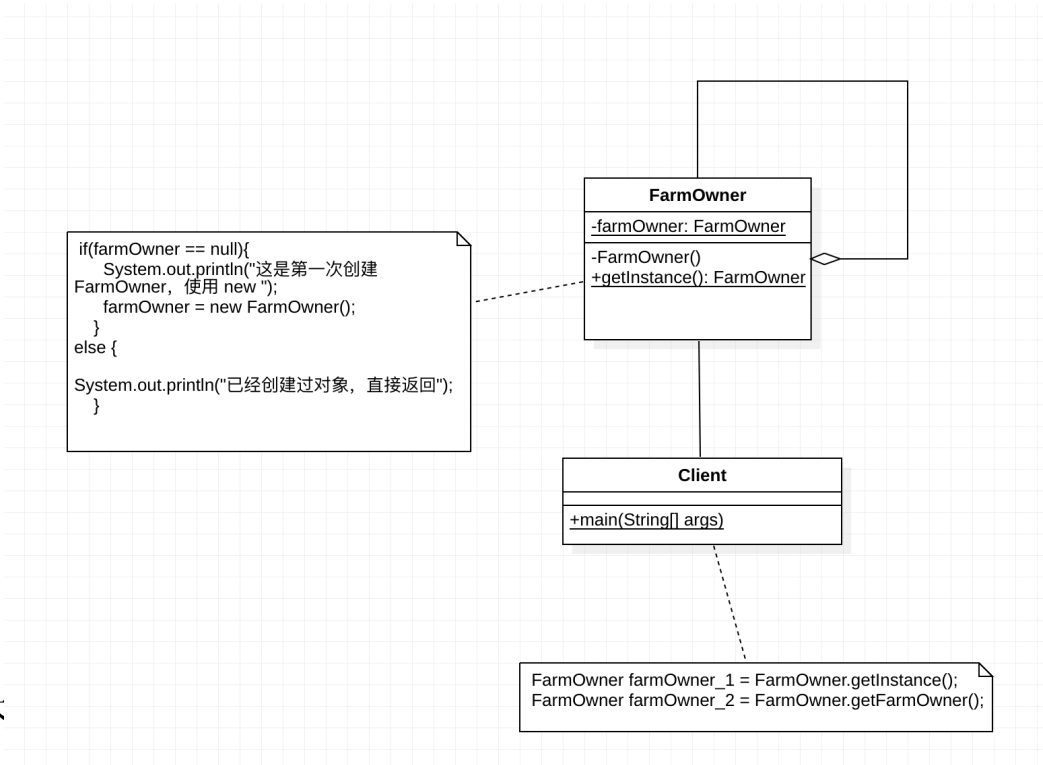
3.19.1 FarmOwner实现API

3.19.1.1 API 描述

考虑到农场只能有一个主人，所以FarmOwner采用Singleton实现。FarmOwner的实例生成只在第一次调用其构造函数的时候使用。使用getInstance函数创建唯一实例。

函数名	作用
getInstance()	创建唯一实例
executeCommand(Command command)	执行指令
FarmOwner()	构造函数，私有函数

3.19.1.2类图



3.20 State Pattern

设计模式简述

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。状态模式的解决思想是：当控制一个对象状态转换的条件表达式过于复杂时，把相关“判断逻辑”提取出来，放到一系列的状态类当中，这样可以把原来复杂的逻辑判断简单化。

3.20.1 植物生长实现API

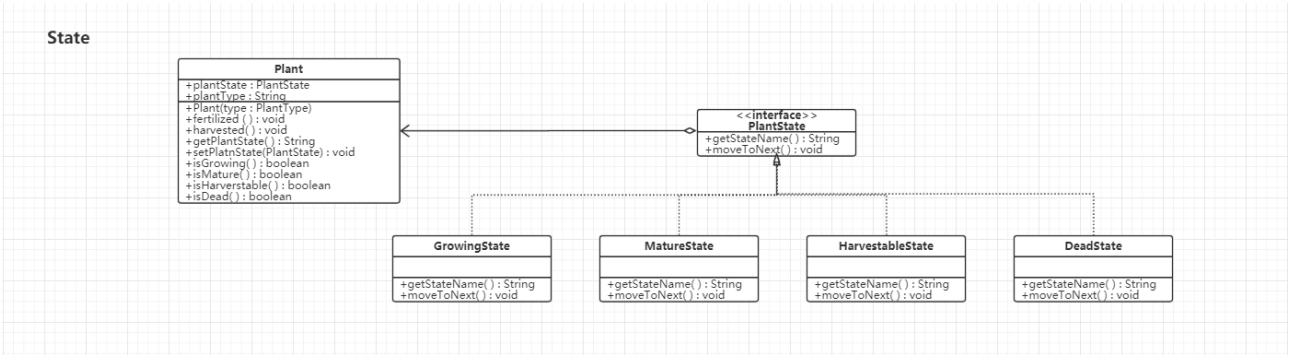
3.20.1.1 API 描述

我们以PlantState为基本状态接口，为植物类（Plant）设立了四种生长状态：

类名	描述
GrowingState	成长阶段
MatureState	成熟阶段
HarvestableState	可收获阶段
DeadState	死亡阶段

四种状态的主要区别在于生长动作moveToNext()方法，同一株植物在不同的生长阶段会有不同的生长方法，如在GrowingState下该方法将会在收到生长命令后输出生长成功的提示，并且重置当前植株的生长状态。

3.20.1.2 类图



3.20.2 Tool状态实现API

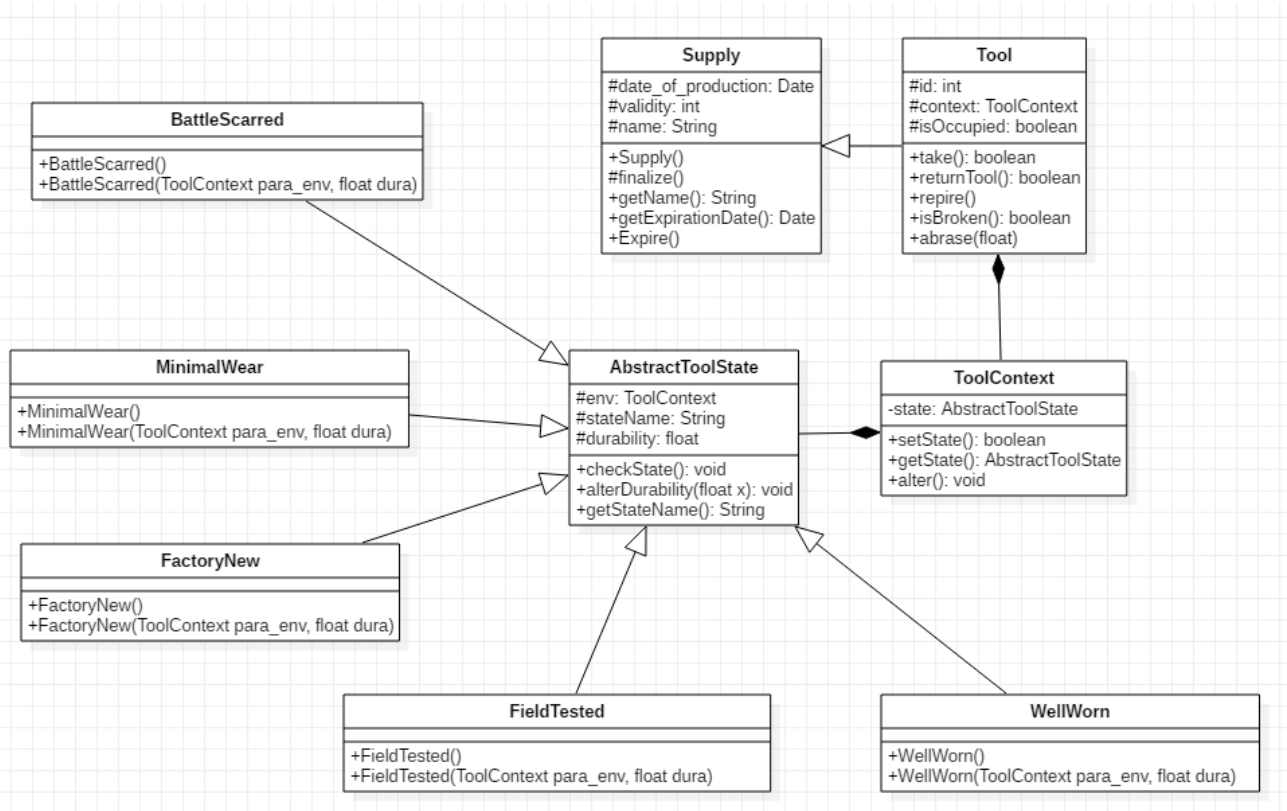
3.20.2.1 API 描述

Tool有根据不同的磨损程度(durability)拥有不同的状态(崭新出厂，略有磨损，久经沙场，破损不堪，战痕累累)，状态的下降会影响其使用效率。

1. 环境（Context）角色：定义一个环境类来描述tool的状态
2. 抽象状态（State）角色：定义一个接口，用以封装环境对象中的特定状态所对应的行为。
3. 具体状态（Concrete State）角色：实现抽象状态所对应的行为。

函数名	作用
void setState()	为该工具环境设置状态
void alter(float val)	改变该工具环境的状态的耐久度
void checkState()	根据该状态的耐久度更改其绑定的环境的状态
ToolContext()	工具环境的构造函数
FactoryNew(ToolContext para_env)	具体状态（崭新出厂）的构造函数，绑定环境实例para_env
MinimalWear(ToolContext para_env)	具体状态（略有磨损）的构造函数，绑定环境实例para_env
FieldTested(ToolContext para_env)	具体状态（久经沙场）的构造函数，绑定环境实例para_env
WellWorn(ToolContext para_env)	具体状态（破损不堪）的构造函数，绑定环境实例para_env
BattleScarred(ToolContext para_env)	具体状态（战痕累累）的构造函数，绑定环境实例para_env

3.20.2.2 类图


Supply
#date_of_production: Date
#validity: int
#name: String
+Supply()
#finalize()
+getName(): String
+getExpirationDate(): Date
+Expire()
Tool
#id: int
#context: ToolContext
#isOccupied: boolean
+take(): boolean
+returnTool(): boolean
+repire()
+isBroken(): boolean
+abrase(float)
BattleScarred
+BattleScarred()
+BattleScarred(ToolContext para_env, float dura)
MinimalWear
+MinimalWear()
+MinimalWear(ToolContext para_env, float dura)
AbstractToolState
#env: ToolContext
#stateName: String
#durability: float
+checkState(): void
+alterDurability(float x): void
+getStateName(): String
ToolContext
-state: AbstractToolState
+setState(): boolean
+getState(): AbstractToolState
+alter(): void
FactoryNew
+FactoryNew()
+FactoryNew(ToolContext para_env, float dura)
FieldTested
+FieldTested()
+FieldTested(ToolContext para_env, float dura)
WellWorn
+WellWorn()
+WellWorn(ToolContext para_env, float dura)

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。

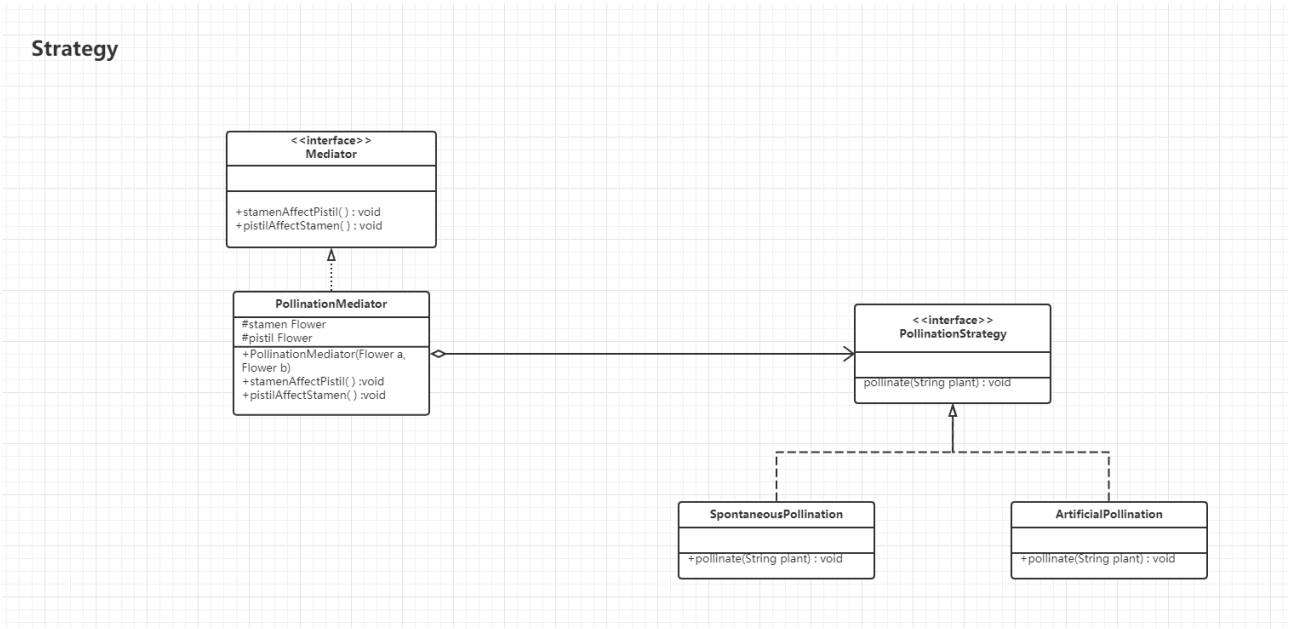
此设计模式分离具体的算法和客户端，使得客户端可以自由切换算法，算法也可以独立于客户端自由进行更改；避免在同一算法中出现大量的条件判断，而是将原本逻辑复杂的算法拆分成多个结构相对简单的独立算法；算法可扩展性良好。但是在结构框架中需要实例化每一个新的策略类，且需要对外暴露所有策略，复杂化了结构。

3.21.1 Plant实现API

3.21.1.1 API 描述

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。

3.21.1.2 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。

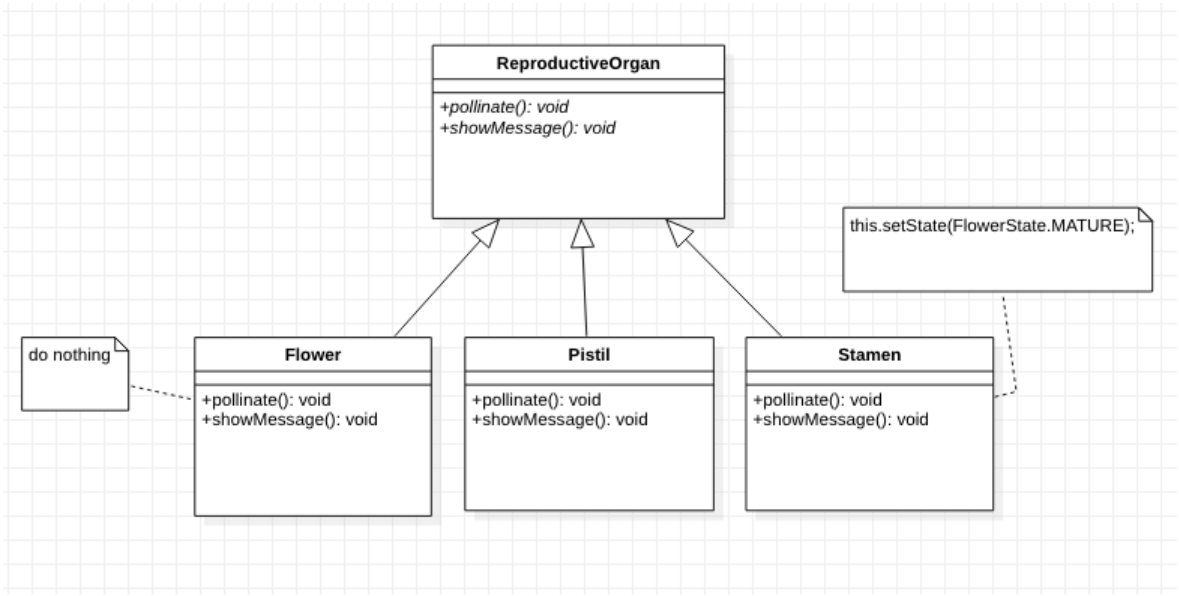
3.22.1 Plant 实现API

3.22.1.1 API描述

对于植物的生殖系统，我们定义了基类ReproductiveOrgan，并在定义了具体的授粉方法pollinate()。在基类ReproductiveOrgan之下我们又具体实现了Flower、Pistil、Stamen三种具体子类，并且为每个子类的pollinate()函数进行了重写，从而为不同植物的生殖器官实现了的不同授粉方式。

函数名	作用
LongTermWorker newWorker()	父类ReproductiveOrgan：授粉抽象方法，具体实现由创建具体子类决定。子类Flower：不做任何处理，因为设置了雌蕊雄蕊，认为花器官不参与这一动作；子类Pistil / Stamen：调整植物的成熟阶段。

3.22.1.2 类图



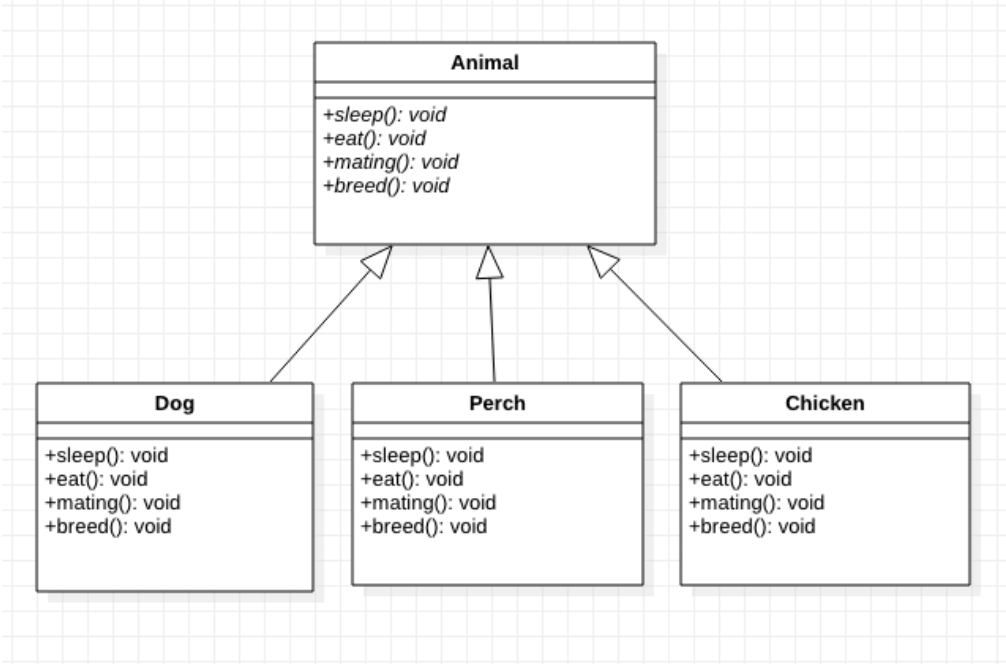
3.22.2 Animal 实现API

3.22.2.1 API描述

定义了动物的基类Animal，并且定义了一系列诸如sleep(), eat(), mating(), breed(), grow()之类的适用于所有动物的操作。不同的动物都会有这种操作，但是对于具体的某一种动物来说，这些操作的细节又不甚相同。将Animal作为抽象父类，以上的方法作为抽象方法，在诸多动物的子类中重新实现。

函数名	作用
父类ReproductiveOrgan：授粉抽象方法，具体实现由创建具体子类决定。子类Flower：不做任何处理，因为设置了雌蕊雄蕊，认为花器官不参与这一动作；子类Pistil / Stamen：调整植物的成熟阶段。	在Animal父类中为所有子类创建动物必须的动作方法，由具体的子类负责实现。

3.22.2.2 类图



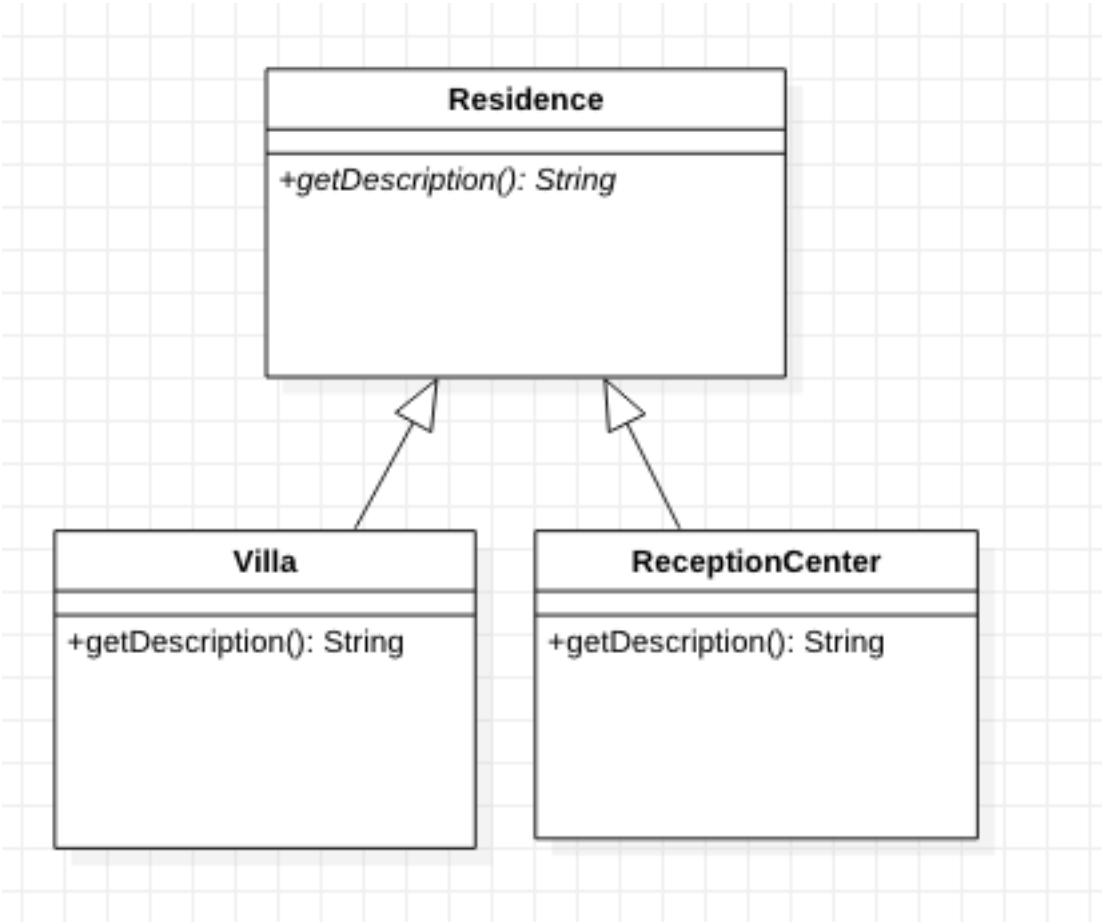
3.22.3 Residence 实现API

3.22.3.1 API描述

定义了一种场景类Resident，并且定义了描述函数getDescription()。不同的房屋种类通过重新实现该描述函数得到独一无二的描述介绍。

函数名	作用
getDescription()	如上所述，通过子类的重载提供每个子类独有的描述信息。

3.22.3.2 类图



3.23 Visitor

设计模式简述

将作用于某种数据结构中的各元素的操作分离出来封装成独立的类，使其在不改变数据结构的前提下可以添加作用于这些元素的新的操作，为数据结构中的每个元素提供多种访问方式。它将对数据的操作与数据结构进行分离，是行为类模式中最复杂的一种模式。

3.23.1 Supply访问实现API

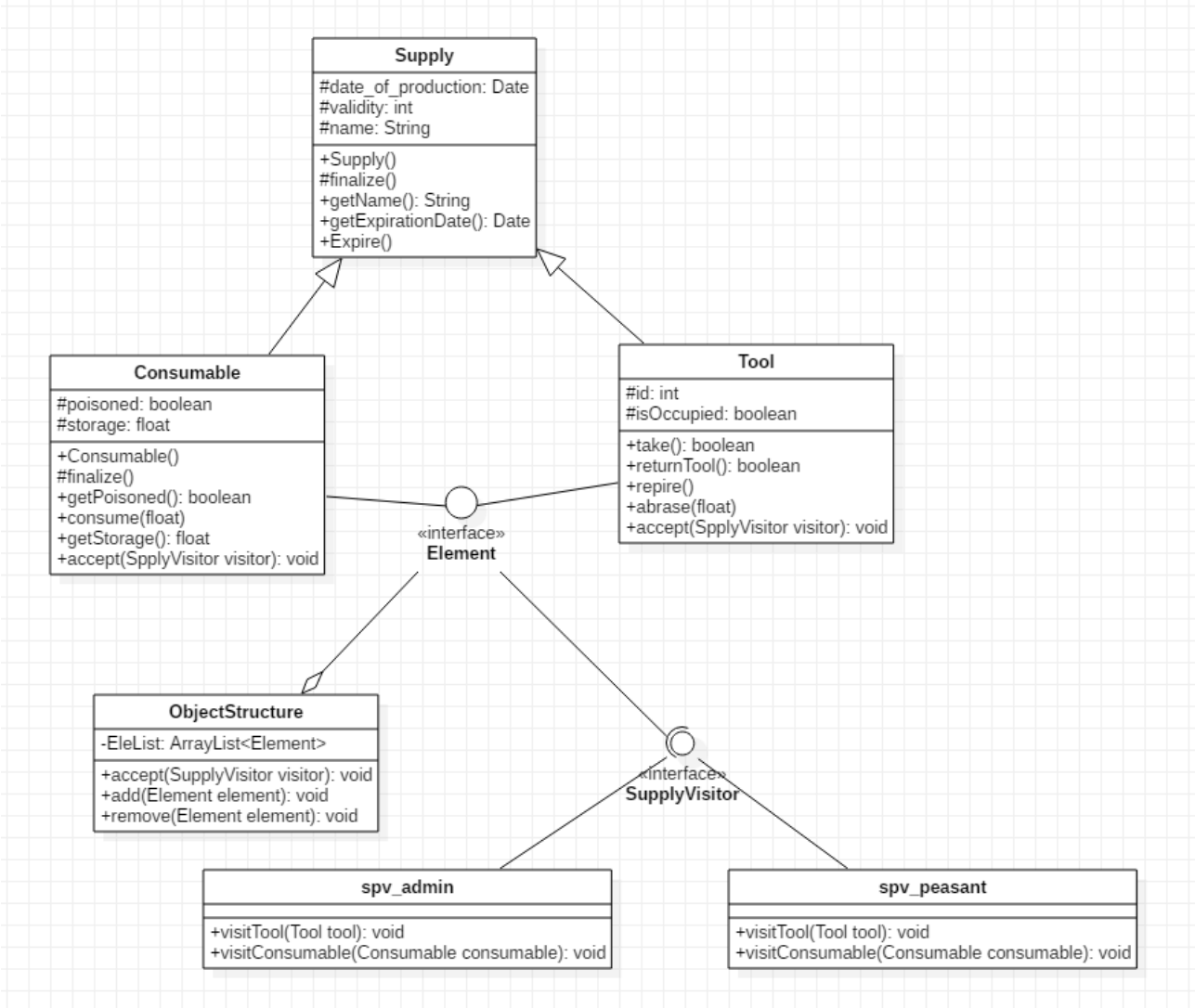
3.23.1.1 API描述

当农夫访问supply时，他们使用消耗品和工具来进行各种耕种活动，他们主要关心对象能否被使用；当管理人员访问supply时，他们关心对象消耗品的量或者是否变质过期，以及工具的耐久度；对Consumable和Tool两个类，不同的人员处理方式不同，故该实例用访问者模式来实现比较合适。

1. 抽象访问者（Visitor）角色：农场人员
2. 具体访问者（ConcreteVisitor）角色：具体的农夫、管理者、工人等等
3. 抽象元素（Element）角色：声明一个包含接受操作 accept() 的接口，被接受的访问者对象作为 accept() 方法的参数。
4. 具体元素（ConcreteElement）角色：实现抽象元素角色提供的 accept() 操作，其方法体通常都是 visitor.visit(this) ，另外具体元素中可能还包含本身业务逻辑的相关操作。
5. 对象结构（Object Structure）角色：是一个包含元素角色的容器，提供让访问者对象遍历容器中的所有元素的方法，通常由 List、Set、Map 等聚合类实现。

函数名	作用
void Element.accept()	Element接受某Visitor访问时调用，被Tool和Consumable重载
void ObjectStructure.accept()	某Visitor遍历访问对象结构中的所有Element
void add()	向对象结构中加入Element
void remove()	在对象结构中删除某个指定Element
void visitTool()	被管理员visitor和农民visitor重载，在Tool.accept()中调用
void visitConsumable()	被管理员visitor和农民visitor重载，在Consumable.accept()中调用

3.23.1.2 类图


Supply
#date_of_production: Date
#validity: int
#name: String
+Supply()
#finalize()
+getName(): String
+getExpirationDate(): Date
+Expire()
Consumable
#poisoned: boolean
#storage: float
+Consumable()
#finalize()
+getPoisoned(): boolean
+consume(float)
+getStorage(): float
+accept(SpplyVisitor visitor): void
Tool
#id: int
#isOccupied: boolean
+take(): boolean
+returnTool(): boolean
+repire()
+abrase(float)
+accept(SpplyVisitor visitor): void
«interface»
Element
ObjectStructure
-EleList: ArrayList<Element>
+accept(SupplyVisitor visitor): void
+add(Element element): void
+remove(Element element): void
«interface»
SupplyVisitor
spv_admin
+visitTool(Tool tool): void
+visitConsumable(Consumable consumable): void
spv_peasant
+visitTool(Tool tool): void
+visitConsumable(Consumable consumable): void

3.24 AOP Pattern

设计模式简述

参考资料

https://zh.wikipedia.org/wiki/%E9%9D%A2%E5%90%91%E5%88%87%E9%9D%A2%E7%9A%84%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1

https://web.archive.org/web/20030716042531/http://aosd.net/index.php

简述

在上学期的数据库课程设计中，我们使用了 asp.net core 中著名的流水线机制，也就是在不需要修改现有代码的基础上，在一个事务的两端添加处理代码，从而抽取公共业务，这就是面向切面编程 AOP 思想的一种体现。在本学期学习 Java EE 课程的过程中，我们又学习了著名的 Java 企业级开发框架 Spring Framework，Spring Framework 的重要指导思想之一也是 AOP。面向切面的程序设计（Aspect-oriented programming，AOP，又译作面向方面的程序设计、剖面导向程序设计）是计算机科学中的一种程序设计思想，旨在将横切关注点与业务主体进行进一步分离，以提高程序代码的模块化程度。通过在现有代码基础上增加额外的通知（Advice）机制，能够对被声明为“切点（Pointcut）”的代码块进行统一管理与装饰。

面向切面的程序设计将代码逻辑切分为不同的模块（即关注点（Concern），一段特定的逻辑功能)。几乎所有的编程思想都涉及代码功能的分类，将各个关注点封装成独立的抽象模块（如函数、过程、模块、类以及方法等），后者又可供进一步实现、封装和重写。部分关注点“横切”程序代码中的数个模块，即在多个模块中都有出现，它们即被称作“横切关注点（Cross-cutting concerns, Horizontal concerns）”。

日志功能即是横切关注点的一个典型案例，因为日志功能往往横跨系统中的每个业务模块，即“横切”所有有日志需求的类及方法体。而对于一个信用卡应用程序来说，存款、取款、帐单管理是它的核心关注点，日志和持久化将成为横切整个对象结构的横切关注点。

在这样的情况下，我们在本次项目中使用了 Java 的动态代理功能，成功完成了一个小型的 AOP 模式实现。

3.24.1 QualityAssuranceTeam、SupplyTeam 实现 API

3.24.1.1 API 描述

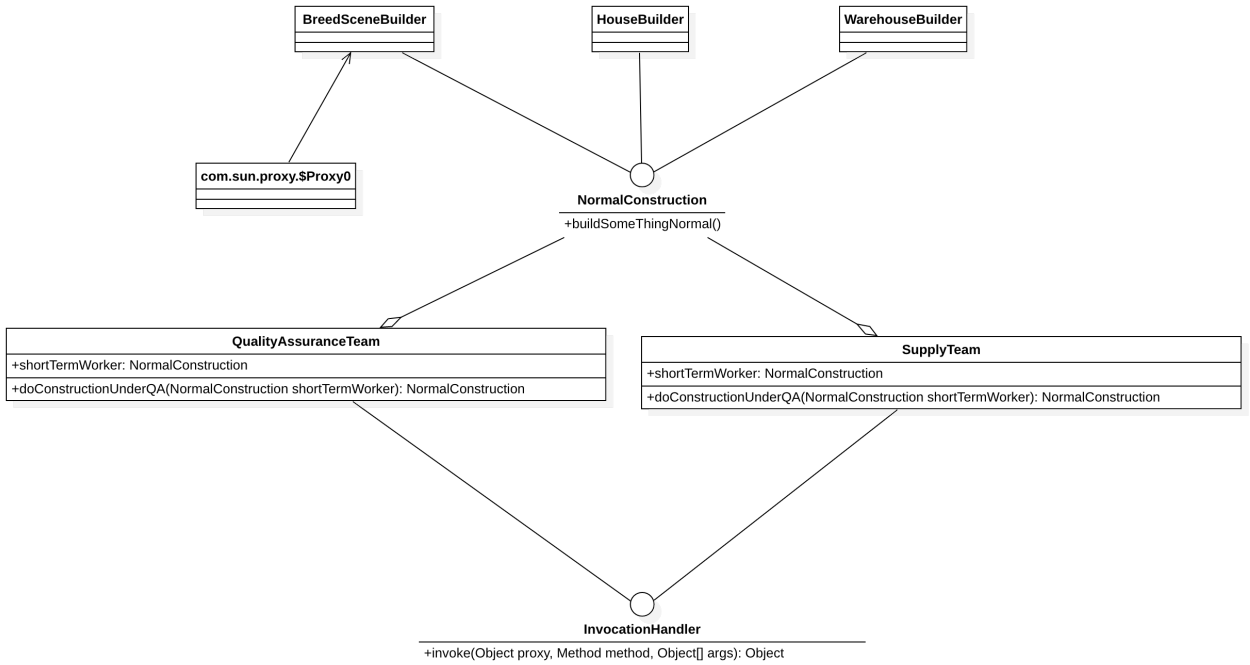
在我们的例子中，我们通过 AOP 的思路抽取出了在所有的工程建设中都具有重要意义的角色：质量保证团队（QualityAssuranceTeam）以及后勤保障团队（SupplyTeam）。它们的工作不论是对于修建养殖场所（BreedSceneBuilder）、住宅场所（HouseBuilder）还是仓库（WarehouseBuilder）来说都是共有的，所以我们应该把相关的实现进行抽取，将它们集合到一个地方。由此，我们创建了 QualityAssuranceTeam 类以及 SupplyTeam 类。这两个类都

继承了 java.lang.reflect.InvocationHandler 接口，从而能够成为动态代理类。具体的函数及其功能如下所示。

函数名	作用
Object invoke(Object proxy, Method method, Object[] args) throws Throwable	本函数是 java.lang.reflect.InvocationHandler 所规定的虚函数，主要用于在用户调用包裹之后的对象的方法时，Java 内部调用从而实现相应的功能。在本例中，我们仅仅做了必要的输出来证明概念，而后就调用了被代理对象的函数来实现确切的功能。
NormalConstruction doConstructionUnderQA(NormalConstruction shortTermWorker)	本函数用于创建代理对象。用户传入需要被代理的 ShortTermWorker 类对象，我们将会返回具有相同接口的代理对象。（返回的对象所属的类是 JVM 在运行时创建的）

3.24.1.2 类图

如上文所述，我们在执行过程中返回的对象并不是我们自己所定义的类的对象，它的命名方式因 JVM 的不同而不同。我们在这里使用 Oracle JVM 的方式进行命名（在 OpenJDK 中就未必如此了，这里只做概念上的展示）



3.25 IOC Pattern

设计模式简述

参考资料

https://martinfowler.com/articles/injection.html

https://zh.wikipedia.org/wiki/%E6%8E%A7%E5%88%B6%E5%8F%8D%E8%BD%AC

https://stackoverflow.com/questions/3058/what-is-inversion-of-control

https://blog.csdn.net/tuke_tuke/article/details/51303467

简述

控制反转最开始是又软件工程的大师 Martin Fowler 提出的，在我们本学期的 Java EE 课程中学习使用的 Spring Framework 中得到的大量的运用。一般来说，乳沟在 Class A 中用到了 Class B 的对象 b，一般情况下，需要在 A 的代码中显式的 new 一个 B 的对象。而正如候捷老师所说，这种编程方式具有相当的问题，因为此时 Class A 和 Class B 紧密的耦合在一起，使得程序的可扩展性严重的下降了（也就是说，这样做违反了重要的 OCP 法则）。而控制反转（Inversion of Control，缩写为IoC），是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。其中最常见的方式叫做依赖注入（Dependency Injection，简称DI），还有一种方式叫“依赖查找”（Dependency Lookup）。通过控制反转，对象在被创建的时候，由一个调控系统内所有对象的外界实体，将其所依赖的对象的引用传递(注入)给它。具体来说，采用依赖注入技术之后，A的代码只需要定义一个私有的B对象，不需要直接new来获得这个对象，而是通过相关的容器控制程序来将B对象在外部new出来并注入到A类里的引用中。而具体获取的方法、对象被获取时的状态由配置文件（如XML）来指定。

IoC 不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是 松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

但是完整实现 IOC 设计模式需要大量的基础设施工作，而这些工作在我们的实际项目中普遍都是交由一些框架来进行处理的（不论是前文提到的 Spring Framework 还是 PHP Laravel 都是这样的），因而我们在本例中实现了本设计模式的一小部分而不是完整的 IOC，具体来说我们的 IOC container 暂时只能注入 Integer、Double 以及 String 等原始类型的成员变量，而不去对用户自定义的类型依赖关系进行手动的处理（那样的处理依赖于较为复杂的数据结构，而脱离了对设计模式本身对认识）。

3.25.1 ClassPathXmlApplicationContext 实现 API

3.25.1.1 API 描述

在我们的例子中，我们通过 XML 文件来定义了两个 WareHouse 对象，并且将它们的引用保存在一个由键值对定义的哈希表中。其中，XML 文件的内容如下所示

```
<beans>
  <bean id="warehouse_1" class="main.java.com.familyFarmSeaside.scene.warehouse.Warehouse">
    <property name="cost" value="65432.1"/>
    <property name="name" value="嘉定一号仓库"/>
    <property name="floorNumber" value="5"/>
  </bean>
  <bean id="warehouse_2" class="main.java.com.familyFarmSeaside.scene.warehouse.Warehouse">
    <property name="cost" value="12345.6"/>
    <property name="name" value="嘉定二号仓库"/>
    <property name="floorNumber" value="2"/>
  </bean>
</beans>
```

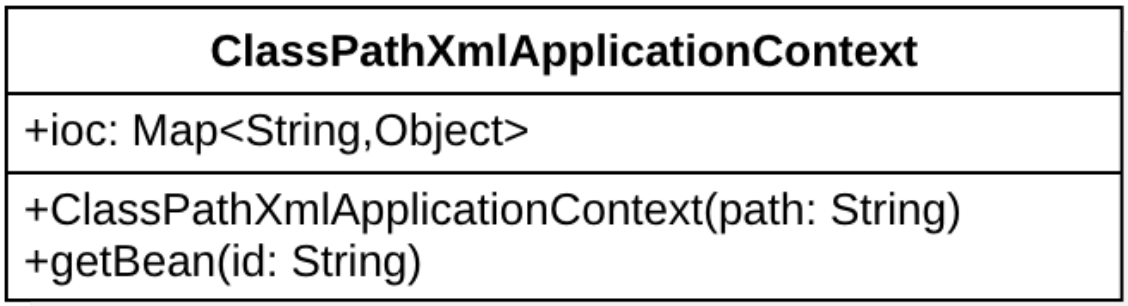
如上所示，我们将由此初始化出来的类的定义如下

```
public class Warehouse extends Scene {
  private double cost;
  private int floorNumber;
  private String name;

  // 这个无参构造函数一定保留，对于 IOC 不可或缺
  public Warehouse(){}
  public void setCost(double cost) {
    this.cost = cost;
  }

  public void setFloorNumber(int floorNumber) {
    this.floorNumber = floorNumber;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

显而易见的，这个类实际上上一个 POJO （**POJO**（Plain Ordinary Java Object）简单的Java对象，实际就是普通JavaBeans，是为了避免和EJB混淆所创造的简称。）它有符合命名规范的 setter 方法，这些方法将被用于 Java 反射从而构造对象。

函数名	作用
ClassPathXmlApplicationContext(String path)	本函数是构造函数，接受参数是用于构造对象的 XML 文件的路径。接受参数之后，本函数将会通过 StAX 接口读取 XML 文件的内容，从而在自身的 Set 中构造相应的对象。
Object getBean(String id)	如上所示的 XML 定义中，用户需要指定相应的对象 ID，这个 ID 在调用本函数时将作为参数传入。

3.25.1.2 类图



3.26 Copy On Write、Reference Counting、Sharable

设计模式简述

写入时复制（Copy-on-write，简称COW）是一种计算机程序设计领域的优化策略。其核心思想是，如果有多个调用者同时请求相同资源（如内存或磁盘上的数据存储），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，而其他调用者所见到的最初的资源仍然保持不变。这过程对其他的调用者都是透明的。

引用计数(Reference Counting)是计算机编程语言中的一种内存管理技术，是指将资源（可以是对象、内存或磁盘空间等等）的被引用次数保存起来，当被引用次数变为零时就将其释放的过程。使用引用计数技术可以实现自动资源管理的目的。

共享(Sharable)用于防止COW过程中重复创建备份。

3.26.1 COW模式实现API

3.26.1.1 API 描述

对一栋宿舍楼来说，它有很多宿舍，每间宿舍里都有床、桌子、椅子等家具。如果在新建一栋宿舍楼时为每间宿舍都创建一套床、桌子、椅子的对象，则会很耗时。考虑到在这些家具在没有被修改之前，所有的宿舍都可以共享一套家具对象（这里的共享并不是说所有的宿舍共用一套家具，而是说共享家具对象）。我们使用工厂类FurnitureFactory 来维护共享逻辑，具体是在工厂类中维护一个关于家具对象的 HashMap 类型furniturePool作为家具池，每当需要的家具对象已经在家具池中时，便可以直接返回，否则创建新的家具对象并将其加入furniturePool中。每当某间宿舍的家具对象需要修改时，提前为该宿舍创建一个副本然后修改。这样以来，可以大大加快新建一栋宿舍的速度，并且如果家具对象没有被修改，就不会有副本被建立，因此多个宿舍只是读取操作时可以共享同一套家具对象。

一个家具对象在新创建后的引用计数为0，每当调用getReference函数时引用计数加一，每当创建备份时原家具对象的引用计数减一。（这里我们假定只有调用家具对象的getReference函数才能进行赋值操作）。

一个家具在创建时是可共享的，在被其他宿舍共享的过程中也是可共享的。不过新建的备份是不可共享的，不然的话每次修改家具对象（移动位置）都要重新创建一个新的备份。

函数名	作用	类
Furniture createFurniture(FurnitureKind kind)	根据给定的家具类型创建家具对象	FurnitureFactory

Furniture getFurnitureByKind(FurnitureKind kind)	当给定的家具类型的对象在furniturePool中已经存在时直接返回，否则新创建该类型的家具对象并将其加入到furniturePool中。	FurnitureFactory
getReference()	返回家具对象的引用，并将其引用计数加一	Furniture
minusReferenceCount()	当家具对象被创建了一个备份后，引用计数减一	Furniture
moveFurniture(FurnitureKind kind)	移动宿舍的家具(修改家具对象), 如果家具对象可共享则创建备份，否则不创建。	Room

3.26.1.2 类图