**Urbanization：有点太城市化了**

——同济大学软件学院2023-24秋 软件设计模式课程项目

卢嘉霖 2050633

宋子阳 2152592

陈莫之 2052750

李昊峰 2152498

李金昊 2151030

王梓鉴 2154157

袁泓博 2152208

郭子瑜 2151760

李锦霖 2152970

张楠 2154154

完成日期 2024-01-01

目录

[1. 项目简介 3](#_Toc155045750)

[2. 设计模式汇总表 4](#_Toc155045751)

[3. 设计模式详述 8](#_Toc155045752)

[3.1 工厂模式 Factory Pattern 8](#_Toc155045753)

[3.1.1 实现描述 8](#_Toc155045754)

[3.1.2 UML图 9](#_Toc155045755)

[3.1.3 核心代码 9](#_Toc155045756)

[3.1.4 优缺点分析 10](#_Toc155045757)

[3.2 抽象工厂模式 Abstract Factory Pattern 10](#_Toc155045758)

[3.2.1 实现描述 10](#_Toc155045759)

[3.2.2 UML图 11](#_Toc155045760)

[3.2.3 核心代码 12](#_Toc155045761)

[3.2.4 优缺点分析 12](#_Toc155045762)

[3.3 单例模式 Singleton Pattern 13](#_Toc155045763)

[3.3.1 实现描述 13](#_Toc155045764)

[3.3.2 UML图 14](#_Toc155045765)

[3.3.3 核心代码 14](#_Toc155045766)

[3.3.4 优缺点分析 15](#_Toc155045767)

[3.4 建造者模式 Builder Pattern 16](#_Toc155045768)

[3.4.1 实现描述 16](#_Toc155045769)

[3.4.2 UML图 17](#_Toc155045770)

[3.4.3 核心代码 17](#_Toc155045771)

[3.4.4 优缺点分析 20](#_Toc155045772)

[3.5 原型模式 Prototype Pattern 21](#_Toc155045773)

[3.5.1 实现描述 21](#_Toc155045774)

[3.5.2 UML图 22](#_Toc155045775)

[3.5.3 核心代码 22](#_Toc155045776)

[3.5.4 优缺点分析 25](#_Toc155045777)

[3.6 适配器模式 Adapter Pattern 25](#_Toc155045778)

[3.6.1 实现描述 26](#_Toc155045779)

[3.6.2 UML图 27](#_Toc155045780)

[3.6.3 核心代码 27](#_Toc155045781)

[3.6.4 优缺点分析 29](#_Toc155045782)

[3.7 桥接模式 Bridge Pattern 30](#_Toc155045783)

[3.7.1 实现描述 30](#_Toc155045784)

[3.7.2 UML图 31](#_Toc155045785)

[3.7.3 核心代码 31](#_Toc155045786)

[3.7.4 优缺点分析 32](#_Toc155045787)

[3.8 过滤器模式 Filter Pattern 32](#_Toc155045788)

[3.8.1 实现描述 32](#_Toc155045789)

[3.8.2 UML图 33](#_Toc155045790)

[3.8.3 核心代码 33](#_Toc155045791)

[3.8.4 优缺点分析 36](#_Toc155045792)

[3.9 组合模式 Composition Pattern 36](#_Toc155045793)

[3.9.1 实现描述 37](#_Toc155045794)

[3.9.2 UML图 37](#_Toc155045795)

[3.9.3 核心代码 37](#_Toc155045796)

[3.9.4 优缺点分析 37](#_Toc155045797)

[3.10 装饰器模式 Decorator Pattern 37](#_Toc155045798)

[3.10.1 实现描述 37](#_Toc155045799)

[3.10.2 UML图 37](#_Toc155045800)

[3.10.3 核心代码 37](#_Toc155045801)

[3.10.4 优缺点分析 37](#_Toc155045802)

[3.11 外观模式 Facade Pattern 38](#_Toc155045803)

[3.11.1 实现描述 38](#_Toc155045804)

[3.11.2 UML图 38](#_Toc155045805)

[3.11.3 核心代码 38](#_Toc155045806)

[3.11.4 优缺点分析 38](#_Toc155045807)

[3.12 责任链模式 Responsibility Chain Pattern 38](#_Toc155045808)

[3.12.1 实现描述 38](#_Toc155045809)

[3.12.2 UML图 39](#_Toc155045810)

[3.12.3 核心代码 39](#_Toc155045811)

[3.12.4 优缺点分析 42](#_Toc155045812)

[3.13 迭代器模式 Iterator Pattern 43](#_Toc155045813)

[3.13.1 实现描述 43](#_Toc155045814)

[3.13.2 UML图 44](#_Toc155045815)

[3.13.3 核心代码 44](#_Toc155045816)

[3.13.4 优缺点分析 48](#_Toc155045817)

[3.14 中介模式 Mediator Pattern 48](#_Toc155045818)

[3.14.1 实现描述 48](#_Toc155045819)

[3.14.2 UML图 49](#_Toc155045820)

[3.14.3 核心代码 49](#_Toc155045821)

[3.14.4 优缺点分析 51](#_Toc155045822)

[3.15 备忘录模式 Memento Pattern 52](#_Toc155045823)

[3.15.1 实现描述 52](#_Toc155045824)

[3.15.2 UML图 52](#_Toc155045825)

[3.15.3 核心代码 53](#_Toc155045826)

[3.15.4 优缺点分析 54](#_Toc155045827)

[3.16 观察者模式 Observer Pattern 54](#_Toc155045828)

[3.16.1 实现描述 54](#_Toc155045829)

[3.16.2 UML图 55](#_Toc155045830)

[3.16.3 核心代码 55](#_Toc155045831)

[3.16.4 优缺点分析 57](#_Toc155045832)

[3.17 状态模式 State Pattern 58](#_Toc155045833)

[3.17.1 实现描述 58](#_Toc155045834)

[3.17.2 UML图 59](#_Toc155045835)

[3.17.3 核心代码 59](#_Toc155045836)

[3.17.4 优缺点分析 61](#_Toc155045837)

[3.18 策略模式 Strategy Pattern 62](#_Toc155045838)

[3.18.1 实现描述 62](#_Toc155045839)

[3.18.2 UML图 64](#_Toc155045840)

[3.18.3 核心代码 64](#_Toc155045841)

[3.18.4 优缺点分析 65](#_Toc155045842)

[3.19 模板模式 Template Pattern 66](#_Toc155045843)

[3.19.1 实现描述 66](#_Toc155045844)

[3.19.2 UML图 67](#_Toc155045845)

[3.19.3 核心代码 67](#_Toc155045846)

[3.19.4 优缺点分析 68](#_Toc155045847)

[3.20 访问者模式 Visitor Pattern 69](#_Toc155045848)

[3.20.1 实现描述 69](#_Toc155045849)

[3.20.2 UML图 70](#_Toc155045850)

[3.20.3 核心代码 70](#_Toc155045851)

[3.20.4 优缺点分析 72](#_Toc155045852)

[3.21 命令模式 Command Pattern 72](#_Toc155045853)

[3.21.1 实现描述 73](#_Toc155045854)

[3.21.2 UML图 74](#_Toc155045855)

[3.21.3 核心代码 74](#_Toc155045856)

[3.21.4 优缺点分析 75](#_Toc155045857)

[4. 附录 76](#_Toc155045858)

# 项目简介

城市化是工业社会发展的主基调。城市化让人类从刀耕火种的农业时代步入万丈高楼平地起的现代社会，在这个过程中，农业产出的充足保障、科技水平的爆炸发展、生产力的不断提高、经济社会的澎湃活力、军事力量的快速壮大以及文化生活的丰富多彩，都是城市化不可或缺的重要方面与关键组成。

尽管作为软件工程专业的学生，我们很难有机会为中国的城市化做出直接贡献，但作为现代社会的一份子，我们都是城市化的见证者。而用我们擅长的软件开发对城市化进行模拟，也不失为一种间接参与的途径。因此，我们基于Unity3D引擎和C#脚本语言，结合20多种软件设计模式，设计了一款名为《Urbanization：有点太城市化了》的游戏。

本项目灵感源自《城市：天际线》《文明6》《钢铁雄心4》等策略、经营、模拟类游戏，从科技、文化、经济等各个方面全方位建模了一座城市的发展历程与脉络。玩家可以在游戏中扮演一名总督，进行城市的设计与建造，如采集和收割自然资源、规划城市功能区块、建造各具特色的建筑、管理城市资源与属性、不断发展与扩张版图等。游戏以单元格为基本操作单位，玩家需要在随机生成的自然地形地图上开始发展，每个单元格具有地形、地貌等初始特征，并将随着游戏进程具有人口、食物产出、科技产出等属性。随着游戏进展，玩家管理的城市将不断发展演化：大兴土木、扩张疆域、提高科技与生产力、丰富文化生活、充实军备等，并最终成为幅员辽阔的帝国之都。

# 设计模式汇总表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **编号** | **设计模式名称** | **涉及的文件** | **何处功能/函数体现** | **备注**  **说明** |
| 1 | 工厂模式Factory Pattern | Factory/Factory/\*Factory.cs | 结合原型模式，生产可克隆的几种对象。 |  |
| 2 | 抽象工厂模式Abstract Factory Pattern | Factory/AbstractFactory/BuildingFactory\_\*.cs | 负责所有建筑的生产。抽象工厂按建筑类型（所属区域）划分。 |  |
| 3 | 单例模式Singleton Pattern | CityInfo/\*.cs, Cache/Cache\_\*.cs等 | 城市资源管理、缓存类、命令栈等全局唯一的类都使用单例模式。 |  |
| 4 | 建造者模式Builder Pattern | Cell/CellBuilder.cs | 单元格的建造分为地形、地貌、资源、状态等多个步骤，使用建造者模式承担。 |  |
| 5 | 原型模式Prototype Pattern | Cache/Cache\_\*.cs | 对区域、地形、地貌、资源这四种没有复杂的属性、变化和继承体系，其子类没有新增属性、继承父类后可以 正常克隆，而又需要大量创建的对象，可以提取创建好并缓存，在需要时对其进行克隆。 |  |
| 6 | 适配器模式Adapter Pattern | Memento/IFileHandler.cs, Memento/MementoManager.cs,Memento/XMLAdapter.cs | 保存地图、存档、数据等文件，用JSON和XML两种格式。需要使用适配器模式进行格式适配。 |  |
| 7 | 桥接模式Bridge Pattern | Cell/Cell.cs | Cell的这些属性需要动态变化，因此用桥接模 式，在运行时通过方法 灵活地修改其动态类型。 |  |
| 8 | 过滤器模式Filter Pattern | Zone/Filter/Filter\_\*.cs | 在建造Zone时需要先判断资源是否足够才能执行消耗资源、建造等逻辑， 将判断过程建模为一系列过滤器组成的链。 |  |
| 9 | 组合模式Composition Pattern | UI/目录下除了Iterator.cs之外的所有文件 | 使用组合模式，让所有UI组件继承相同的类，形成树形结构。UI对象分层级设计，在渲染UI时对根节点UI类调用render()即可，在隐藏UI时对根节点UI类调用hide() 即可。 |  |
| 10 | 装饰器模式Decorator Pattern | Building/Decorator/\*, Factory/AbstractFactory/BuildingFactory\_\*.cs | 建筑的产出值需要根据单元格幸福指数、污染指数两个指标进行百分比变化，使得建筑对象的产出行为(yield()方法)具有额外 行为，因此使用装饰器模式将该设定与具体建筑类解耦。 |  |
| 11 | 外观模式Facade Pattern | Cache/CacheFacade.cs | 外观模式为缓存系统向外提供一 系列易用的接口，封装所有Cache类。客户端只需要一个CacheFacade对象，调用其 clone\*\*\*()方法即可。 |  |
| 12 | 责任链模式Responsibility Chain Pattern | Zone/ResponsibilityChain/Handler\_\*.cs | 将建造区域消耗资源这一过程建模为一系列接受者类的从头到尾合作完成过程。 |  |
| 13 | 迭代器模式Iterator Pattern | UI/UI.cs, Iterator.cs | 根节点UI需要在接受访问方法update\_info()中遍历其Label类型组件，递归调用其这 个方法。使用迭代器模式帮助UI进行遍历:将要遍历的组件顺序写在next()里。 |  |
| 14 | 中介者模式Mediator Pattern | CityInfo/CellManager.cs | 每个单元格只有自身坐标，需要访问 CellManager才能获取其他邻居，把需求委托给中介者 CellManager，可以方便地集中实现获取邻居的逻辑。 |  |
| 15 | 备忘录模式Memento Pattern | Memento/\*.cs | MementoManager进行存档管理，方便保存、删除、读档和存档命名等业务逻辑。备忘录模式通过Memento数 据类抽象了游戏存档所需的所有相关数据，使之容易管理。 |  |
| 16 | 观察者模式Observer Pattern | Cell/Cell.cs, Building/Building.cs | 让单元格的所有建筑对象订阅其所在单元格，当单元格属性发生变化时(被建筑的yield产出触发)自动执行Recalculate()方法，表示被观察 者状态改变了，通知所有建筑对象并调用其yield()来重新计算产出值。 |  |
| 17 | 状态模式State Pattern | State/State\_\*.cs, Cell/Cell.cs | 根据策划，游戏中单元格具有多种状态且其行为取决于状态，这些状态之间通过一些操作相互转化，有些之间不能转换。因此使用状态模式，将各种状态封装为类，将状态之间的转换关系写成方法。 |  |
| 18 | 策略模式Strategy Pattern | Command/Strategy/Strategy\_\*.cs, Command\_ConstructZone.cs | 同样是建造区域命令的执行方法，现在会根据勾选情 况具有不同的执行策略。通过策略模式，将各种策略封装为类、将其业务逻辑写在类的方法里，方便程序运行时改 变策略。 |  |
| 19 | 模板模式Template Pattern | Command/Command\_\*.cs | 每个具体命令的execute()都有一套相似的过程:对单元格、建筑 等进行检查，不符合则返回;然后消耗资源、进行产出、更新单元格状态。通过使用模板模式，每个命令基本不需 要重写execute()方法，只需要按照这套模板改写每 个步骤的函数。 |  |
| 20 | 访问者模式Visitor Pattern | UI/UI\_Component.cs、UI.cs、UI\_updater.cs | UI\_Component的子类都是被访问者，实现update\_info()方法，接收访问 者UI\_Updater的操作。UI\_Updater类实现update()方法，根据访问的组件名进 行相应操作 |  |
| 21 | 命令模式Command Pattern | Command/Command\_\*.cs | 命令模式将操作的发起者(前端组件)和执行者(单元格、区域、建筑等被具体命令持有并使用的对象)解耦。 |  |

# 3. 设计模式详述

## 3.1 工厂模式 Factory Pattern

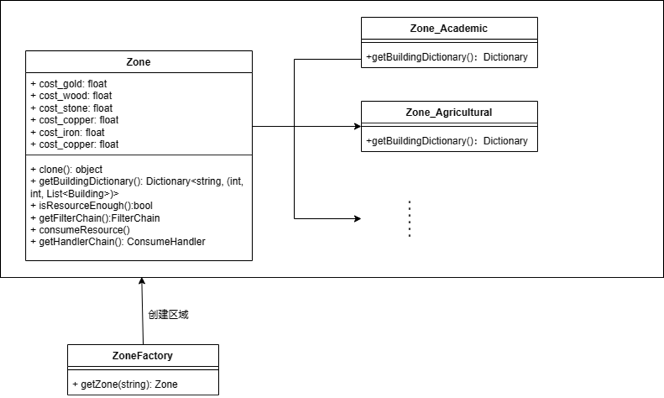
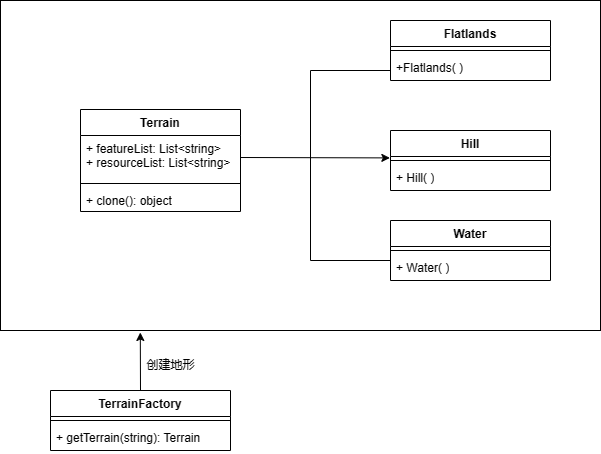
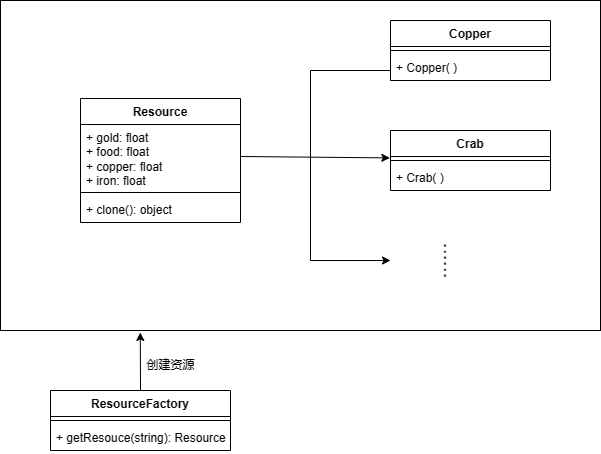
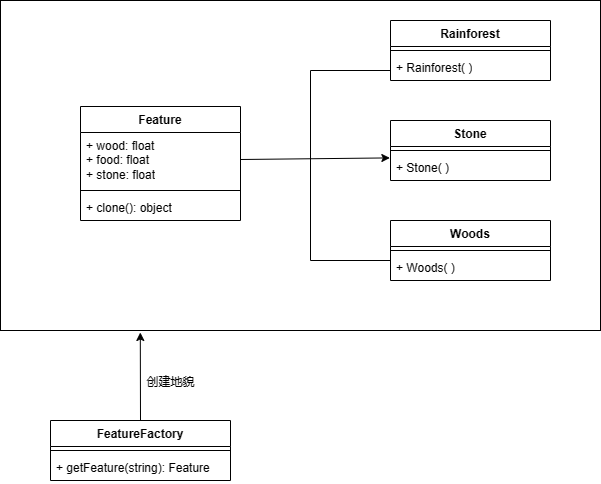
### 3.1.1 实现描述

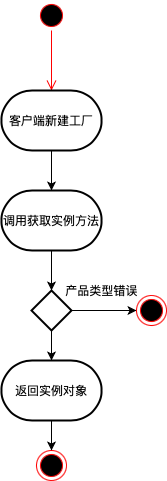
工厂模式是一种创建型设计模式，用于封装和管理对象的创建过程。在工厂模式中，对象的创建被抽象化，不直接使用new关键字来创建对象，而是通过一个工厂类来创建。

工厂模式分离了对象的创建和使用，通过使用工厂类来创建对象，客户端代码可以与具体的类实现细节解耦，使得客户端代码更加简洁和容易维护。同时工厂模式增加了代码的灵活性，当需要更改、增加或替换对象时，只需修改工厂类，而不需要修改使用对象的代码。

在本游戏开发中，需要根据游戏环境动态创建地貌、资源、地形和区域等，因此使用工厂模式，提供了一种高效和灵活的方式来创建和管理游戏世界的复杂元素。 使用工厂模式的这几种类都没有复杂的属性、变化和继承体系，只是基类下有一系列子类、客户端将基类作为静态类型将子类作为动态类型，因此适合用工厂模式。此时客户端只需要用字符串的方式就可以获取对象，而非直接使用“new+类名”的方式。从而使客户端代码与对象构造代码解耦，方便在不知道创建哪种对象时将构造过程推迟到运行时执行。 例如，地貌工厂类FeatureFactory根据不同的游戏环境创建相应的地貌特征。资源工厂ResourceFactory用于生成游戏中的各种资源，如铜、铁、木材，根据游戏区域的特性和玩家的发展阶段来动态地产生不同类型的资源。地形工厂TerrainFactory创建各种地形类型，可以根据当前的游戏世界状态或玩家的行为来适应地形的变化。区域工厂ZoneFactory可以统一管理区域的创建。

### 3.1.2 UML图





### 3.1.3 核心代码

以地貌工厂类FeatureFactory为例，当游戏中需要展示新的环境时，可以通过地貌工厂来生成合适的地貌，而不需要在游戏代码中硬编码这些细节。该类的代码如下

 public class FeatureFactory  
 {  
     public Feature getFeature(string featureType)  
    {  
         return CacheFacade.getInstance().cloneFeature(featureType);  
    }  
 }

该类的主要作用是根据提供的 featureType 参数来获取特定类型的 Feature 对象。类的实现和功能如下：getFeature方法接受一个字符串featureType，用来指定所需的地貌类型，返回Feature类型的对象。具体方式是调用cloneFeature( )方法，以提供的 featureType为参数，创建一个新的 Feature 对象。

### 3.1.4 优缺点分析

**优点：**

1、调用者想创建一个对象时，只需要知道其名称。

2、增加代码扩展性：如果想增加一个产品，只需要扩展一个工厂类。

3、屏蔽产品的具体实现，使调用者仅需要关心产品的接口。

**缺点：**

1、每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。

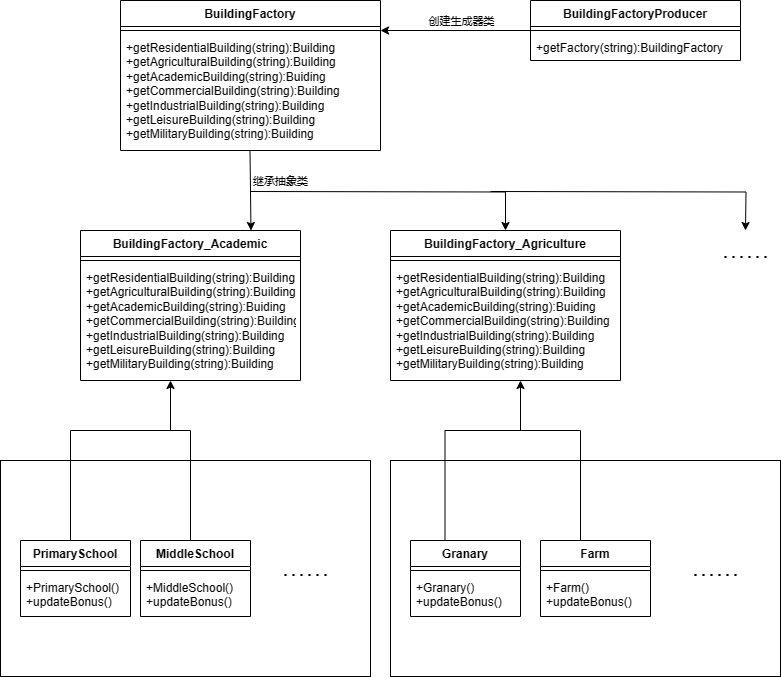
## 3.2 抽象工厂模式 Abstract Factory Pattern

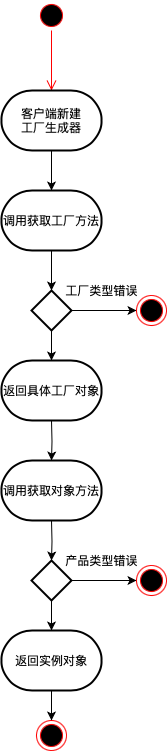
### 3.2.1 实现描述

抽象工厂模式属于创建型模式，含义是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。在抽象工厂模式中，接口是负责创建一个相关对象的工厂，不需要显式指定它们的类。每个生成的工厂都能按照工厂模式提供对象。 该设计模式的功能是，创建一系列相关或相互依赖的对象，而无需指定它们具体的类。一个抽象工厂定义了用于创建一组产品的方法，但将实际创建工作留给其子类来实现。

在该游戏中，建筑有⾮常多种，分属于不同类型的区域，因此使⽤不同类型的⼯⼚来创建建筑对象，⽽每种⼯⼚有需要⼀个上层的BuildingFactoryProducer来⽅便创建，因此使⽤抽象⼯⼚模式。具体实现过程是，客户端通过 zoneType、buildingType两个字符串定位到⼀个建筑对象并获取它，从而解耦了客户端代码和复杂的寻找建筑类的逻辑。

### 3.2.2 UML图





### 3.2.3 核心代码

如下代码所示，由BuildingFactoryProducer类生成的建筑工厂基类BuildingFactory定义了创建不同类型建筑物的抽象方法。每个方法都返回一个 Building 类型的对象，通过建筑类型字符串获取建筑。

 public abstract class BuildingFactory  
 {  
     public abstract Building getResidentialBuilding(string buildingType);  
     public abstract Building getMilitaryBuilding(string buildingType)  
 ……  
 }

该类的功能是创建住宅、农业、学术、商业、工业、休闲和军事等类型建筑。 进一步地，BuildingFactory的⼦类对于不是本区域类型的⽅法报错并返回null，对本区域类型的⽅法则根据字符串构造建筑对象。

### 3.2.4 优缺点分析

**优点：**

1、抽象工厂模式支持一致性。当设计中的多个对象预期要一起使用时，这种模式确保客户端始终仅使用同一产品族中的对象。这对于保持不同产品间的兼容性和协同工作非常重要。

2、有助于分离客户端代码和具体类实现，提升了代码的模块化和可维护性。

**缺点：**

1、使产品族的扩展变得比较困难。若要引入新系列的产品，不仅需要在抽象的创建者（Creator）类中添加新的代码，以定义新的产品创建方法，还必须在具体的工厂实现中添加实际的创建逻辑。这增加了代码的复杂性，并可能迫使开发者修改已有的工厂类。

2、由于抽象工厂通常定义了多个创建方法，任何新添加的产品类型都需要在所有现有工厂实现中得到体现，这可能导致大量重复的代码和维护上的麻烦。

## 3.3 单例模式 Singleton Pattern

### 3.3.1 实现描述

单例模式（Singleton Pattern）是一种常用的软件设计模式，其核心目的是确保一个类只有一个实例，并提供一个全局访问点来获取这个实例。单例模式在很多场景中都非常有用，尤其是当你需要控制某些共享资源的访问，例如配置文件、硬件接口等。

**主要特点**

单一实例：确保类只有一个实例存在。

全局访问点：提供一个全局访问该实例的点，通常是一个静态方法。

**实现方式**

在单例模式的实现中，通常会涉及以下几个步骤：

1. 私有化构造函数：为了避免外部通过new关键字创建多个实例，将构造函数设置为私有。
2. 创建静态私有实例：在类内部创建一个静态的私有实例。
3. 提供静态公共方法：通过一个公共的静态方法返回这个实例，如果实例不存在，则在这个方法内创建它。

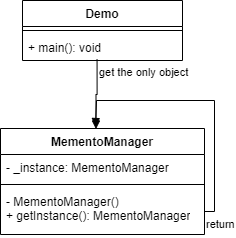
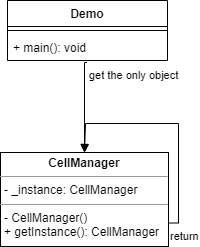
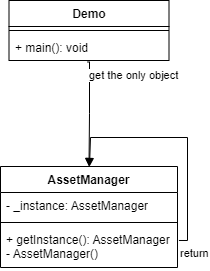
**应用场景**

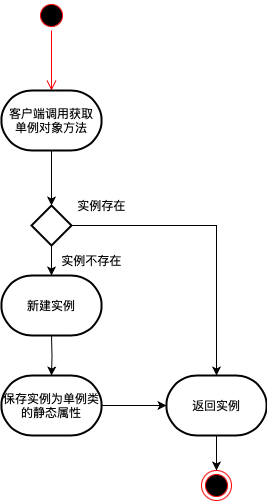
需要频繁实例化然后销毁的对象。

创建对象时耗费资源过多，如访问IO和数据库等资源。

需要定义大量的全局变量和共享资源，如配置信息、全局缓存等。

### 3.3.2 UML图





除了这三个以外，CityInfo/StatisticManager.cs管理城市统计数据、Cache/Cache\_\*.cs所有缓存类和缓存外观类CacheFacade.cs、Command/CommandStack.cs命令栈中也使用了单例模式，这些类逻辑上全局需要且只需要一个，因此用单例模式，类图相似。

### 3.3.3 核心代码

类中只有一个实例，并且提供一个全局访问getInstance（）来获取这个实例。

其他的单例模式代码结构与此相同。

 // 单例模式：管理所有资源  
 public class AssetManager  
 {  
     private static AssetManager \_instance = new AssetManager();  
     private AssetManager() { }  
     public static AssetManager getInstance()  
    {  
         return \_instance;  
    }  
 }

### 3.3.4 优缺点分析

**优点**

1. **资源控制**：单例模式可以严格控制客户端如何以及何时访问某个类的实例，这对于管理资源和控制实例的创建非常有用。
2. **内存占用减少**：由于只创建一个实例，因此减少了系统内存的占用，降低了系统的性能开销。
3. **全局访问点**：提供了一个全局访问点，可以从应用的任何位置访问单例对象，这在某些场景（如配置管理器或日志记录）中非常方便。
4. **共享资源**：可以用于共享资源，如配置数据、数据库连接等，确保所有对象访问的都是同一资源。
5. **实例控制**：可以更好地控制实例的生命周期，比如延迟初始化（Lazy Initialization）。

**缺点**

1. **全局状态**：单例类的全局状态可能导致代码之间的紧密耦合，从而降低代码的可测试性和可维护性。
2. **违反单一职责原则**：单例类通常承担着过多的职责，从对象创建到业务逻辑的管理，这违反了单一职责原则。
3. **扩展困难**：由于单例模式的特性，扩展单例类可能是困难的，这可能限制了类的灵活性和可重用性。
4. **多线程问题**：在多线程环境下，如果单例类没有正确实现，可能会导致多个实例被创建，违反了单例模式的原则。正确实现线程安全的单例模式可能会导致性能下降。
5. **难以测试**：由于单例的全局状态，写单元测试可能会变得更加困难。依赖于单例的代码可能难以模拟或者隔离，从而影响测试的准确性和清晰度。

## 3.4 建造者模式 Builder Pattern

### 3.4.1 实现描述

建造者模式（Builder Pattern）是一种常用的软件设计模式，用于分离复杂对象的构建和表示。这种模式允许你创建不同的表示或设计，同时使用相同的构建过程。建造者模式主要用于构建复杂的对象，特别是当这些对象需要多个步骤来创建时。

**主要组成**

建造者模式通常涉及以下四个核心组件：

产品（Product）：这是最终要构建的复杂对象。产品可能包含多个部分，这些部分在建造者模式中逐步构建。

建造者（Builder）：这是一个抽象接口，用于定义创建产品对象的各个部分的步骤。不同的建造者实现这个接口，提供不同的实现逻辑。

具体建造者（Concrete Builder）：实现Builder接口的类，提供具体的构建逻辑。它们负责创建产品的具体部分，并保持创建过程记录。

指挥者（Director）：负责管理构建过程。它接受一个建造者对象，并通过建造者接口指导构建过程。指挥者知道建造者要创建的对象的构建顺序。

客户端（Client）：客户端是使用这个模式的类。它创建指挥者和具体建造者，并通过指挥者来构建对象。

**工作流程**

客户端创建具体建造者：客户端决定使用哪个具体的建造者实例。

客户端创建指挥者：并将建造者实例提供给指挥者。

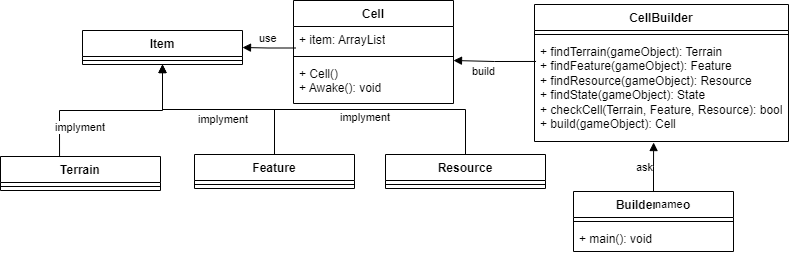
指挥者控制构建过程：指挥者指导建造者开始构建产品的不同部分。

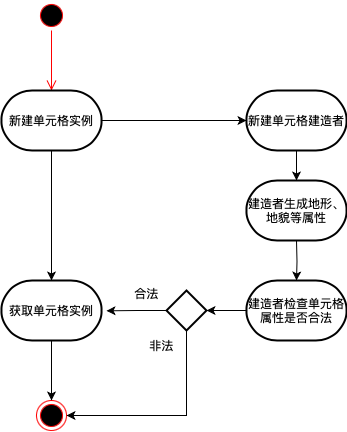
建造者返回产品：一旦产品部件被构建完毕，建造者会将产品返回给客户端。

**应用场景**

建造者模式适用于构建过程复杂的对象，特别是当对象需要多个部分组合而成，且这些部分面临着复杂的变化或扩展时。例如，用于构建复杂的文档（如XML或JSON对象）、UI界面或复杂对象的配置系统等。

### 3.4.2 UML图





### 3.4.3 核心代码

单元格（Cell）对象由于其复杂性，具体体现在地形、地貌、资源等多个属性的组合，这些属性共同能够形成36种不同的类型。为了有效管理这种复杂性，我们采用了建造者模式来负责Cell的构造逻辑。

在Cell类中，我们实现了两种构造函数：一种是无参构造函数，主要用于Unity的序列化过程中自动实例化游戏对象；另一种是带参数的构造函数，专为建造者模式设计。当Unity创建Cell实例时，首先调用无参构造函数，创建一个所有属性均为null的初始对象。随后，Unity会自动调用Awake()方法，该方法在无参构造函数之后执行。在Awake()方法中，我们通过建造者模式的委托来获取Cell的四个初始属性：terrain、feature、resource、state。

建造者模式在此处的主要逻辑包括：

* 1. 创建地形对象、地貌对象和资源对象。这些对象的创建基于Unity中游戏对象的名称。
  2. 检查地形、地貌和资源的组合是否符合预定义的自然单元格描述。
  3. 创建状态对象，这一步骤依据地貌和资源的特点。

通过暴露的build接口，这些步骤被组合起来，用于创建并返回一个带有全部初始属性的Cell临时对象。

在Awake()方法中，临时对象的属性值被赋值到实例本身，完成最终对象的构造。此后，临时对象被销毁，标志着Cell对象构造的完成。通过这种方式，建造者模式使得Cell对象的初始化变得更加模块化和清晰，同时便于管理和维护。

 public class Cell : MonoBehaviour  
 {  
     // 默认构造函数，用于Unity序列化  
     public Cell()  
    {  
 ​  
    }  
 ​  
     // 构造函数，便于CellBuilder构造对象  
     public Cell(State state, Terrain terrain, Feature feature, Resource resource)  
    {  
         this.state = state;  
         this.terrain = terrain;  
         this.feature = feature;  
         this.resource = resource;  
    }  
 ​  
     // 建造者模式：将构造对象的逻辑委托给CellBuilder  
     void Awake()   
    {  
         Cell cell = new CellBuilder().build(gameObject);  
          
         terrain = cell.terrain;  
         feature = cell.feature;  
         resource = cell.resource;  
         state = cell.state;  
         state.cell = this;  // 改变原cell对象中state的引用  
    }  
 }

 // 建造者模式：单元格建造者  
 // 初始单元格具有地形、地貌、资源三个可拓展类型的属性，可组合成多种单元格  
 // 通过建造者模式，将单元格的组合过程与单元格本身分离，使得单元格的组合过程更加灵活  
 public class CellBuilder  
 {  
     Terrain findTerrain(GameObject gameObject)  
    {  
         ///  
    }  
 ​  
     Feature findFeature(GameObject gameObject)  
    {  
         ///  
    }  
 ​  
     Resource findResource(GameObject gameObject)  
    {  
         ///  
    }  
 ​  
     State findState(Cell cell, Feature feature, Resource resource)  
    {  
         ///  
    }  
 ​  
     public Cell build(GameObject gameObject)  
    {  
         Terrain terrain = findTerrain(gameObject);  
         Feature feature = findFeature(gameObject);  
         Resource resource = findResource(gameObject);  
         if(!checkCell(terrain, feature, resource))  
        {  
             Debug.Log("单元格建造失败");  
             return null;  
        }  
 ​  
         Cell cell = new Cell(null, terrain, feature, resource);  
 ​  
         State state = findState(cell, feature, resource);  
         cell.state = state;  
 ​  
         return cell;  
    }  
 }

### 3.4.4 优缺点分析

**优点**

1. **分离构建与表示**：使得同样的构建过程可以创建不同的表示。
2. **封装性**：客户端不需要知道产品内部的组成细节。
3. **构建过程分步骤**：建造者模式允许产品的内部表示可以逐步构建，甚至可以更改构建过程。
4. **更好的控制复杂结构**：当一个对象必须通过多个步骤构建时，建造者模式提供了更好的控制。

**缺点**

1. **增加系统复杂度**：引入多个新类，增加了系统的复杂性。
2. **产品的差异性**：不同的建造者可能产生不同的产品，需要谨慎处理这些差异。

## 3.5 原型模式 Prototype Pattern

### 3.5.1 实现描述

原型模式（Prototype Pattern）是一种创建型设计模式，它用于创建对象的副本，同时保持性能。这个模式特别适用于创建成本高昂的对象场景，通过复制已有对象来避免这些成本。

**原理和组成**

原型模式的核心在于克隆（复制）一个现有对象而不是新建实例，从而减少新实例创建的成本。它通常涉及以下几个组件：

原型（Prototype）：这是一个声明克隆自身方法的接口。通常包含一个用来克隆对象的方法。

具体原型（Concrete Prototype）：实现原型接口的类。这个类定义了复制自身的方式。

客户端（Client）：创建一个新对象的客户端会向原型请求一个克隆，而不是直接从类新建一个对象。

**实现方式**

在实现原型模式时，关键是提供一个克隆方法。在某些语言中（如Python、Java），可以利用内置的克隆支持。克隆可以是浅拷贝也可以是深拷贝：

浅拷贝（Shallow Copy）：只复制对象本身和对象中的值类型字段，而引用类型的字段仅复制引用，不复制引用的对象。

深拷贝（Deep Copy）：复制对象及其所有子对象，确保原始对象和副本完全独立。

**应用场景**

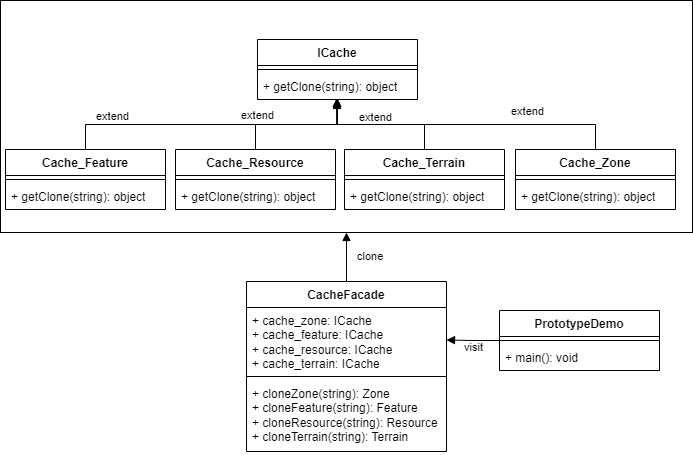
原型模式特别适用于以下情况：

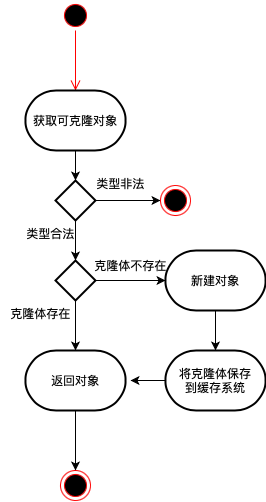
当创建一个对象比复制一个对象更昂贵或更复杂时。

当系统应该独立于它需要创建的对象的类时，例如在动态加载模块的情况下。

当需要实例化的类是在运行时指定时，例如通过动态加载。

### 3.5.2 UML图





### 3.5.3 核心代码

在我们的系统中，区域（Zone）、地形（Terrain）、地貌（Feature）和资源（Resource）这四种类别的对象具有简单的属性结构，没有复杂的变化和继承体系。它们的子类不新增属性，且继承自父类后能够被正常克隆。考虑到需要大量快速创建这些对象，我们采用了原型模式。具体来说，这些类都实现了Cloneable接口（参见Zone/Zone.cs, Cell/Terrain/Terrain.cs, Cell/Feature/Feature.cs, Cell/Resource/Resource.cs）。

为了高效地管理这些对象的实例，我们为每个类别实现了一个缓存类，这些缓存类遵循单例模式。这些缓存类使用类型为<String, xxx>的字典来存储对象的副本。在构造函数中，我们预先创建并存储所有必需的对象。当需要创建新的对象实例时，相应的xxxFactory会调用xxxCache的clonexxx(String xxxName)方法来克隆一个对象。这样，Cache子类承担了所有工厂类的对象创建逻辑，极大地提高了对象创建的效率。

进一步地，为了便于管理这些缓存类，我们采用了外观模式。具体而言，我们创建了一个CacheFacade类。它不仅封装了各种Cache的操作，还负责将克隆出来的object对象转换为期望的具体类型。这种做法简化了客户端对缓存类的操作，使得对象的克隆和类型转换过程更加透明和高效。

 // 原型模式：提前缓存创建的类作为原型，需要时克隆原型。  
 public interface ICache  
 {  
     object getClone(string type);  
 }

 public class CacheFacade  
 {  
     ICache cache\_zone = Cache\_Zone.getInstance();  
     ICache cache\_feature = Cache\_Feature.getInstance();  
     ICache cache\_resource = Cache\_Resource.getInstance();  
     ICache cache\_terrain = Cache\_Terrain.getInstance();  
 ​  
     public Zone cloneZone(string zoneType)  
    {  
         return (Zone)cache\_zone.getClone(zoneType);  
    }  
 ​  
     public Feature cloneFeature(string featureType)  
    {  
         return (Feature)cache\_feature.getClone(featureType);  
    }  
 ​  
     public Resource cloneResource(string resourceType)  
    {  
         return (Resource)cache\_resource.getClone(resourceType);  
    }  
 ​  
     public Terrain cloneTerrain(string terrainType)  
    {  
         return (Terrain)cache\_terrain.getClone(terrainType);  
    }  
 }

下面是原型模式抽象类的实体类：

 public class Cache\_Feature : ICache  
 {  
     public object getClone(string type)  
    {  
         if (!featureDict.ContainsKey(type))  
             return null;  
         return featureDict[type].Clone();  
    }  
 }

 public class Cache\_Resource : ICache  
 {  
     public object getClone(string type)  
    {  
         if (!resourceDict.ContainsKey(type))  
             return null;  
         return resourceDict[type].Clone();  
    }  
 }

 public class Cache\_Terrain : ICache  
 {  
     public object getClone(string type)  
    {  
         if (!terrainDict.ContainsKey(type))  
             return null;  
         return terrainDict[type].Clone();  
    }  
 }

 public class Cache\_Zone : ICache  
 {  
     public object getClone(string type)  
    {  
         if (!zoneDict.ContainsKey(type))  
             return null;  
         return zoneDict[type].Clone();  
    }  
 }

上述类都是从ICache抽象类中拓展出来的实体类。

### 3.5.4 优缺点分析

**优点**

1. **性能优化**：直接复制对象比通过新建对象更有效率，特别是对于复杂对象。
2. **绕开构造函数**：克隆方法允许你绕开复杂的构造过程，直接创建对象。
3. **动态增加或减少产品类**：原型模式允许在运行时添加或删除产品的具体类。

**缺点**

1. **复杂的克隆操作**：如果对象含有循环引用或复杂的结构，克隆可能会很复杂。
2. **克隆与原始对象的关系处理**：需要在克隆和原始对象的关系中保持平衡，尤其是在深拷贝时。
3. **克隆破坏单例模式**：如果克隆的是单例模式的对象，可能会破坏单例模式的原则。

## 3.6 适配器模式 Adapter Pattern

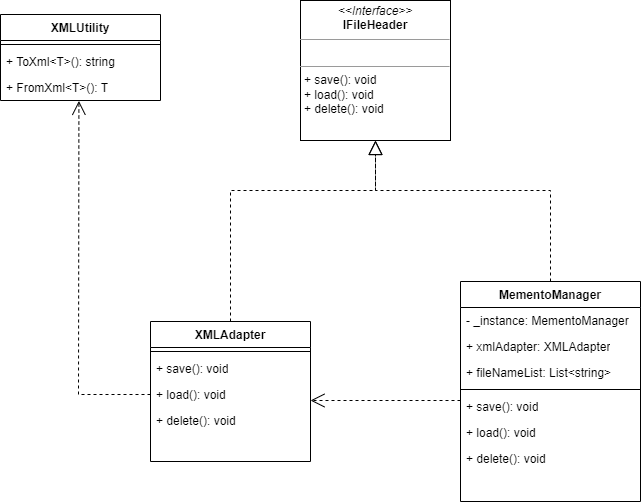
### 3.6.1 实现描述

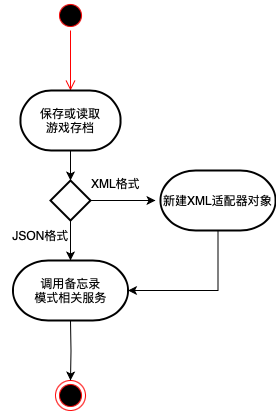
适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。

在这个城市规划类游戏项目中，地图、存档和数据等资源扮演着至关重要的角色。为了降低数据故障的概率，我们采用了JSON和XML两种格式来保存文件，以确保数据的可靠性和兼容性。我们使用了Memento对象来负责保存游戏存档。这个Memento对象负责存储所有关键的游戏数据，包括城市规划、建筑信息等。通过使用Memento模式，我们能够在需要时保存游戏状态，并在需要时还原到先前的状态，提供了灵活性和可靠性；为了管理存档，我们引入了MementoManager作为原生存档管理类。默认情况下，该管理类只支持处理JSON文件。此时，MementoManager通过实现IFileHandler接口，提供了文件的保存（save）、加载（load）、删除（delete）等功能；同时，为了增加对XML格式的支持，我们引入了XMLAdapter，该对象也实现了IFileHandler接口。XMLAdapter充当了适配器的角色，直接完成了Memento对象到.xml磁盘文件的相互转化。XMLUtility.cs用于辅助XMLAdapter的处理过程，使其看起来与JSON的处理过程类似，从而保持了一致的接口。

在整个存档管理系统中，MementoManager负责检测文件格式。如果检测到文件是JSON格式，MementoManager将自行处理；如果是XML格式，MementoManager将利用XMLAdapter处理。这种设计使得游戏系统具备了良好的扩展性和灵活性，使得未来可以轻松地引入其他文件格式的支持。这样的架构有助于游戏项目的可维护性和可扩展性，同时确保了数据的安全性和一致性。

### 3.6.2 UML图





### 3.6.3 核心代码

适配器体系如下。接口IfileHeader定义了统一的动作规范。

 public interface IFileHandler  
 {  
     void save(string fileName, string fileType);  
     void load(string fileName, string fileType);  
     void delete(string fileName, string fileType);  
 }  
 ​

MementorManager实现了IFileHeader来提供文件的保存、下载和删除功能。在默认情况下，MementorManager只能识别并保存JSON格式的文件。XMLAdapter可以保存xml格式的文件，在MementorManager中通过使用XMLAdapter来处理xml格式的文件。

 public class XMLAdapter : IFileHandler  
 {  
     public void save(string fileName, string fileType)  
    {  
  //保存xml格式文件  
    }  
 ​  
     public void load(string fileName, string fileType)  
    {  
         //下载xml格式文件  
    }  
 ​  
     public void delete(string fileName, string fileType)  
    {  
  //删除xml格式文件  
    }  
 }

 public class MementoManager : IFileHandler  
 {  
     //...  
     XMLAdapter xmlAdapter;  
     public List<string> fileNameList = new List<string>();  
 ​  
     public void save(string fileName, string fileType)  
    {  
         //...  
         else if (fileName == "xml")  
        {  
             // 使用XML适配器对象  
             xmlAdapter = new XMLAdapter();  
             xmlAdapter.load(fileName, fileType);  
             return;  
        }  
    }  
 ​  
     public void load(string fileName, string fileType)  
    {  
         //...  
         else if (fileName == "xml")  
        {  
             // 使用XML适配器对象  
             xmlAdapter = new XMLAdapter();  
             xmlAdapter.load(fileName, fileType);  
             return;  
        }  
    }  
 ​  
     public void delete(string fileName, string fileType)  
    {  
  //...  
         else if (fileName == "xml")  
        {  
             // 使用XML适配器对象  
             xmlAdapter = new XMLAdapter();  
             xmlAdapter.delete(fileName, fileType);  
 ​  
             return;  
        }  
    }  
 }

### 3.6.4 优缺点分析

**优点：**

1. 可以让任何两个没有关联的类一起运行。
2. 提高了类的复用。
3. 增加了类的透明度。
4. 灵活性好。

**缺点：**

1. 过多地使用适配器，会让系统非常零乱，不易整体进行把握。
2. 由于 JAVA 至多继承一个类，所以至多只能适配一个适配者类，而且目标类必须是抽象类。
3. 增加了类与类的耦合

## 3.7 桥接模式 Bridge Pattern

### 3.7.1 实现描述

桥接（Bridge）是用于把抽象化与实现化解耦，使得二者可以独立变化。这种类型的设计模式属于结构型模式，它通过提供抽象化和实现化之间的桥接结构，来实现二者的解耦。

这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类，这两种类型的类可被结构化改变而互不影响。

桥接模式的目的是将抽象与实现分离，使它们可以独立地变化，该模式通过将一个对象的抽象部分与它的实现部分分离，使它们可以独立地改变。它通过组合的方式，而不是继承的方式，将抽象和实现的部分连接起来。

根据策划，Cell单元格类有这几个基础属性：

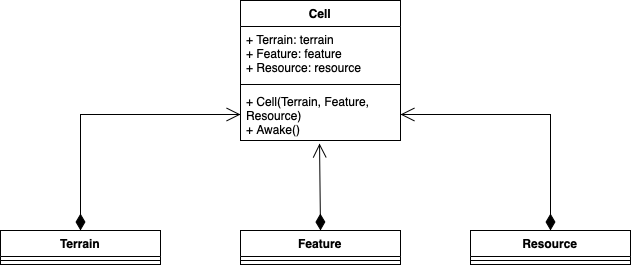
Terrain：地形基类，派生类包括Flatlands，Hill，Water

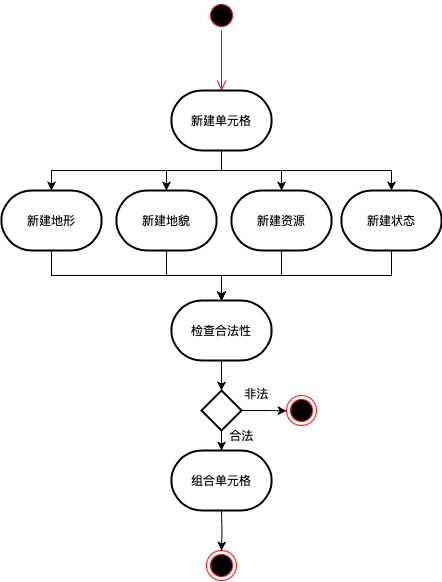
Feature：地貌基类，派生类包括Stone，Woods，Rainforest

Resource：地貌基类，派生类包括Pig，Sheep，Horse等9种子类

再根据它们的匹配规则，初始的Cell有1\*4+2\*4\*4=36种类型。如果用Cell子类表示，数量多、难以扩展。同时，Cell的这些属性需要动态变化，如收获资源后Resource=null，种树后Feature = new Woods()。因此用桥接模式，让Cell持有Terrain、Feature、Resource三个属性，在构造时初始化（通过建造者模式），在运行时通过方法灵活地修改其动态类型。这样，通过让三个维度独立变化，将它们相互解耦，极大减少了类数量、提高了可扩展性。

### 3.7.2 UML图





### 核心代码

Cell类持有Terrain, Feature, Resource三个表示自然单元格初始属性的对象，并用合适的方法将这些属性赋初值，从而用组合替代了继承，大量省去了不必要的派生类。

 public class Cell : MonoBehaviour  
 {  
     public Terrain terrain;  
     public Feature feature;  
     public Resource resource;  
  ...  
 ​  
     public Cell(State state, Terrain terrain, Feature feature, Resource resource)  
    {  
         this.state = state;  
         this.terrain = terrain;  
         this.feature = feature;  
         this.resource = resource;  
    }  
 ​  
     void Awake()   
    {  
         Cell cell = new CellBuilder().build(gameObject);  
          
         terrain = cell.terrain;  
         feature = cell.feature;  
         resource = cell.resource;  
         state = cell.state;  
         state.cell = this;  
    }  
 }

### 3.7.4 优缺点分析

**优点：**

1. 你可以创建与平台无关的类和程序。
2. 客户端代码仅与高层抽象部分进行互动，不会接触到平台的详细信息。
3. 开闭原则。 你可以新增抽象部分和实现部分， 且它们之间不会相互影响。
4. 单一职责原则。 抽象部分专注于处理高层逻辑， 实现部分处理平台细节。

**缺点：**

1. 对高内聚的类使用该模式可能会让代码更加复杂。

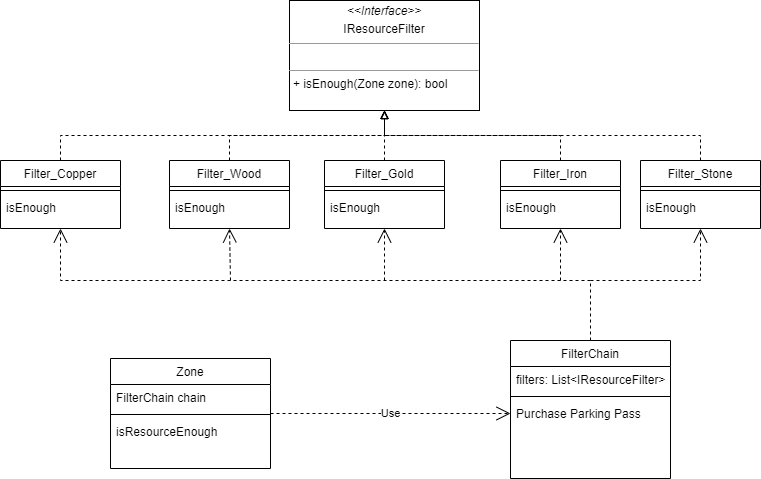
## 3.8 过滤器模式 Filter Pattern

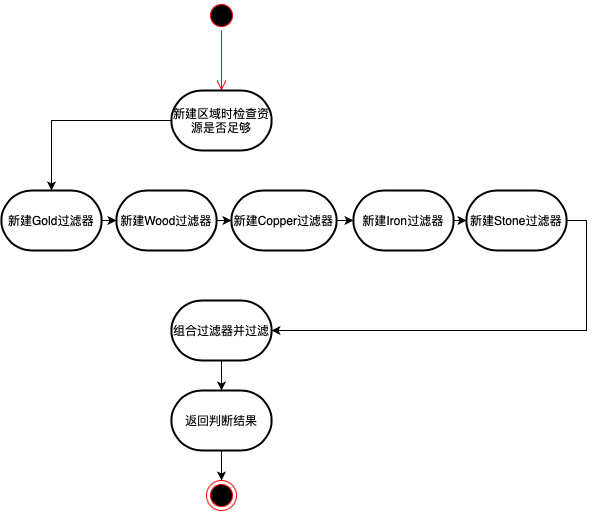
### 3.8.1 实现描述

过滤模式是一种设计模式，这种模式允许开发人员使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们连接起来。这种类型的设计模式属于结构型模式，它结合多个标准来获得单一标准。

在游戏开发中，经常需要处理建造zone的逻辑，其中关键的一步是在执行消耗资源和建造等后续动作之前，先判断是否拥有足够的资源。为了更好地组织和解耦这一判断过程，我们引入了一系列过滤器构成的链。每个过滤器类都实现了IResourceFilter接口中的bool isEnough方法，用于判断其负责的资源种类是否足够，返回相应的布尔值结果。这种设计能够灵活地扩展和修改资源判断逻辑，而不影响其他部分的代码。为了管理这一系列过滤器，我们引入了FilterChain类，该类维护了一个过滤器列表，并提供了isEnough()接口遍历过滤器列表，获取每个过滤器的判断结果，并根据这些结果得出最终的资源是否足够。在Zone类中，判断资源是否足够的方法逻辑与具体的资源判断逻辑解耦，只需构造一个FilterChain，将一系列需要用到的资源过滤器放入链中，然后调用chain的isEnough()方法即可完成资源足够性的判断。这种设计使得代码更加清晰、可维护，并支持灵活的资源判断策略。

### 3.8.2 UML图





### 3.8.3 核心代码

过滤器体系如下。接口IResourceFilter定义了统一的动作规范。

 public interface IResourceFilter  
 {  
     // 判断资源是否充足  
     public bool isEnough(Zone zone);  
 }

Filter\_Copper、Filter\_Gold、Filter\_Iron、Filter\_Stone和Filter\_Wood类实现了接口IResourceFilter，分别对不同的资源进行判断。

 public class Filter\_Copper : IResourceFilter  
 {  
     // 判断铜是否充足  
     public bool isEnough(Zone zone)  
    {  
         return AssetManager.getInstance().Copper\_Stock >= zone.cost\_copper;  
    }  
 }  
 ​  
 public class Filter\_Gold : IResourceFilter  
 {  
     // 判断金是否充足  
     public bool isEnough(Zone zone)  
    {  
         return AssetManager.getInstance().Gold\_Stock >= zone.cost\_gold;  
    }  
 }  
 ​  
 public class Filter\_Iron : IResourceFilter  
 {  
     // 判断铁是否充足  
     public bool isEnough(Zone zone)  
    {  
         return AssetManager.getInstance().Iron\_Stock >= zone.cost\_iron;  
    }  
 }  
 ​  
 public class Filter\_Stone : IResourceFilter  
 {  
     // 判断石头是否充足  
     public bool isEnough(Zone zone)  
    {  
         return AssetManager.getInstance().Stone\_Stock >= zone.cost\_stone;  
    }  
 }  
 ​  
 public class Filter\_Wood : IResourceFilter  
 {  
     // 判断木头是否充足  
     public bool isEnough(Zone zone)  
    {  
         return AssetManager.getInstance().Wood\_Stock >= zone.cost\_wood;  
    }  
 }

FilterChain类中创建链表来存储对不同类的判断，zone添加不同的过滤器来判断不同的建造是否可行。

 public class FilterChain {  
     private List<IResourceFilter> filters = new List<IResourceFilter>();  
     // 加入要过滤的类  
     public void addFilter(IResourceFilter filter) {  
         filters.Add(filter);  
    }  
     public bool isEnough(Zone zone) {  
         foreach (IResourceFilter filter in filters) {  
             if (!filter.isEnough(zone)) {  
                 return false;  
            }  
        }  
         return true;  
    }  
 }

最后，在Zone类中调用FilterChain来判断不同的建造是否满足条件，从而筛选出批量建造过程中哪些建造行为是可行的。

 public abstract class Zone : ICloneable  
 {  
     // ...  
     FilterChain chain = new FilterChain();  
     // 过滤器模式：检查建造区域的资源是否足够  
     public bool isResourceEnough()  
    {  
         return getFilterChain().isEnough(this);  
    }  
     // ...  
 }  
 ​

### 3.8.4 优缺点分析

**优点：**

1. **解耦性**： 过滤器模式可以将过滤逻辑从目标对象中分离出来，使得过滤器和目标对象之间的耦合度降低。这有助于提高代码的灵活性和可维护性。
2. **可扩展性**： 可以很容易地添加新的过滤器，而不需要修改现有的代码。这种可扩展性使系统更容易适应变化，并且新的过滤条件可以方便地添加到系统中。
3. **可复用性**： 过滤器可以在不同的上下文中重复使用。如果你有不同的对象集合，可以使用相同的过滤器逻辑来筛选它们，提高代码的重用性。
4. **易于理解**： 过滤器模式提供了一种清晰的方式来定义过滤条件，并将其应用于对象集合。这使得代码更易于理解和维护。

**缺点：**

1. **性能问题**：当有大量数据需要过滤时，可能会引起性能问题，特别是在使用多个过滤器的情况下。每个过滤器都需要遍历整个数据集，可能导致性能下降。
2. **过多的类**：如果有许多不同的过滤条件，可能会导致系统中存在大量的过滤器类，这可能增加代码的复杂性。
3. **复杂性增加**：对于简单的过滤需求，引入过滤器模式可能显得过于繁琐和复杂，可能会增加代码的复杂性而不是简化它。

## 3.9 组合模式 Composition Pattern

### 3.9.1 实现描述

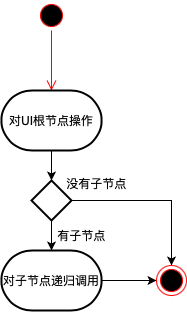
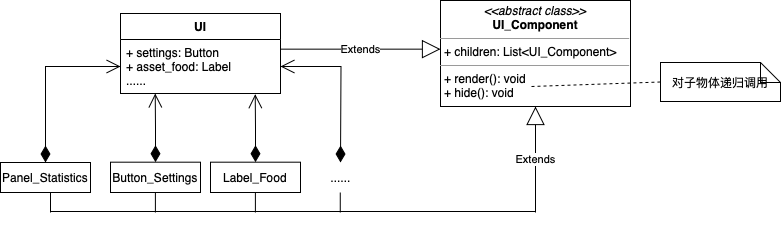
组合模式是一种结构性设计模式，它允许你将对象组合成树形结构以表示"部分-整体"的层次结构。通过使用组合模式，客户端可以统一处理单个对象和对象组合，而无需区分它们的类型。

在组合模式中，有两种主要类型的对象：叶节点和容器节点。叶节点表示树结构中的最终对象，而容器节点表示包含子节点的对象。这两种节点都实现相同的接口，使得客户端可以以统一的方式处理它们。

以下是组合模式的主要元素和工作原理：

1. **Component（组件）**：定义组合模式中所有对象的通用接口，确保叶节点和容器节点都可以通过相同的方式进行操作。
2. **Leaf（叶节点）**：实现Component接口的叶节点表示树结构中的最终对象。叶节点没有子节点，它们执行具体的操作。
3. **Composite（容器节点）**：实现Component接口的容器节点表示包含子节点的对象。容器节点可以包含叶节点和其他容器节点。它们通常将操作委托给其子节点。

### 3.9.2 UML图



### 3.9.3 核心代码

由于一个UI组件下面可能由多个UI组件构成，这种复杂的层次结构如果直接使用会非常的麻烦，我们使用组合模式可以简单的通过调用整体来实现调用其树的子节点。

在渲染或者操作UI\_Component的时候，统一的调用可以使得程序像处理简单元素一样处理复杂元素。

 public class UI\_Component : MonoBehaviour     
 {  
     List<UI\_Component> children = new List<UI\_Component>();  
 ​  
     void Start()  
    {  
         // 找到所有子物体  
         foreach (Transform child in transform)  
        {  
             UI\_Component child\_component = child.GetComponent<UI\_Component>();  
             if (child\_component != null)  
            {  
                 children.Add(child\_component);  
            }  
        }  
    }  
 ​  
     public void render()  
    {  
         // 渲染自身  
         gameObject.SetActive(true);  
 ​  
         // 渲染子节点  
         foreach (UI\_Component child in children)  
        {  
             child.render();  
        }  
    }  
 ​  
     public void hide()  
    {  
         // 隐藏自身  
         gameObject.SetActive(false);  
 ​  
         // 隐藏子节点  
         foreach (UI\_Component child in children)  
        {  
             child.hide();  
        }  
    }  
 ​  
     // 访问者模式的accept()方法  
     public virtual void update\_info(UI\_Updater updater)  
    {  
 ​  
    }  
 }

对于Start()，render()和hide()三个方法，使用组合模式可以实现只操作一个对象就实现了多个对象的操作，父节点会递归的去调用子节点的方法。这样就可以实现客户程序和复杂元素的内部解耦。

### 3.9.4 优缺点分析

**优点：**

1. **统一接口：** 组合模式定义了统一的接口，使得客户端可以一致地处理单个对象和对象组合。这简化了客户端代码，使其更加清晰和易于理解。
2. **透明性：** 客户端无需关心正在处理的是叶节点还是容器节点，因为它们都实现了相同的接口。这提高了系统的透明性，减少了客户端的复杂性。
3. **代码重用：** 由于叶节点和容器节点共享相同的接口，可以更容易地重用现有的代码。
4. **容易组织复杂结构：** 组合模式特别适用于组织复杂的树形结构，例如图形用户界面或文件系统。
5. **灵活性和可扩展性：** 可以轻松地增加新的组件，扩展系统的功能。通过添加新的叶节点或容器节点，可以更灵活地构建树形结构。

**缺点：**

1. **过度一般化：** 有时候，为了实现通用性和一致性，组合模式可能会导致过度一般化，增加了一些不必要的复杂性。
2. **性能考虑：** 在处理大型树结构时，可能会引起性能问题，因为递归访问和处理每个节点可能会导致性能开销。
3. **复杂性：** 对于相对简单的结构，引入组合模式可能会显得繁琐。在这种情况下，使用组合模式可能会导致不必要的复杂性。
4. 在使用组合模式时，其叶子和树枝的声明都是实现类，而不是接口，违反了依赖倒置原则。

## 3.10 装饰器模式 Decorator Pattern

### 3.10.1 实现描述

装饰器模式是一种结构性设计模式，它允许在不改变对象接口的情况下动态地添加功能。这种模式是通过创建一个包装类，也就是装饰器，来包装原始类的实例来实现的。

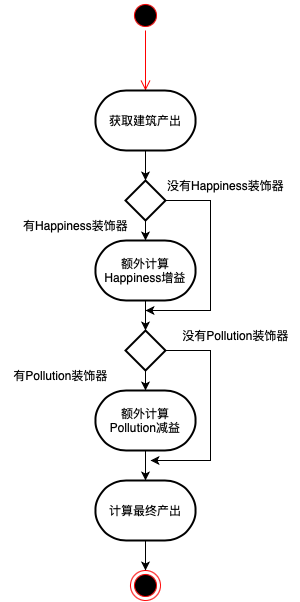
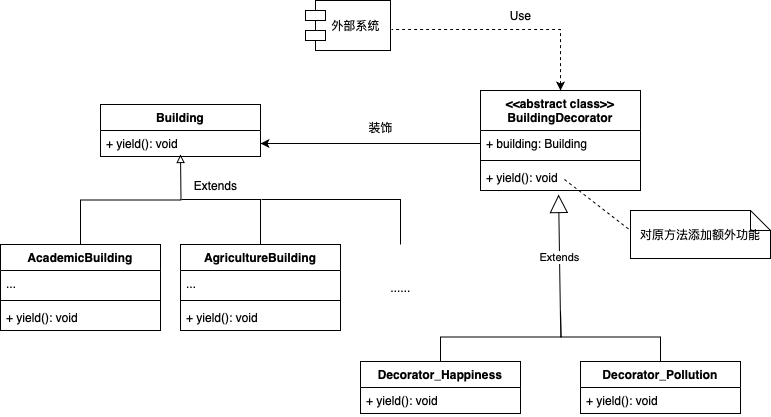
以下是装饰器模式的主要参与者：

1. **Component（组件）：** 定义了一个抽象接口，可以被具体组件和装饰器共享。这是装饰器模式的核心接口。
2. **ConcreteComponent（具体组件）：** 实现了Component接口的具体组件，是被装饰的原始对象。
3. **Decorator（装饰器）：** 实现了Component接口，并包含一个Component实例。这个抽象类的子类可以添加额外的行为，以动态地扩展原始对象的功能。
4. **ConcreteDecorator（具体装饰器）：** 是Decorator的具体子类，实现了具体的装饰逻辑。可以有多个具体装饰器，它们可以按需组合使用。

装饰器模式的工作流程如下：

1. 客户端创建一个具体组件的实例，并可以根据需要添加一系列的装饰器。
2. 每个装饰器都包装了一个组件实例，并在调用原始组件的相应方法之前或之后执行额外的逻辑。
3. 装饰器的层级可以根据需要嵌套，以形成一个复杂的装饰链。
4. 最终，客户端使用这个装饰过的组件，而不需要了解底层的装饰器结构。

### 3.10.2 UML图



### 3.10.3 核心代码

游戏策划规定建筑的产出值需要根据单元格幸福指数、污染指数两个指标进⾏百分⽐变化。⽽逻辑上这两个值和某个建筑⽆关，但⼜切实影响了所有建筑的所有属性，相当于使得建筑对象的产出⾏为（yield()⽅法）具有额外⾏为，因此使⽤装饰器模式将该设定与具体建筑类解耦，创建建筑装饰器类管理这些逻辑。装饰器模式需要装饰器继承和⽬标⼀样的基类，这样套过⼏层装饰器后的对象依然可以当成基类对象使⽤。

装饰器基类：

 public abstract class BuildingDecorator : Building  
 {  
     protected Building building;  
     public BuildingDecorator(Building building)  
    {  
         this.building = building;  
    }  
 ​  
     public override void yield(bool recaculate = true)  
    {  
         building.yield();  
    }  
 }

 // 装饰器子类  
 public class Decorator\_Pollution : BuildingDecorator  
 {  
     public Decorator\_Pollution(Building building) : base(building) { }  
 ​  
     public override void yield(bool recaculate = true)  
    {  
         float lambda = 1;  
         if (cell.pollution >= 300)  
        {  
             lambda = 0.2f;  
        }  
         else if (cell.pollution >= 200)  
        {  
             lambda = 0.5f;  
        }  
         else if (cell.pollution >= 100)  
        {  
             lambda = 0.8f;  
        }  
         building.updateScale(lambda);  
 ​  
         building.yield();  
    }  
 }

 // 装饰器子类  
 public class Decorator\_Happiness : BuildingDecorator  
 {  
     public Decorator\_Happiness(Building building) : base(building) { }  
 ​  
     public override void yield(bool recaculate = true)  
    {  
         float lambda = 1;  
         if (cell.happiness >= 200)  
        {  
             lambda = 1.5f;  
        }  
         else if (cell.happiness >= 100)  
        {  
             lambda = 1.2f;  
        }  
         else if (cell.happiness <= -100)  
        {  
             lambda = 0.5f;  
        }  
         else if (cell.happiness < 0)  
        {  
             lambda = 0.8f;  
        }  
         building.updateScale(lambda);  
           
         building.yield();  
    }  
 }

 //装饰器修饰  
 new Decorator\_Pollution(new Decorator\_Happiness(academicBuilding));

### 3.10.4 优缺点分析

**优点：**

1. **灵活性和扩展性：** 装饰器模式允许动态地添加或移除功能，而无需修改现有代码。这提供了更大的灵活性，使得系统更容易扩展。
2. **遵循开闭原则：** 装饰器模式遵循开闭原则，允许在不修改组件的情况下扩展其功能。这意味着可以方便地引入新的装饰器类，而不影响现有代码。
3. **可组合性：** 装饰器可以按需组合，创建不同的组合以满足特定需求。这种组合性使得构建具有复杂功能的对象变得相对简单。
4. **单一职责原则：** 装饰器模式遵循单一职责原则，每个装饰器只关注一个特定的功能，使得代码更清晰、易于理解和维护。

**缺点：**

1. **复杂性增加：** 随着装饰器的增加，对象的构建和理解可能变得更加复杂。特别是当有多个装饰器嵌套使用时，代码可读性可能降低。
2. **潜在的性能开销：** 每个装饰器都会引入一些额外的开销，可能影响程序的性能。特别是在装饰链较长时，可能会产生一些性能成本。
3. **可能引入冗余代码：** 如果使用不当，可能会引入一些冗余的装饰器，使得代码变得复杂而难以维护。
4. **不适用于所有情况：** 装饰器模式并不适用于所有的场景。在一些情况下，更简单的模式或者继承关系可能更加合适。

## 3.11 外观模式 Facade Pattern

### 3.11.1 实现描述

外观模式（Facade Pattern）是一种结构型设计模式，旨在提供一个简化的接口，隐藏系统的复杂性，并为客户端提供一个更方便的入口点。外观模式通过创建一个包装类，该类包含系统中一组相互关联的类的接口，从而使客户端能够更轻松地与系统进行交互。

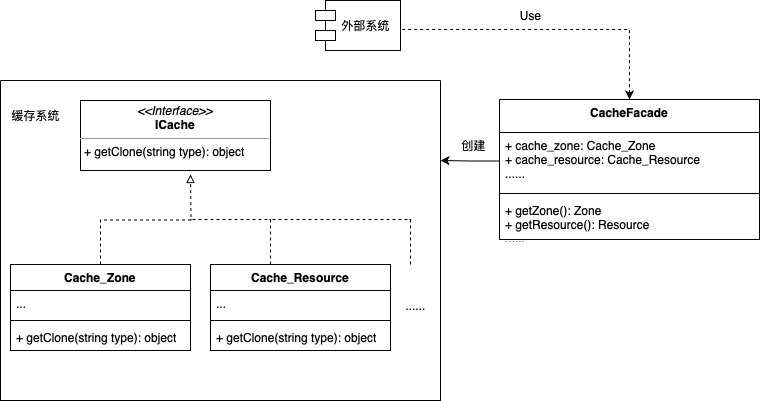
以下是外观模式的主要参与者：

1. **Facade（外观）：** 提供了一个简化而统一的接口，封装了系统中一组复杂的子系统的功能。外观模式的核心就是这个类。
2. **Subsystem（子系统）：** 包含了系统中的一组相关的类，每个子系统负责处理系统的一部分功能。这些子系统可以直接被客户端使用，但外观模式通过外观类将它们封装起来，使得客户端不需要了解系统的内部结构。

外观模式的工作流程如下：

1. 客户端通过与外观类交互来完成某项功能，而不需要直接与系统的多个子系统交互。
2. 外观类内部封装了对多个子系统的调用，根据客户端的需求，协调这些子系统的工作。
3. 外观类向客户端提供了一个简化的接口，隐藏了系统的复杂性，使客户端更容易使用系统。

### 3.11.2 UML图



### 3.11.3 核心代码

这个项⽬设计了缓存系统，有⼀堆缓存类来负责可克隆对象的快速获取。为了便于管理这个系统，向外提供⼀系列易⽤的接⼝，使⽤外观模式封装所有Cache类。这⾥如果没有外观，客户端需要知道具体的对应Cache类，调⽤其getClone()⽅法后还需要强制类型转换；⽽使⽤外观类后，客户端只需要⼀个CacheFacade对象，调⽤其clone()⽅法即可，既降低了耦合度，⼜增强了代码可读性，⽽且⼦系统可以独⽴于客户端变化（增加新Cache类），扩展性好。

代码如下：

 public class CacheFacade  
 {  
     private static CacheFacade \_instance = new CacheFacade();  
     private CacheFacade() { }  
     public static CacheFacade getInstance()  
    {  
         return \_instance;  
    }  
 ​  
     ICache cache\_zone = Cache\_Zone.getInstance();  
     ICache cache\_feature = Cache\_Feature.getInstance();  
     ICache cache\_resource = Cache\_Resource.getInstance();  
     ICache cache\_terrain = Cache\_Terrain.getInstance();  
 ​  
     public Zone cloneZone(string zoneType)  
    {  
         return (Zone)cache\_zone.getClone(zoneType);  
    }  
 ​  
     public Feature cloneFeature(string featureType)  
    {  
         return (Feature)cache\_feature.getClone(featureType);  
    }  
 ​  
     public Resource cloneResource(string resourceType)  
    {  
         return (Resource)cache\_resource.getClone(resourceType);  
    }  
 ​  
     public Terrain cloneTerrain(string terrainType)  
    {  
         return (Terrain)cache\_terrain.getClone(terrainType);  
    }  
 }  
 ​

### 3.11.4 优缺点分析

**优点：**

1. **简化客户端使用：** 外观模式提供了一个简化的接口，使得客户端更容易使用系统，而不需要了解系统的复杂内部结构。这有助于降低使用成本和提高用户友好性。
2. **降低耦合度：** 外观模式将客户端与系统的具体实现解耦，客户端只需要通过外观类与系统交互。这降低了客户端和子系统之间的耦合度，使得系统的变化不会影响到客户端。
3. **提高灵活性：** 外观模式使得系统更容易扩展和维护。因为客户端只依赖于外观，而不依赖于系统的具体实现，所以可以更灵活地对系统进行修改和升级。
4. **隐藏实现细节：** 外观模式可以隐藏系统的实现细节，提高系统的安全性。客户端只能访问外观提供的接口，而无法直接访问系统的底层组件。
5. **提高可重用性：** 外观模式可以提高系统中各个子系统的可重用性。因为系统的实现细节被封装在外观类中，可以更容易地在其他项目中复用这些子系统。

**缺点：**

1. **不适用于所有情况：** 外观模式并不适用于所有的系统设计场景。在某些情况下，直接使用子系统的接口可能更加灵活。
2. **可能引入不必要的复杂性：** 如果系统本身比较简单，使用外观模式可能会引入不必要的复杂性，使得系统更难理解。
3. **不符合开闭原则：** 在一定程度上，外观模式可能违反了开闭原则。当需要添加新的功能时，可能需要修改外观类。
4. **可能影响性能：** 如果外观类的设计不合理，可能会引入性能问题。特别是在频繁调用外观接口的情况下，可能会增加额外的开销。

## 3.12 责任链模式 Responsibility Chain Pattern

### 3.12.1 实现描述

顾名思义，责任链模式（Chain of Responsibility Pattern）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。这种类型的设计模式属于行为型模式。

在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

**意图**

避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。

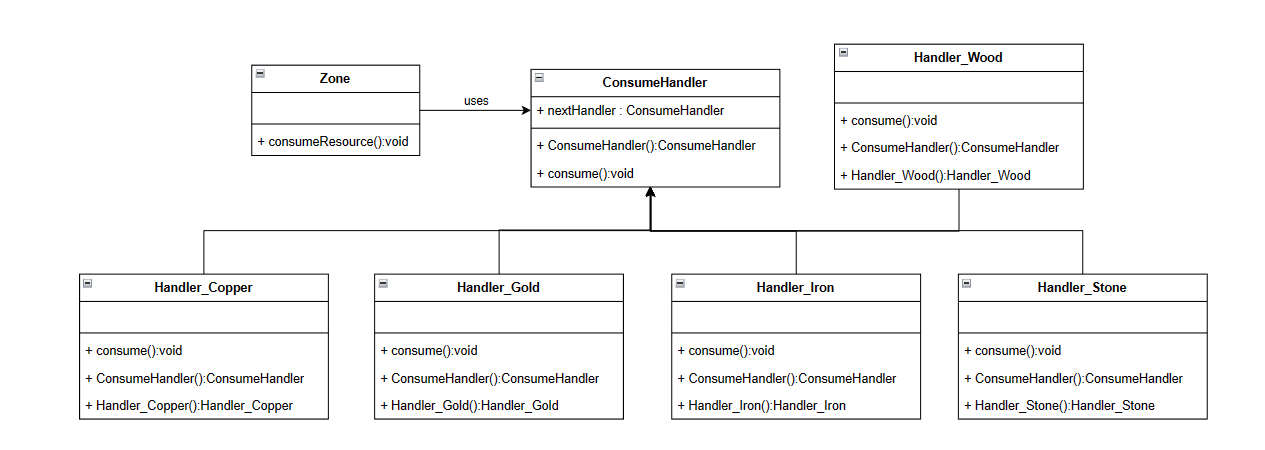
**主要解决**

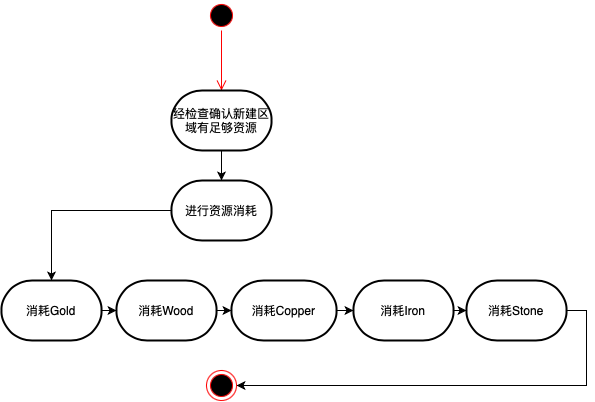
职责链上的处理者负责处理请求，客户只需要将请求发送到职责链上即可，无须关心请求的处理细节和请求的传递，所以职责链将请求的发送者和请求的处理者解耦了。

**应用场景**

1. 有多个对象可以处理同一个请求，具体哪个对象处理该请求由运行时刻自动确定。
2. 在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
3. 可动态指定一组对象处理请求。

### 3.12.2 UML图





### 3.12.3 核心代码

Handler\_\*类都继承ConsumeHandler抽象类，具有下⼀个Handler的指针、构造函数接收该指针并初始 化（即⼀个链表节点），⼦类要实现抽象⽅法consume(Zone zone)，每个⼦类负责⼀种区域消耗的资源，包括 gold, wood, iron, copper, stone。⼦类实现的consume()⽅法先做资源消耗的逻辑，然后将请求继续传递给下⼀ 个handler。

这样Zone类的consumeResource()⽅法逻辑和具体的消耗资源逻辑解耦，Zone类⾥只需要构造好 handler链，调⽤头节点的consume()⽅法即可完成整个消耗资源过程

创建抽象的记录器类。

 // 责任链模式：建造区域的开销有消耗金钱、木材、石材、铜、铁等步骤，建模成责任链  
 public abstract class ConsumeHandler  
 {  
     protected ConsumeHandler nextHandler = null;  
     public ConsumeHandler(ConsumeHandler handler)  
    {  
         nextHandler = handler;  
    }  
 ​  
     public abstract void consume(Zone zone);  
 }

创建扩展了该记录器类的实体类。

 public class Handler\_Copper : ConsumeHandler  
 {  
     public Handler\_Copper(ConsumeHandler handler) : base(handler) { }  
     public override void consume(Zone zone)  
    {  
         AssetManager.getInstance().Copper\_Stock -= zone.cost\_copper;  
         if (nextHandler != null)  
             nextHandler.consume(zone);  
    }  
 }

 public class Handler\_Gold : ConsumeHandler  
 {  
     public Handler\_Gold(ConsumeHandler handler) : base(handler) { }  
     public override void consume(Zone zone)  
    {  
         AssetManager.getInstance().Gold\_Stock -= zone.cost\_gold;  
         if (nextHandler != null)  
             nextHandler.consume(zone);  
    }  
 }

 public class Handler\_Iron : ConsumeHandler  
 {  
     public Handler\_Iron(ConsumeHandler handler) : base(handler) { }  
     public override void consume(Zone zone)  
    {  
         AssetManager.getInstance().Iron\_Stock -= zone.cost\_iron;  
         if (nextHandler != null)  
             nextHandler.consume(zone);  
    }  
 }

 public class Handler\_Stone : ConsumeHandler  
 {  
     public Handler\_Stone(ConsumeHandler handler) : base(handler) { }  
     public override void consume(Zone zone)  
    {  
         AssetManager.getInstance().Stone\_Stock -= zone.cost\_stone;  
         if (nextHandler != null)  
             nextHandler.consume(zone);  
    }  
 }

 public class Handler\_Wood : ConsumeHandler  
 {  
     public Handler\_Wood(ConsumeHandler handler) : base(handler) { }  
     public override void consume(Zone zone)  
    {  
         AssetManager.getInstance().Wood\_Stock -= zone.cost\_wood;  
         if (nextHandler != null)  
             nextHandler.consume(zone);  
    }  
 }

创建不同类型的记录器。赋予它们不同的错误级别，并在每个记录器中设置下一个记录器。每个记录器中的下一个记录器代表的是链的一部分。

 public abstract class Zone : ICloneable  
 {  
 ​  
    ...  
 ​  
     // 责任链模式：消耗资源  
     public void consumeResource()  
    {  
         getHandlerChain().consume(this);  
    }  
 ​  
     public ConsumeHandler getHandlerChain()  
    {  
         ConsumeHandler handler\_iron = new Handler\_Iron(null);  
         ConsumeHandler handler\_copper = new Handler\_Copper(handler\_iron);  
         ConsumeHandler handler\_stone = new Handler\_Stone(handler\_copper);  
         ConsumeHandler handler\_wood = new Handler\_Wood(handler\_stone);  
         ConsumeHandler handler\_gold = new Handler\_Gold(handler\_wood);  
         return handler\_gold;  
    }  
 }  
 ​

### 3.12.4 优缺点分析

**优点**

1. **降低耦合度**。它将请求的发送者和接收者解耦。
2. **简化了对象**。使得对象不需要知道链的结构。
3. **增强给对象指派职责的灵活性**。通过改变链内的成员或者调动它们的次序，允许动态地新增或者删除责任。
4. 增加新的请求处理类很方便。

**缺点**

1. 不能保证请求一定被接收。
2. 系统性能将受到一定影响，而且在进行代码调试时不太方便，可能会造成循环调用。
3. 可能不容易观察运行时的特征，有碍于除错。

## 3.13 迭代器模式 Iterator Pattern

### 3.13.1 实现描述

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。

**意图**

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。

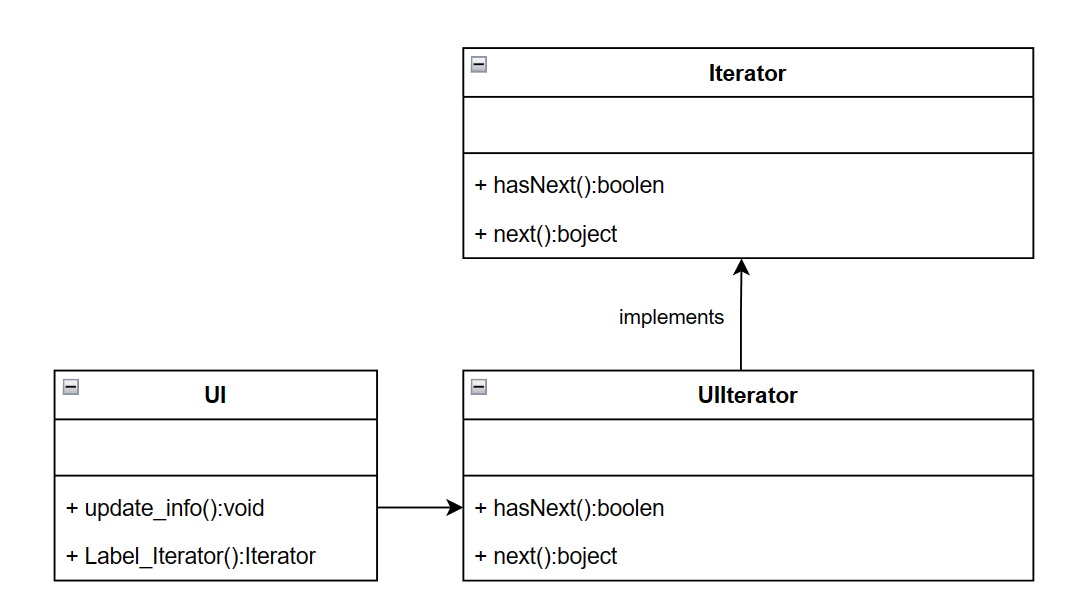
**主要解决**

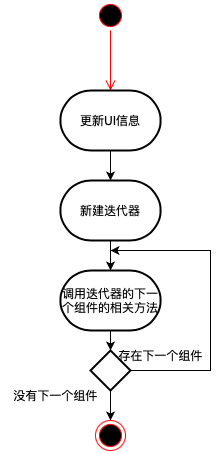
不同的方式来遍历整个整合对象。

**使用场景**

1. 访问一个聚合对象的内容而无须暴露它的内部表示。
2. 需要为聚合对象提供多种遍历方式。
3. 为遍历不同的聚合结构提供一个统一的接口。

### 3.13.2 UML图





### 3.13.3 核心代码

正如访问者模式中所说，根节点UI需要在接受访问⽅法update\_info()中遍历其Label类型组件，递归调⽤其这个⽅法。

⽽UI根节点并没有结构化存储这些组件，遍历起来不⽅便，且如果UI组件的层次结构发⽣变化，还要修改UI类的代码。

于是使⽤迭代器模式帮助UI进⾏遍历：将要遍历的组件顺序写在next()⾥，UI就可以⼀个for循环完成遍历。

update\_info()

 // 访问者模式的根节点的accept()方法，遍历label组件调用其update\_info()方法  
     public override void update\_info(UI\_Updater updater)  
    {  
         for(Iterator iterator = new Label\_Iterator(this); iterator.hasNext(); )  
        {  
             UI\_Component label = (UI\_Component)iterator.next();  
             label.update\_info(updater);  
        }  
    }

迭代器类

 // 迭代器模式：迭代器，帮助UI根节点遍历label组件  
     private class Label\_Iterator : Iterator  
    {  
         int index = 0;  
         int count = 25; // 游戏UI中的label数  
         UI ui;  
         public Label\_Iterator(UI ui)  
        {  
             this.ui = ui;  
        }  
 ​  
         public bool hasNext()  
        {  
             return index < count;  
        }  
 ​  
         public object next()  
        {  
             index++;  
             switch (index)  
            {  
                 // asset  
                 case 1: return ui.label\_population\_stock;  
                 case 2: return ui.label\_population\_increment;  
                 case 3: return ui.label\_accommodation;  
                 case 4: return ui.label\_food\_stock;  
                 case 5: return ui.label\_food\_increment;  
                 case 6: return ui.label\_gold\_stock;  
                 case 7: return ui.label\_gold\_increment;  
                 case 8: return ui.label\_wood;  
                 case 9: return ui.label\_stone;  
                 case 10: return ui.label\_copper;  
                 case 11: return ui.label\_iron;  
 ​  
                 // statistic  
                 case 12: return ui.label\_urbanization;  
                 case 13: return ui.label\_science;  
                 case 14: return ui.label\_culture;  
                 case 15: return ui.label\_commercialization;  
                 case 16: return ui.label\_productivity;  
                 case 17: return ui.label\_electricity;  
                 case 18: return ui.label\_military\_power;  
                 case 19: return ui.label\_happiness;  
                 case 20: return ui.label\_pollutation;  
 ​  
                 // control  
                 case 21: return ui.label\_cellinfo;  
                 case 22: return ui.label\_featureinfo;  
                 case 23: return ui.label\_resourceinfo;  
                 case 24: return ui.label\_zoneinfo;  
                 case 25: return ui.label\_buildinginfo;  
                 default: return null;  
            }  
        }  
 ​  
         public void reset()  
        {  
             index = 0;  
        }  
    }

创建实现了 update\_info 的实体类UI。该类有实现了 Iterator 的内部类 Label\_Iterator。

 public class UI : UI\_Component  
 {  
       
     // Asset  
     public Panel\_Asset panel\_asset;  
     public Label\_Population\_Stock label\_population\_stock;  
     public Label\_Population\_Increment label\_population\_increment;  
     public Label\_Accommodation label\_accommodation;  
     public Label\_Food\_Stock label\_food\_stock;  
     public Label\_Food\_Increment label\_food\_increment;  
     public Label\_Gold\_Stock label\_gold\_stock;  
     public Label\_Gold\_Increment label\_gold\_increment;  
     public Label\_Wood label\_wood;  
     public Label\_Stone label\_stone;  
     public Label\_Copper label\_copper;  
     public Label\_Iron label\_iron;  
 ​  
     // Statistic  
     public Panel\_Statistic panel\_statistic;  
     public Label\_Urbanization label\_urbanization;  
     public Label\_Science label\_science;  
     public Label\_Culture label\_culture;  
     public Label\_Commercialization label\_commercialization;  
     public Label\_Productivity label\_productivity;  
     public Label\_Electricity label\_electricity;  
     public Label\_Military\_Power label\_military\_power;  
     public Label\_Happiness label\_happiness;  
     public Label\_Pollutation label\_pollutation;  
 ​  
     // Minimap  
     public Minimap minimap;  
 ​  
     // Settings  
     public Button\_Settings button\_settings;  
 ​  
     // Control  
     public Panel\_Control panel\_control;  
     public Label\_CellInfo label\_cellinfo;  
     public Label\_FeatureInfo label\_featureinfo;  
     public Button\_RemoveFeature button\_removefeature;  
     public Label\_ResourceInfo label\_resourceinfo;  
     public Button\_HarvestResource button\_harvestresource;  
     public Label\_ZoneInfo label\_zoneinfo;  
     public Button\_ConstructZone button\_constructzone;  
     public Label\_BuildingInfo label\_buildinginfo;  
     public Button\_ConstructBuilding button\_constructbuilding; // TODO:有多种建筑，按钮个数需要根据区域类型生成  
     public Button\_Demolish button\_demolish;  
     public Button\_Afforest button\_afforest;  
 ​  
     // 迭代器模式：迭代器，帮助UI根节点遍历label组件  
     private class Label\_Iterator : Iterator  
    {  
        ...  
    }  
 ​  
     // 访问者模式的根节点的accept()方法，遍历label组件调用其update\_info()方法  
     public override void update\_info(UI\_Updater updater)  
    {  
        ...  
    }  
 }  
 ​

### 3.13.4 优缺点分析

**优点**

1. 它支持以不同的方式遍历一个聚合对象。
2. 迭代器简化了聚合类。。
3. 在同一个聚合上可以有多个遍历。
4. 在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。

**缺点**

1. 于迭代器模式将存储数据和遍历数据的职责分离，增加新的聚合类需要对应增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性。

## 3.14 中介者模式 Mediator Pattern

### 3.14.1 实现描述

中介者模式（Mediator Pattern）是用来降低多个对象和类之间的通信复杂性。这种模式提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护。中介者模式属于行为型模式。

**意图**

用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

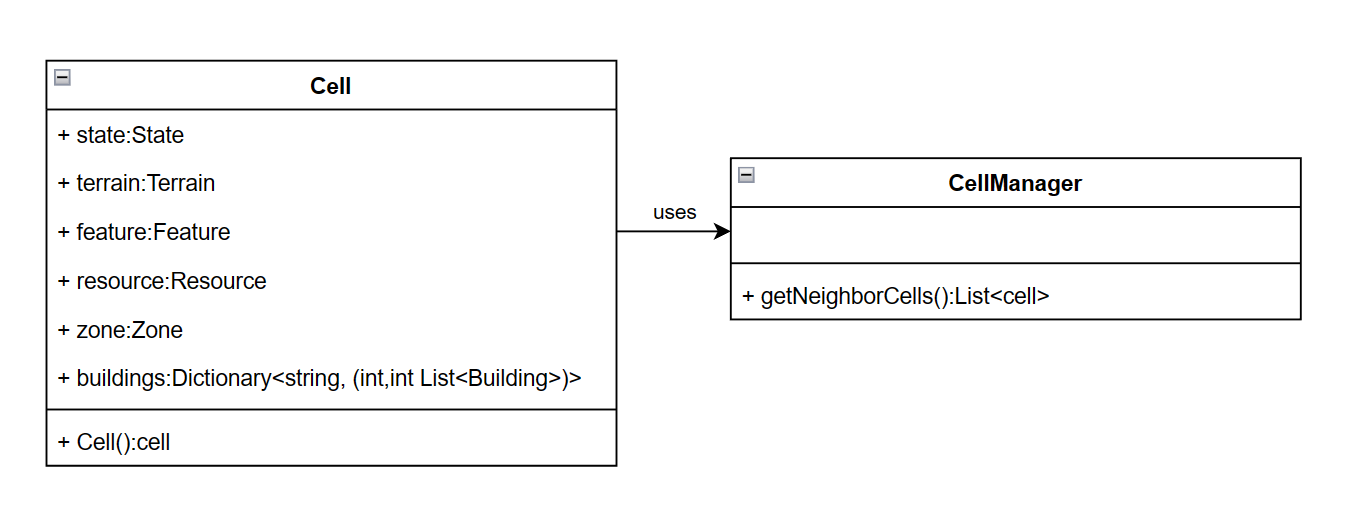
**主要解决**

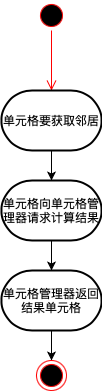
对象与对象之间存在大量的关联关系，这样势必会导致系统的结构变得很复杂，同时若一个对象发生改变，我们也需要跟踪与之相关联的对象，同时做出相应的处理。

**使用场景**

1. 系统中对象之间存在比较复杂的引用关系，导致它们之间的依赖关系结构混乱而且难以复用该对象。
2. 想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。

### 3.14.2 UML图





### 3.14.3 核心代码

中介者模式将类之间的交互委托给中介者集中式处理，从⽽将对象（节点）和交互⾏为（边）构成的图的复杂 度极⼤降低。该模式减少类之间的直接通信，降低其在业务逻辑⽅⾯的耦合度，且抽象出来的中介者通常具有现实 意义，如UI主⾯板等，使得代码容易理解、可扩展性好。

这个项⽬中，单元格之间具有⼀些交互作⽤，如获得与某 个单元格相距x以内的单元格数组，如果让单元格之间直接通信，每个单元格只有⾃身坐标，需要访问 CellManager才能获取其他邻居，且这些逻辑和单元格⾃身⽆关（违反高内聚），因此把这个需求委托给中介者 CellManager，可以⽅便地集中实现获取邻居的逻辑。

创建 Cell 类。

 public class Cell : MonoBehaviour  
 {  
     // 坐标 [0, x\_max], [0, y\_max]  
     public int x;  
     public int y;  
 ​  
     // 持有对象  
     public State state;  
     public Terrain terrain;  
     public Feature feature;  
     public Resource resource;  
     public Zone zone = null;  
     public Dictionary<string, (int, int, List<Building>)> buildings; // 存建筑  
 ​  
    ...  
 ​  
     // 默认构造函数，用于Unity序列化  
     public Cell()  
    {  
 ​  
    }  
 ​  
    ...  
 ​  
 }

创建中介类。

 public class CellManager : MonoBehaviour  
 {  
 ​  
    ...  
 ​  
     // 中介者模式：将单元格之间的距离计算委托给CellManager  
     // CellManager作为Cell的Mediator  
     public List<Cell> getNeighborCells(Cell cell, int distance)  
    {  
         List<Cell> neighborCells = new List<Cell>();  
         int x = cell.x;  
         int y = cell.y;  
         for (int i = x - distance; i <= x + distance; i++)  
        {  
             for (int j = y - distance; j <= y + distance; j++)  
            {  
                 if (i >= 0 && i < cells.GetLength(0) && j >= 0 && j < cells.GetLength(1))  
                {  
                     neighborCells.Add(cells[i, j]);  
                }  
            }  
        }  
         return neighborCells;  
    }  
 }  
 ​

### 3.14.4 优缺点分析

**优点**

1. 降低了类的复杂度，将一对多转化成了一对一。
2. 各个类之间的解耦。
3. 符合迪米特原则。

**缺点**

1. 中介者会庞大，变得复杂难以维护。

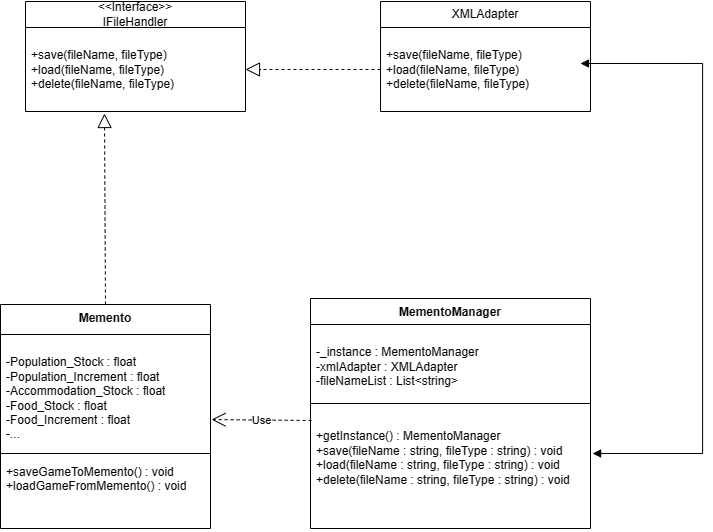
## 3.15 备忘录模式 Memento Pattern

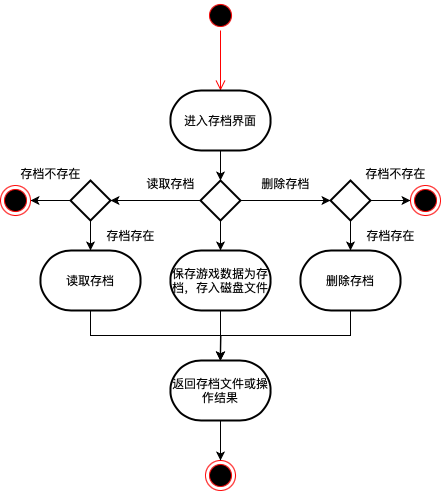
### 3.15.1 实现描述

备忘录模式主要用于捕获并保存对象的当前状态，以便在以后的时间点上可以将对象恢复到此状态。在游戏存档功能中，备忘录模式被用于实现游戏的保存和加载机制。

在城市规划类游戏中，玩家的每个决策都会影响游戏的状态，包括城市的布局、资源分配、人口管理等。备忘录模式允许在任何给定时刻捕获游戏的完整状态，并在需要时恢复。这对于提供存档和读档功能至关重要。

### 3.15.2 UML图





### 3.15.3 核心代码

Memento类负责封装游戏状态，saveGameToMemento和loadGameFromMemento方法分别用于捕获和恢复这些状态。

 public void saveGameToMemento()  
 {  
     Population\_Stock = AssetManager.getInstance().Population\_Stock;  
 ​  
 ​  
 }  
 ​  
 public void loadGameFromMemento()  
 {  
     
     AssetManager.getInstance().Population\_Stock = Population\_Stock;  
 ​  
 }  
 ​  
 ​

MementoManager通过单例模式管理这些存档，并提供了灵活的文件格式支持。

 private static MementoManager \_instance = new MementoManager();  
 public static MementoManager getInstance()  
 {  
     return \_instance;  
 }  
 ​  
 public void save(string fileName, string fileType)  
 {  
     // 存档文件的保存逻辑  
     Memento memento = new Memento();  
     memento.saveGameToMemento();  
     string json = JsonUtility.ToJson(memento);  
     System.IO.File.WriteAllText(fileName + ".json", json);  
 }  
 ​  
 public void load(string fileName, string fileType)  
 {  
     // 存档文件的加载逻辑  
     string json = System.IO.File.ReadAllText(fileName + ".json");  
     Memento memento = JsonUtility.FromJson<Memento>(json);  
     memento.loadGameFromMemento();  
 }

### 3.15.4 优缺点分析

**优点：**

1. 状态封装与管理:备忘录模式通过Memento类有效地封装了游戏状态，使得状态管理更加清晰和模块化。
2. 撤销/重做的实现:该模式可用于实现撤销和重做操作，这在城市规划游戏中尤其有用，玩家可以撤销不理想的操作或重新执行它们。
3. 游戏存档的灵活性:允许玩家在任何时候保存和加载游戏状态，提高了游戏的可玩性和用户体验。

**缺点：**

1. 性能考虑:如果游戏状态非常复杂，频繁地保存和加载大量数据可能会影响性能，尤其是在低端设备上。
2. 存档数据的安全性:存档文件可能容易受到篡改或损坏，需要考虑加密和数据完整性校验。

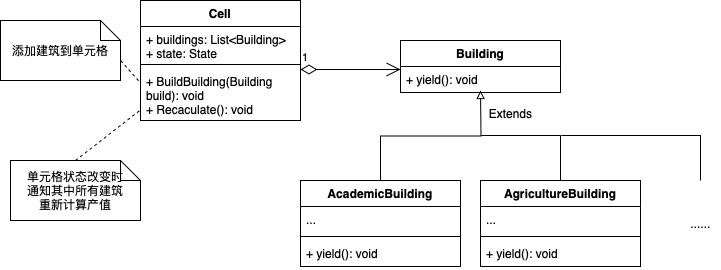
## 3.16 观察者模式 Observer Pattern

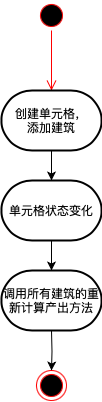
### 3.16.1 实现描述

观察者模式中，一个对象（称为观察者）订阅另一个对象（称为被观察者），以便在被观察者状态发生变化时得到通知。这种模式通常用于实现事件处理系统。

在该游戏项目中，观察者模式允许建筑（观察者）监听其所在单元格（被观察者）的状态变化。当单元格的属性如电力、幸福指数、污染指数等发生变化时，建筑的产出会根据这些变化动态调整。

### 3.16.2 UML图





### 3.16.3 核心代码

当单元格属性发⽣变化时（被建筑的yield产出触发）⾃动执⾏Recalculate()⽅法，表示被观察 者状态改变了，通知所有建筑对象并调⽤其yield()（要禁⽌⽆限递归）来重新计算产出值。

Recalculate方法负责重置单元格的属性并重新计算它们。这通常是在单元格的状态变化后触发的，如当一个建筑的产出发生变化时。

 ​  
     // 观察者模式：当单元格（被观察者）某些属性发生变化、可能产生额外增减益时，  
     // 调用此方法让其中所有建筑（观察者）重新计算属性  
     public void Recaculate()  
    {  
         if(state is not State\_Developed)  
        {  
             Debug.LogError("单元格未开发，无法重新计算属性");  
             return;  
        }  
 ​  
         ResetProperties();  
 ​  
         foreach(var buildingList in buildings.Values)  
        {  
             foreach(var building in buildingList.Item3)  
            {  
                 building.yield(false); // 防止无限递归  
            }  
        }  
    }  
 ​  
 ​

此方法是Building类的核心功能，它负责计算建筑产出并更新所在单元格的属性。参数recalculate用于控制是否应该重新计算单元格的属性。

  public virtual void yield(bool recaculate = true) // 将产出计算到单元格和城市的Asset和Statistics里  
    {  
         cell.accommodation += yield\_accommodation;  
         cell.food += yield\_food;  
         cell.gold += yield\_gold\_incre;  
         cell.science += yield\_science;  
         cell.culture += yield\_culture;  
         cell.commercialization += yield\_commercialization;  
         cell.productivity += yield\_productivity;  
         cell.military += yield\_military;  
 ​  
         if (recaculate)  
             cell.Recaculate();  
 ​  
         foreach (Cell c in cell.getNeighborCells(yield\_electricity.Item1))  
        {  
             c.electricity += yield\_electricity.Item2;  
             if (recaculate)  
                 c.Recaculate();  
        }  
         foreach (Cell c in cell.getNeighborCells(yield\_happiness.Item1))  
        {  
             c.happiness += yield\_happiness.Item2;  
             if (recaculate)  
                 c.Recaculate();  
        }  
         foreach (Cell c in cell.getNeighborCells(yield\_pollution.Item1))  
        {  
             c.pollution += yield\_pollution.Item2;  
             if (recaculate)  
                 c.Recaculate();  
        }  
    }  
 ​

### 3.16.4 优缺点分析

**优点：**

1. **解耦**:观察者模式允许Cell和Building类之间解耦。Cell不需要知道具体的Building实现细节，只需要调用yield()方法。
2. **动态订阅**:建筑可以动态地添加到单元格中，且每个建筑自动成为观察者。这为动态环境变化提供了很好的支持。
3. **集中管理**:单元格的状态变化可以集中管理，并且容易触发所有相关建筑的产出重新计算。
4. **即时响应**:当单元格属性发生变化时，相关的建筑可以即时响应并更新它们的产出，这对于游戏的实时性和互动性至关重要。

**缺点：**

1. **复杂性管理**:需要维护Building和Cell之间的关系，特别是在有大量建筑和单元格时，这可能会导致代码的复杂性增加。
2. **性能考虑**:每次单元格属性发生变化都会触发所有建筑的yield()方法，这可能会在大规模更新时影响性能。

## 3.17 状态模式 State Pattern

### 3.17.1 实现描述

状态模式（State Pattern）是一种常用的软件设计模式，它主要用于解决对象在不同状态下的行为变化问题。在状态模式中，一个对象的行为取决于其内部状态，且该对象可以在运行时改变其状态，从而改变其行为。状态模式的核心思想是将不同的状态封装成独立的类，并且使这些状态类具有一致的接口，以便在不同状态之间切换时能够无缝地进行转换。同时，状态模式还通过将对象的行为委托给当前状态对象来实现行为的动态变化。

**模式的结构**

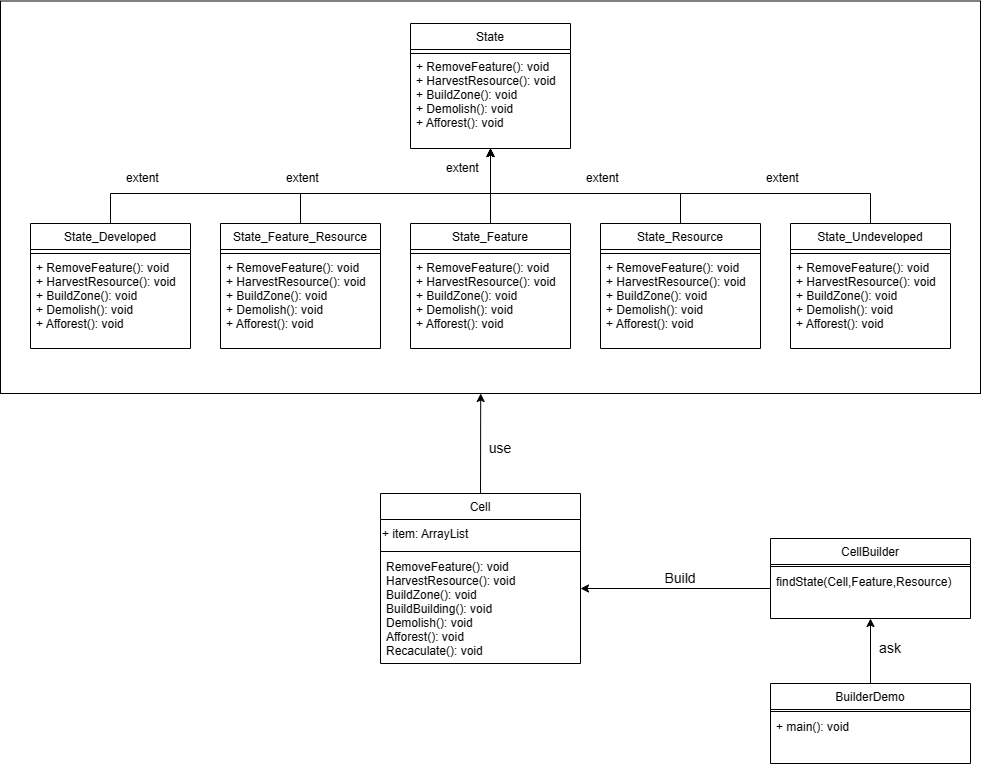
状态模式包含以下主要角色。

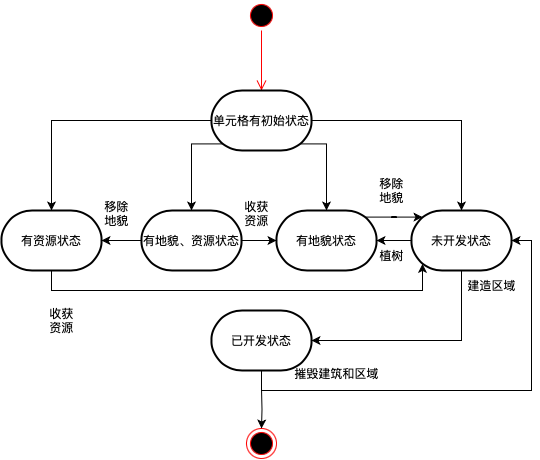
1. **环境类（Context）角色**：也称为上下文，它定义了客户端需要的接口，内部维护一个当前状态，并负责具体状态的切换。
2. **抽象状态（State）角色**：定义一个接口，用以封装环境对象中的特定状态所对应的行为，可以有一个或多个行为。
3. **具体状态（Concrete State）角色**：实现抽象状态所对应的行为，并且在需要的情况下进行状态切换。

**应用场景**

1. 对象的行为随状态变化而改变：当一个对象的行为在不同状态下有所不同，并且需要在运行时动态改变对象的行为时，可以使用状态模式。
2. 复杂的条件判断语句：当对象的状态引发复杂的条件判断语句时，可以使用状态模式来简化代码结构，提高可读性和可维护性。
3. 有限状态机（Finite State Machine）：状态模式常用于实现有限状态机，在有限状态机中，对象的行为取决于其当前状态，通过状态模式可以更好地管理状态之间的转换和行为的变化。

### 3.17.2 UML图





### 3.17.3 核心代码

根据策划，游戏中单元格具有多种状态且其⾏为取决于状态，这些状态之间通过⼀些操作相互转化，有些之间不能转换。因此使⽤状态模式，将各种状态封装为类，将状态之间的转换关系写成⽅法（也就对应了单元格⽀持的⼏种操作）。当单元格执⾏这些操作，其状态也需要改变，此时只需要调⽤state的相应操作⽅法即可，如Cell的Afforest⾥只需要state.Afforest()，不需要if或switch语句即可⽅便地实现状态转换。

在我们的系统中，共有“无地貌、无资源，但尚未开发或被废弃”、“有资源”、“有地貌”、“有地貌有资源”、“有区域”这五种类别的状态（五种状态接口分别参见State/State\_Undeveloped, State/State\_Resource, State/State\_Feature, State/State\_Feature\_Resource, State/State\_Developed）。

构造时对状态进行初始化

 public Cell(State state, Terrain terrain, Feature feature, Resource resource)  
 {  
     this.state = state;  
     this.terrain = terrain;  
     this.feature = feature;  
     this.resource = resource;  
 }

在建造者模式中将构造对象的逻辑委托给CellBuilder

 void Awake()   
 {  
     Cell cell = new CellBuilder().build(gameObject);  
          
     terrain = cell.terrain;  
     feature = cell.feature;  
     resource = cell.resource;  
     state = cell.state;  
     state.cell = this;  // 改变原cell对象中state的引用  
 }

在执行一些操作时对状态进行更改

 public void RemoveFeature()  
 {  
     state.RemoveFeature();  
     feature = null;  
 }  
 ​  
 public void HarvestResource()  
 {  
     state.HarvestResource();  
     resource = null;  
 }  
 ​  
 public void BuildZone(Zone zone)  
 {  
     state.BuildZone();  
     this.zone = zone;  
 ​  
     // 根据区域类型初始化建筑字典  
     buildings = zone.getBuildingDictionary();  
 }  
 public void Demolish()  
 {  
     state.Demolish();  
     zone = null;  
     buildings.Clear();  
 ​  
     ResetProperties();  
 }  
 ​  
 public void Afforest()  
 {  
     state.Afforest();  
     feature = new FeatureFactory().getFeature("Woods");  
 }

### 3.17.4 优缺点分析

**优点**

1. **结构清晰**：状态模式将与特定状态相关的行为局部化到一个状态中，并且将不同状态的行为分割开来，满足“单一职责原则”。
2. **减少对象间的相互依赖**：将状态转换显示化，减少对象间的相互依赖。将不同的状态引入独立的对象中会使得状态转换变得更加明确，且减少对象间的相互依赖。
3. **拓展性好**：状态类职责明确，有利于程序的扩展。通过定义新的子类很容易地增加新的状态和转换。

**缺点**

1. **增加了类的数量**：状态模式会引入多个具体状态类，每个状态都需要定义一个具体类，这样会增加类的数量，可能会使代码变得复杂。
2. **状态转换逻辑复杂**：如果状态之间的转换关系比较复杂，那么状态模式的实现可能会比较困难。在某些情况下，需要引入额外的上下文或状态管理器来处理状态之间的转换逻辑。
3. **上下文对象的状态依赖**：状态模式通常需要在上下文对象中维护一个对当前状态对象的引用。这意味着上下文对象的行为可能会受到状态对象的影响，导致上下文与状态之间存在一定的耦合性。
4. **状态类的复用性较差**：由于每个状态都需要定义一个具体类，因此随着状态的增加，状态类的数量也会增加。如果某些状态之间的行为相似，代码重复的情况可能会增加。
5. **不适用于频繁改变状态的场景**：状态模式适用于状态变化相对稳定且有限的场景。如果一个对象的状态变化频繁且无法确定，可能不适合使用状态模式，因为频繁切换状态会增加系统的复杂性。

## 3.18 策略模式 Strategy Pattern

### 3.18.1 实现描述

策略模式（Strategy Pattern）定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。

**模式的结构**

策略模式包含以下几个核心角色：

1. **环境（Context）**：维护一个对策略对象的引用，负责将客户端请求委派给具体的策略对象执行。环境类可以通过依赖注入、简单工厂等方式来获取具体策略对象。
2. **抽象策略（Abstract Strategy）**：定义了策略对象的公共接口或抽象类，规定了具体策略类必须实现的方法。
3. **具体策略（Concrete Strategy）**：实现了抽象策略定义的接口或抽象类，包含了具体的算法实现。

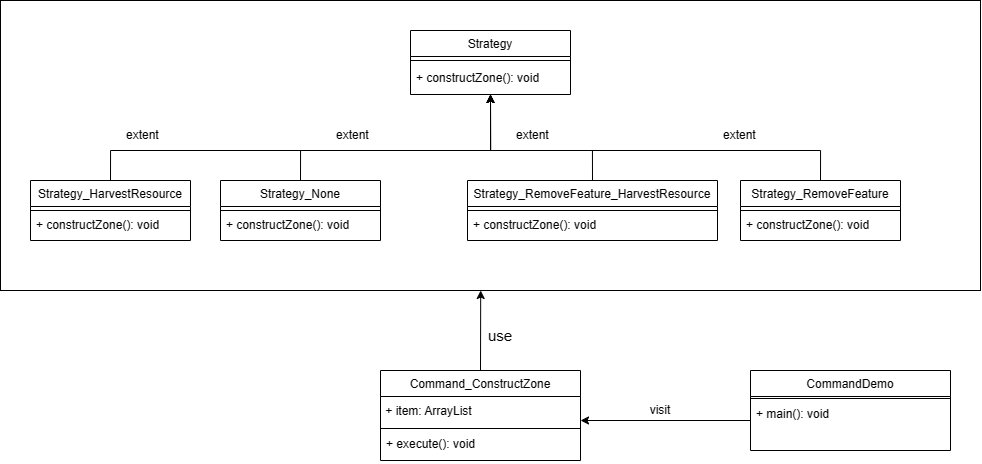
策略模式通过将算法与使用算法的代码解耦，提供了一种动态选择不同算法的方法。客户端代码不需要知道具体的算法细节，而是通过调用环境类来使用所选择的策略。

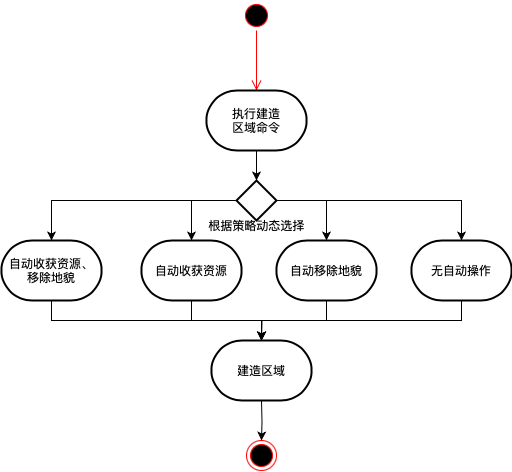
**应用场景**

策略模式在很多地方用到，如Java SE中的容器布局管理就是一个典型的实例，Java SE中的每个容器都存在多种布局供用户选择。在程序设计中，通常在以下几种情况中使用策略模式较多。

1. 一个系统需要动态地在几种算法中选择一种时，可将每个算法封装到策略类中。
2. 一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现，可将每个条件分支移入它们各自的策略类中以代替这些条件语句。
3. 系统中各算法彼此完全独立，且要求对客户隐藏具体算法的实现细节时。
4. 系统要求使用算法的客户不应该知道其操作的数据时，可使用策略模式来隐藏与算法相关的数据结构。
5. 多个类只区别在表现行为不同，可以使用策略模式，在运行时动态选择具体要执行的行为。

### 3.18.2 UML图





### 3.18.3 核心代码

在我们的系统中，有两种策略可供用户选择：自动移除地貌、自动收获资源。可以通过点击建造区域按钮，自动移除地貌和收获资源之后再进行建造区域。

通过策略模式，将各种策略封装为类、将其业务逻辑写在类的⽅法⾥，⽅便程序运⾏时改变策略。（四种策略接口参见Command/Strategy/Strategy\_HarvestResource, Command/Strategy/Strategy\_None, Command/Strategy/Strategy\_RemoveFeature\_HarvestResource, Command/Strategy/Strategy\_RemoveFeature）以下是对这四种策略的说明：

* Strategy\_None：两个都不勾选，不会⾃动移除地貌或收获资源，只能在单元格没有地貌、没有资源时建造区域
* Strategy\_RemoveFeature：只勾选移除地貌，建造区域时⾃动移除地貌，只能在单元格没有资源时建造区域
* Strategy\_HarvestResource：只勾选收获资源，建造区域时⾃动收获资源，只能在单元格没有地貌时建造区域
* Strategy\_RemoveFeature\_HarvestResource：⾃动移除地貌和收获资源，可以在除了⽔域外的任意单元格执⾏建造区域操作

 public class Command\_ConstructZone : Command  
 {  
     public Command\_ConstructZone(Cell cell, Zone zone, Strategy strategy)  
    {  
         this.cell = cell;  
         this.zone = zone;  
         this.strategy = strategy;  
    }  
     public override void execute()  
    {  
         strategy.constructZone(); // 策略：单元格可能自动执行移除地貌、收获资源等操作  
         base.execute();  
    }  
 }

### 3.18.4 优缺点分析

**优点**

1. **算法的独立性**：策略模式将不同的算法封装在独立的策略类中，使得每个算法都可以独立地进行修改、测试和维护，增强了代码的可读性和可维护性。
2. **扩展性好**：通过定义新的策略类，可以方便地增加新的算法，而无需修改原有代码。这样可以很容易地扩展系统的功能。
3. **减少条件判断**：策略模式可以避免使用大量的条件判断语句来选择不同的算法，提高了代码的简洁性和可读性。
4. **类之间的解耦**：策略模式将算法的具体实现封装在策略类中，客户端只需要关注选择合适的策略，而不需要了解其具体实现细节。这样可以降低类之间的耦合度，提高系统的灵活性和可维护性。

**缺点**

1. **增加了类的数量**：策略模式会引入多个策略类，每个策略类都要实现一种算法，这样会增加类的数量，可能会使代码变得复杂。
2. **客户端需要了解不同的策略**：客户端需要明确知道不同策略之间的区别和如何选择合适的策略，如果策略过多或者选择错误，可能会导致系统性能下降。
3. **上下文切换开销**：在使用策略模式时，需要在上下文对象中进行策略的切换。如果上下文对象频繁切换策略，可能会产生一定的性能开销。

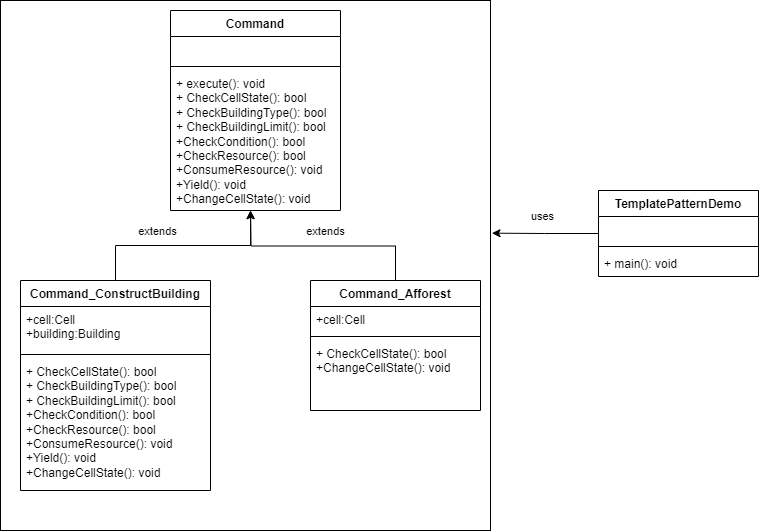
## 3.19 模板模式 Template Pattern

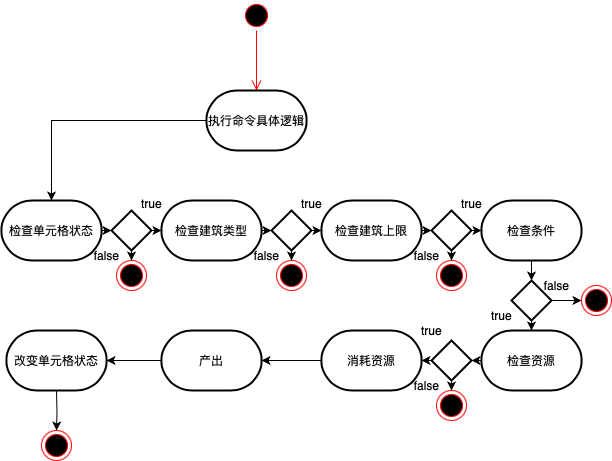
### 3.19.1 实现描述

模板模式是一种行为设计模式，它定义了一个算法的框架，将一些步骤推迟到子类。这个模式允许子类在不改变算法结构的情况下重新定义算法中的某些步骤。在模板模式中，定义一个抽象类，其中包含了算法的主要结构，以及一些由子类实现的抽象方法。这样，具体的子类可以提供特定的实现，但整体算法的流程保持不变。

这个项⽬中模板模式和命令模式紧密结合。每个具体命令的execute()都有⼀套相似的过程：对单元格、建筑等进⾏检查，不符合则返回；然后消耗资源、进⾏产出、更新单元格状态。通过使⽤模板模式，每个命令基本不需要重写execute()⽅法（ConstructZone特殊，因为要⽤策略模式，但也极为相似），只需要按照这套模板改写每个步骤的函数。这样减少了代码重复、⽅便可能的修改，且各种命令的总体执⾏过程和可变部分⼀⽬了然。

### 3.19.2 UML图





### 3.19.3 核心代码

基类Command中定义了一些通用的方法，然后子类重写用到的通用方法。例如：

 // 模板模式：对单元格的命令都使用类似的步骤模板  
 public class Command  
 {  
     public virtual bool CheckCellState()  
    {  
         return true;  
    }  
 ​  
     // ...  
 ​  
     public virtual void ChangeCellState()  
    {  
 ​  
    }  
 }

 // 命令：植树  
 public class Command\_Afforest : Command  
 {  
     // ...  
 ​  
     public override bool CheckCellState()  
    {  
         // ...  
         return true;  
    }  
     public override void ChangeCellState()  
    {  
         // ...  
    }  
 }

### 3.19.4 优缺点分析

**优点:**

1. **封装变化**：模板方法将不同的实现细节封装在不同的方法中，使得变化不会影响到算法的整体结构，提高了代码的可维护性。
2. **代码复用**：模板方法模式通过将算法的通用部分封装在模板中，可以在多个地方重复使用，避免了重复的代码。
3. **扩展性**：子类可以通过覆盖或扩展模板中的方法来定制算法的特定步骤，从而实现灵活的扩展。

**缺点：**

1. **不适用于每个场景**：并不是每个场景都适合使用模板方法模式，特别是当算法的不同部分之间关系较为复杂时，可能会导致设计变得过于繁琐。
2. **维护困难**：当模板方法的结构发生改变时，所有依赖于该模板的子类可能都需要相应的修改，维护成本可能会增加。

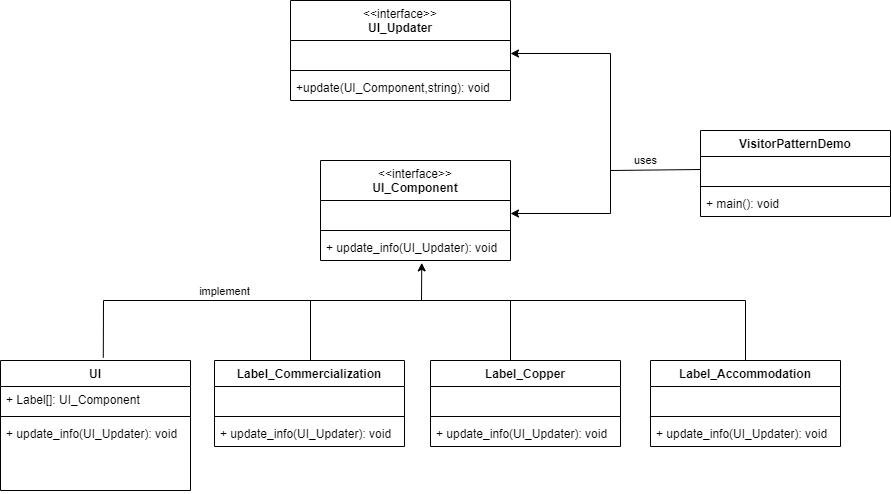
## 3.20 访问者模式 Visitor Pattern

### 3.20.1 实现描述

访问者模式是一种行为型设计模式，它允许定义一组操作，而无需改变被操作的对象的类。该模式通过在被访问的对象中引入一个访问者接口，使得在不同的具体访问者中可以定义不同的操作。这样，通过访问者模式，可以在不改变对象结构的前提下，添加新的操作或算法。

在本项目中，UI组件尤其是Label需要时常更新信息，但这块逻辑和游戏内部的统计信息、对象状态等数据有关，和UI本身⽆关，因此使⽤访问者模式解耦两者：UI\_Component的⼦类都是被访问者，实现update\_info()⽅法，接收访问者UI\_Updater的操作（具体组件是调⽤访问者的update操作，UI根节点组件特殊，需要结合迭代器模式遍历所有需要更新信息的组件，递归调⽤它们的update\_info()）；UI\_Updater类实现update()⽅法，根据访问的组件名进⾏相应操作。通过访问者模式，UI组件可以执⾏额外操作（时常更新信息），⽽不需要修改类本身，这些逻辑被委托给了访问者。

### 3.20.2 UML图





### 3.20.3 核心代码

定义一个表示元素的接口。

 // 访问者模式：UI\_Component都是被访问者，UI根节点和label组件重写update\_info()方法  
 ​  
 public class UI\_Component : MonoBehaviour     
 {  
     List<UI\_Component> children = new List<UI\_Component>();  
 ​  
     // ...  
 ​  
     // 访问者模式的accept()方法  
     public virtual void update\_info(UI\_Updater updater)  
    {  
 ​  
    }  
 }

创建扩展了上述类的实体类。

 // UI类：所有UI组件的根节点  
 public class UI : UI\_Component  
 {  
       
     // Asset  
     public Panel\_Asset panel\_asset;  
     public Label\_Population\_Stock label\_population\_stock;  
     // 别的组件声明不在展示  
 ​  
     // 迭代器模式：迭代器，帮助UI根节点遍历label组件  
     private class Label\_Iterator : Iterator  
    {  
         // ...  
    }  
 ​  
     // 访问者模式的根节点的accept()方法，遍历label组件调用其update\_info()方法  
     public override void update\_info(UI\_Updater updater)  
    {  
         for(Iterator iterator = new Label\_Iterator(this); iterator.hasNext(); )  
        {  
             UI\_Component label = (UI\_Component)iterator.next();  
             label.update\_info(updater);  
        }  
    }  
 }

 // 别的组件类不在展示  
 public class Label\_Commercialization : UI\_Component  
 {  
     public override void update\_info(UI\_Updater updater)  
    {  
         updater.update(this, "commercialization");  
    }  
 }  
 ​

实现访问者类。

 public class UI\_Updater  
 {  
     // 访问者模式：UI\_Component都是被访问者，UI根节点和label组件重写update\_info()方法  
     public void update(UI\_Component component, string info)  
    {  
         // ...  
    }  
 }

### 3.20.4 优缺点分析

**优点：**

1. **符合单一职责原则**：每个具体访问者类负责实现一个特定的操作，从而使得各个访问者类具有单一职责。
2. **解耦操作和元素类**：访问者模式将具体元素的结构和具体访问者的操作分离，使得可以在不修改元素类的情况下定义新的操作。这有助于保持元素类的稳定性。
3. **可扩展性**：当需要添加新的操作时，只需创建新的具体访问者类即可，而无需修改现有的元素类。这符合开闭原则，使得系统更容易扩展。

**缺点：**

1. **破坏封装**：访问者模式可能导致访问者访问元素的私有成员，破坏了元素的封装性。这使得访问者能够访问元素的内部细节，增加了代码的耦合性。
2. **元素变更困难**：当需要增加新的元素类时，除了需要创建新的具体元素类外，还需要修改所有的具体访问者类，以适应新的元素。这增加了系统的复杂性。

## 3.21 命令模式 Command Pattern

### 3.21.1 实现描述

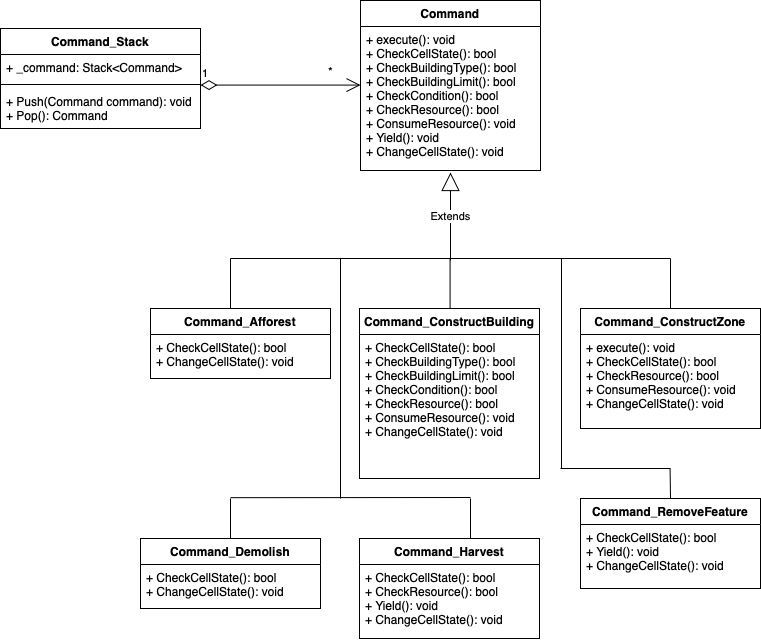
命令模式是一种行为型模式。请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。它可将请求转换为一个包含与请求相关的所有信息的独立对象。

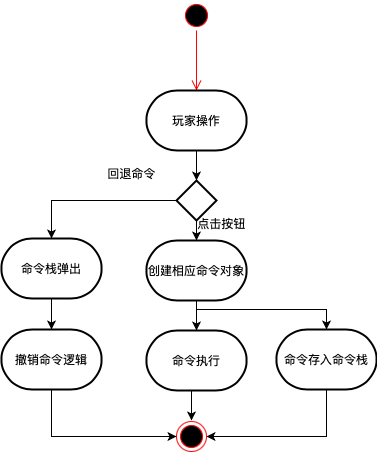
作为一个有GUI的游戏，必不可少的就是一堆按钮及其功能。为了遵从前后端分离的理念，必须使用命令模型，将操作的发起者（前端组件）和执行者（单元格、区域、建筑等被具体命令持有并使用的对象）解耦。对于一个用户的操作，程序执行流程如下：GUI的组件触发，与其对应的Command具体对象被创建；该命令对象执行execute()方法，其中是对单元格、区域、建筑等执行操作对象的方法调用，且命令存入命令栈；具体操作对象的方法被调用，分头完成整个操作的所有逻辑。

此外，命令模式还是为了操作的回退、重做等，就像word的ctrl+z、shift+ctrl+z等。因此通过命令模型将操作的执行抽象为一个个Command对象存入命令栈，通过出入栈操作实现回退和重做。

在本项目中，几种对单元格的主要操作封装成命令：移除地貌、收获资源、建造区域、废弃、种植树林、建造建筑。分别对应ControlPanel里的Button，点击后创建命令、execute()、放入命令栈。

### 3.21.2 UML图





### 3.21.3 核心代码

命令继承体系如下，基类Command具有多个派生类。基类不仅给了具体命令类共同的静态类型，方便统一管理，而且定义了具体命令应当遵守的业务逻辑模板（详见模板模式）。

 public class Command  
 {  
   // ...  
 }  
 ​  
 public class Command\_Afforest : Command  
 {  
   // ...  
 }  
 ​  
 // ...其他具体Command子类

命令栈类代码如下，负责管理其成员变量\_commands，可以存储执行过的命令、加入新命令、回退命令。

 public class CommandStack  
 {  
     // ...  
 ​  
     private Stack<Command> \_commands = new Stack<Command>();  
     public void Push(Command command)  
    {  
         \_commands.Push(command);  
    }  
     public Command Pop()  
    {  
         return \_commands.Pop();  
    }  
 }

### 3.21.4 优缺点分析

**优点：**

1. **单一职责原则**。命令发送者和接收者被解耦，双方只需负责好自己的命令收发任务，而不需要和对方打交道。
2. **开闭原则**。 可以通过增加具体命令类方便地进行扩展，而不需要修改现有代码。
3. 可以实现撤销和恢复功能。

**缺点：**

1. 代码变复杂，命令发送者和命令接收者之间增加了一个新的层次。

# 附录

|  |  |  |
| --- | --- | --- |
| **成员** | **负责工作** | **贡献比例** |
| 卢嘉霖 | 主要代码，命令模式 | 10% |
| 袁泓博 | Unity游戏界面开发 | 10% |
| 陈莫之 | 工厂模式，抽象工厂模式 | 10% |
| 李昊峰 | 单例模式，建造者模式，原型模式 | 10% |
| 宋子阳 | 适配器模式，桥接模式，过滤器模式 | 10% |
| 李锦霖 | 组合模式，装饰器模式，外观模式 | 10% |
| 王梓鉴 | 责任链模式，迭代器模式，中介者模式 | 10% |
| 张楠 | 备忘录模式，观察者模式 | 10% |
| 李金昊 | 状态模式，策略模式 | 10% |
| 郭子瑜 | 模板模式，访问者模式 | 10% |