3.1 外观模式

3.1.1 实现描述

外观模式定义了一系列算法，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

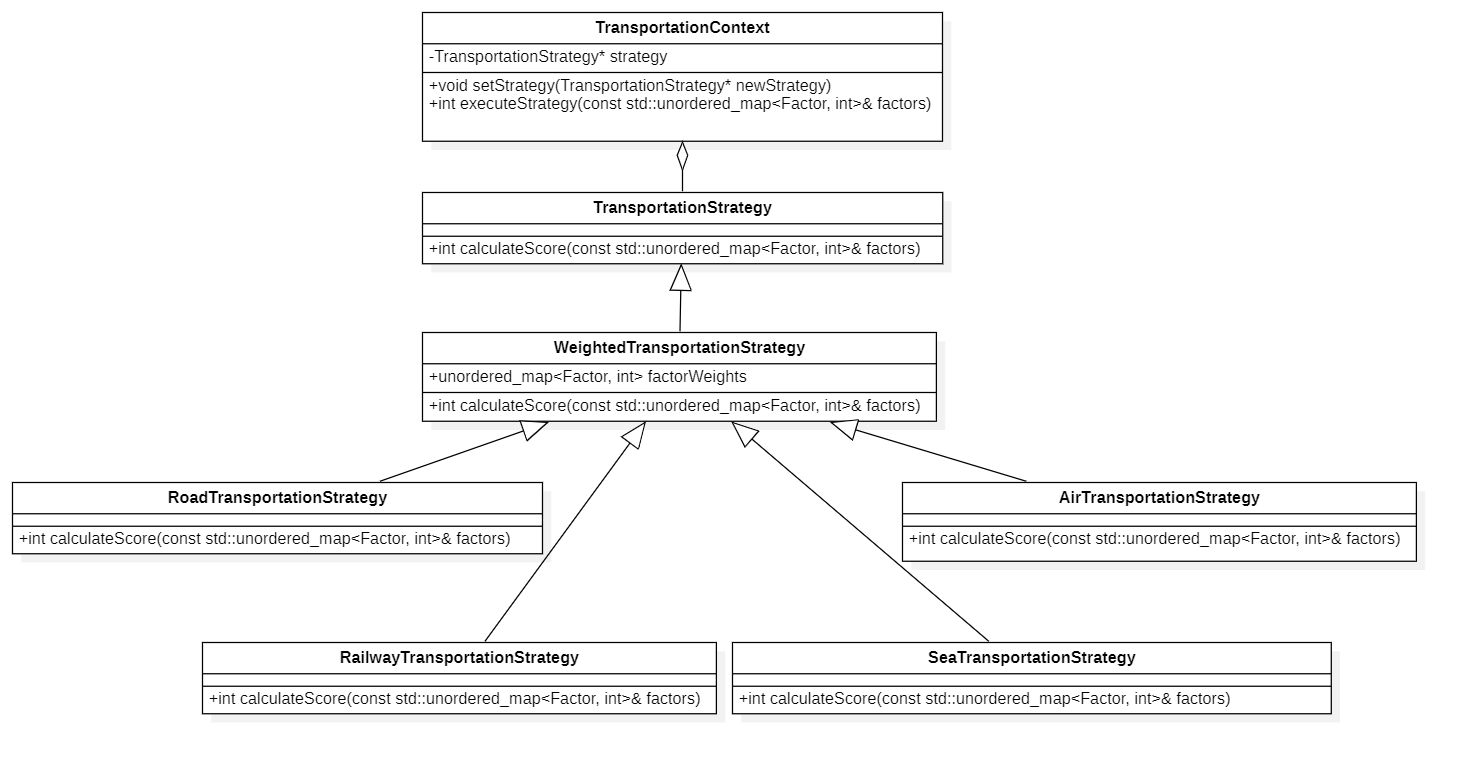
策略模式的主要角色包括定义策略类所需的接口的抽象策略（Strategy）类，和实现了抽象策略定义的接口，提供具体的算法实现或行为的具体策略（Concrete Strategy）类以及持有一个策略类的引用，最终给客户端调用的环境（Context）类

在本案例中，考虑对于新能源汽车有多种不同的运输方式，每一种外界条件在其中的权重不同，当用户输入当前的条件时，自动推荐最佳运输方式。模拟参数如下：

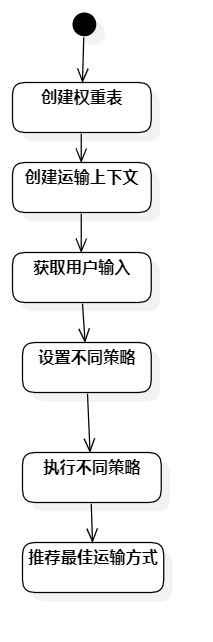
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 公路运输 | 铁路运输 | 海路运输 | 航空运输 |
| 距离 | 8 | 7 | 9 | 8 |
| 时间 | 7 | 6 | 5 | 9 |
| 成本 | 8 | 7 | 9 | 6 |
| 安全性 | 7 | 8 | 6 | 9 |
| 数量 | 7 | 8 | 9 | 6 |

首先定义了一个运输策略抽象类，然后分别定义四个运输策略类继承于该抽象类，并且重写其中的得分计算方法，然后定义一个策略上下文，供用户调用。

3.1.2 类图以及流程图



1. WeightTable 类负责存储权重表。
2. TransportationContext 类为上下文类，包含一个指向 TransportationStrategy 接口的引用，可以在运行时设置不同的具体策略对象。
3. TransportationStrategy 接口定义了算法家族的共同接口。
4. WeightedTransportationStrategy 类是具体策略的基类，包含了因素权重的处理逻辑。
5. RoadTransportationStrategy、RailwayTransportationStrategy、SeaTransportationStrategy、AirTransportationStrategy 分别是具体的策略类，实现了具体的评分计算逻辑。
6. TransportationMode 和 Factor 是枚举类型，表示运输方式和因素。



1. 首先建立不同运输方式的权重表，然后创建运输上下文，接着获取用户输入，使用dijkstra算法不同运输方式的距离，然后执行不同策略，最后推荐最佳的运输方式

3.1.3 代价分析

1. 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。
2. 策略模式将造成产生很多策略类，可以通过使用享元模式在一定程度上减少对象的数量。
3. 类膨胀：每个运输方式都有一个对应的策略类，如果运输方式的数量较多，可能导致类的数量增加，引起一定的类膨胀。
4. 维护成本：需要确保策略接口的稳定性，因为修改接口可能会影响到所有的具体策略类。这可能增加了维护成本，尤其是在接口发生变化时。
5. 运行时开销：在运行时选择算法可能引入一些开销，尤其是在频繁切换算法的情况下。如果性能是关键考虑因素，可能需要谨慎选择是否使用策略模式。
6. 适用场景：如果每个运输方式的评分逻辑相对简单且不太可能变化，引入策略模式可能过于繁琐。适用场景更倾向于在算法复杂或者经常变化时使用。

3.1.4 其他内容

总结为以下场景时使用策略模式较为合适：

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。