抽象工厂学习报告 智能2402 符航康 202408040228

一、类的作用及类与类之间的关系

1.1 类的作用

本项目旨在通过抽象工厂模式管理一家车辆销售企业的多种车辆类型，包括燃油车、电动车、混合动力车和摩托车。主要涉及以下类：

- **抽象工厂类（`abstractFactory`）**：定义了创建车辆的接口。
- **具体工厂类（`carFactory` 和 `motorFactory`）**：实现了抽象工厂接口，用于创建具体类型的车辆。
- **抽象产品类（`vehicle`）**：定义了车辆的基本接口，包括输入、输出和计算综合里程的方法。
- **具体产品类（`gasCar`、`electricCar`、`mixCar` 和 `motor`）**：继承自 `vehicle`，实现了不同类型车辆的具体属性和行为。
- **链表管理类（`vehicleNode`）**：用于管理车辆对象的链表，实现车辆的插入、排序和展示功能。

1.2 类与类之间的关系

- **工厂与产品关系**：`abstractFactory` 是抽象工厂类，`carFactory` 和 `motorFactory` 是具体工厂类，负责创建具体的车辆产品（如 `gasCar`、`electricCar`、`mixCar` 和 `motor`）。
- **继承关系**：
  - `gasCar`、`electricCar` 和 `mixCar` 继承自 `car` 类，而 `car` 类继承自 `vehicle`。
  - `motor` 类直接继承自 `vehicle`。
- **关联关系**：
  - `vehicleNode` 类通过指针关联 `vehicle` 对象，实现链表的数据结构。
  - `main` 函数通过工厂类创建车辆对象，并将其插入到 `vehicleNode` 链表中。

二、类成员以及类方法的作用以及实现过程

2.1 抽象工厂类（`abstractFactory`）

抽象工厂类 `abstractFactory` 定义了一个纯虚函数 `BuildVehicle(int style)`，该方法用于创建车辆对象。这个接口由具体工厂类实现，允许根据不同的参数生成不同类型的车辆。

2.2 具体工厂类（`carFactory` 和 `motorFactory`）

具体工厂类 `carFactory` 和 `motorFactory` 继承自 `abstractFactory`，并实现了 `BuildVehicle` 方法。`carFactory` 根据传入的 `style` 参数创建不同类型的汽车对象，包括

燃油车（`gasCar`）、电动车（`electricCar`）和混合动力车（`mixCar`）。如果 `style` 参数为1，则创建燃油车；为2时创建电动车；其他情况则创建混合动力车。`motorFactory` 则专门用于创建摩托车对象，不需要根据 `style` 参数区分具体类型，直接返回一个新的 `motor` 对象。

2.3 抽象产品类（`vehicle`）

抽象产品类 `vehicle` 定义了三个纯虚方法：`in()`、`out()` 和 `sumMile()`。`in()` 方法用于输入车辆的属性，`out()` 方法用于输出车辆的信息，`sumMile()` 方法用于计算车辆的综合里程。这些方法由具体产品类实现，以适应不同类型车辆的具体需求。

2.4 具体产品类（`gasCar`、`electricCar`、`mixCar` 和 `motor`）

具体产品类 `gasCar`、`electricCar`、`mixCar` 和 `motor` 继承自 `vehicle`，并实现了各自的属性和方法。

- `gasCar` 类包含品牌（`brand`）、油箱大小（`power`）和最大行驶里程（`mile`）三个成员变量。其 `in()` 方法用于输入这三个属性，`out()` 方法用于输出燃油车的详细信息，`sumMile()` 方法返回燃油车的最大行驶里程。
- `electricCar` 类包含品牌（`brand`）、电池容量（`battery`）和最大续航里程（`emile`）三个成员变量。其 `in()` 方法用于输入这三个属性，`out()` 方法用于输出电动车的详细信息，`sumMile()` 方法返回电动车的最大续航里程。
- `mixCar` 类包含品牌（`brand`）、油箱大小（`power`）、最大行驶里程（`mile`）、电池容量（`battery`）和纯电续航里程（`emile`）五个成员变量。其 `in()` 方法用于输入这些属性，`out()` 方法用于输出混合动力车的详细信息，`sumMile()` 方法返回油箱行驶里程与电池续航里程之和，作为混合动力车的综合里程。
- `motor` 类包含品牌（`brand`）、轮子数量（`wheels`）和最大行驶里程（`mile`）三个成员变量。其 `in()` 方法用于输入这些属性，`out()` 方法用于输出摩托车的详细信息，`sumMile()` 方法返回摩托车的最大行驶里程。

2.5 链表管理类（`vehicleNode`）

链表管理类 `vehicleNode` 通过静态成员变量 `cnt` 和 `head` 分别记录车辆的数量和链表的头节点。每个 `vehicleNode` 对象包含一个指向 `vehicle` 对象的指针 `s` 以及一个指向下一个节点的指针 `next`。

默认构造函数初始化 `next` 指针为 `NULL`。参数化构造函数接受一个 `vehicle` 指针作为参数，初始化 `s` 指针指向该车辆对象，并将 `next` 指针设为 `NULL`，同时将车辆计数 `cnt` 增加1。析构函数负责销毁链表中的所有节点，释放内存资源。

`vehicleNode` 类重载了小于运算符，通过比较两个车辆节点的综合里程来实现排序功能。`insertNode` 方法将新的车辆节点插入到链表的末尾，通过遍历链表找到最后一个节点并将新节点连接上去。`show` 方法遍历链表，依次调用每个车辆对象的 `out()` 方法，输出车辆的信息。`sortL` 方法采用冒泡排序算法，对链表中的车辆按照综合里程从大到小进行排序，确保最终输出的车辆列表符合要求。

2.6 主函数（`main.cpp`）

主函数首先创建了 `carFactory` 和 `motorFactory` 的实例，用于后续车辆的创建。然后进入一个循环，持续接收用户输入的车辆类型（1-4）。根据用户的选择，调用相应的工厂类的 `BuildVehicle` 方法创建相应类型的车辆对象。创建完车辆对象后，调用其 `in()` 方法输入车辆的具体属性，并将该车辆对象插入到 `vehicleNode` 链表中。

当用户输入结束（例如通过 Ctrl+Z 结束输入）后，主函数调用 `vehicleNode` 的 `sortL()` 方法对链表中的车辆按照综合里程进行排序，随后调用 `show()` 方法输出排序后的车辆列表。最后，程序释放工厂对象的内存资源并结束运行。

三、对抽象工厂模式的基本认识

抽象工厂模式是一种创建型设计模式，它提供一个接口，用于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。该模式强调产品的系列性，即一组相关的产品应该一起被使用。在本项目中，抽象工厂模式的应用体现在定义了一个抽象工厂类 `abstractFactory`，该类提供了一个创建车辆对象的接口 `BuildVehicle(int style)`。具体工厂类 `carFactory` 和 `motorFactory` 分别实现了这一接口，负责创建具体类型的汽车和摩托车。

这种设计使得系统在扩展新的产品类型时，只需添加新的具体工厂类和具体产品类，而无需修改现有的代码结构，从而实现了良好的扩展性和可维护性。抽象工厂模式的优点在于封装了对象的创建过程，提高了系统的灵活性和一致性，确保同一系列的产品被一起使用。然而，该模式也增加了系统的复杂性，因为需要为每个产品族创建相应的工厂和产品类，可能导致类的数量增多。此外，若需要添加一个新的产品类别，必须修改抽象工厂接口及所有具体工厂类，这在一定程度上增加了维护成本。

四、学习心得

通过此次学习和实践，我深入理解了抽象工厂模式在实际项目中的应用。设计模式为软件开发提供了成熟的解决方案，使代码结构更加清晰、可维护性更高。抽象工厂模式尤其适用于需要创建多个相关对象的场景，能够有效地管理和扩展系统功能。通过继承和多态，我体会到了面向对象编程的优势，不仅实现了代码的复用和扩展，还提高了系统的灵活性。链表数据结构的应用让我了解到如何动态管理数据，并通过重载运算符和排序算法，实现了车辆列表的按需排序和展示。代码的模块化设计将不同的功能模块分离到不同的类和文件中，极大地提高了代码的可读性和可维护性。这不仅加深了我对抽象工厂模式的理解，也提升了我在面向对象编程和设计模式应用方面的实际能力。