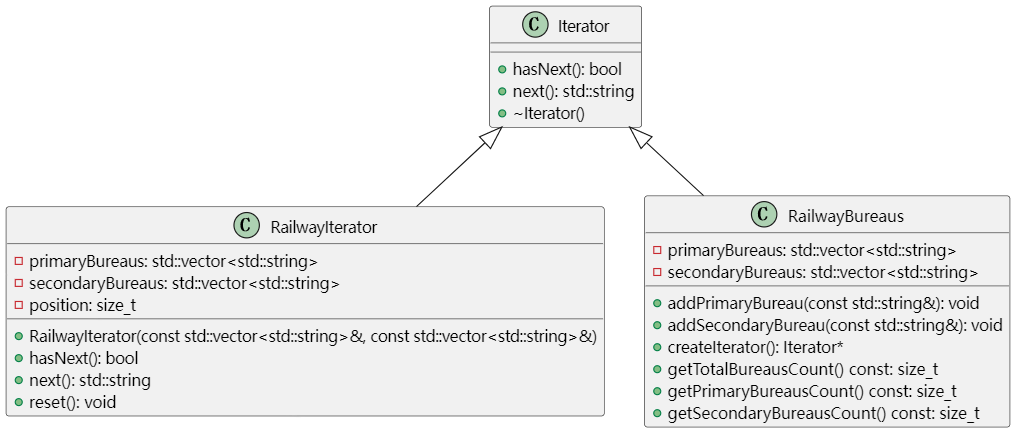
迭代器模式介绍

# 实现功能

实现了一个铁路基地管理系统。可以向铁路基地系统中添加基地并获取基地数量，然后使用迭代器遍历并输出不同级别的基地信息。

# 类图



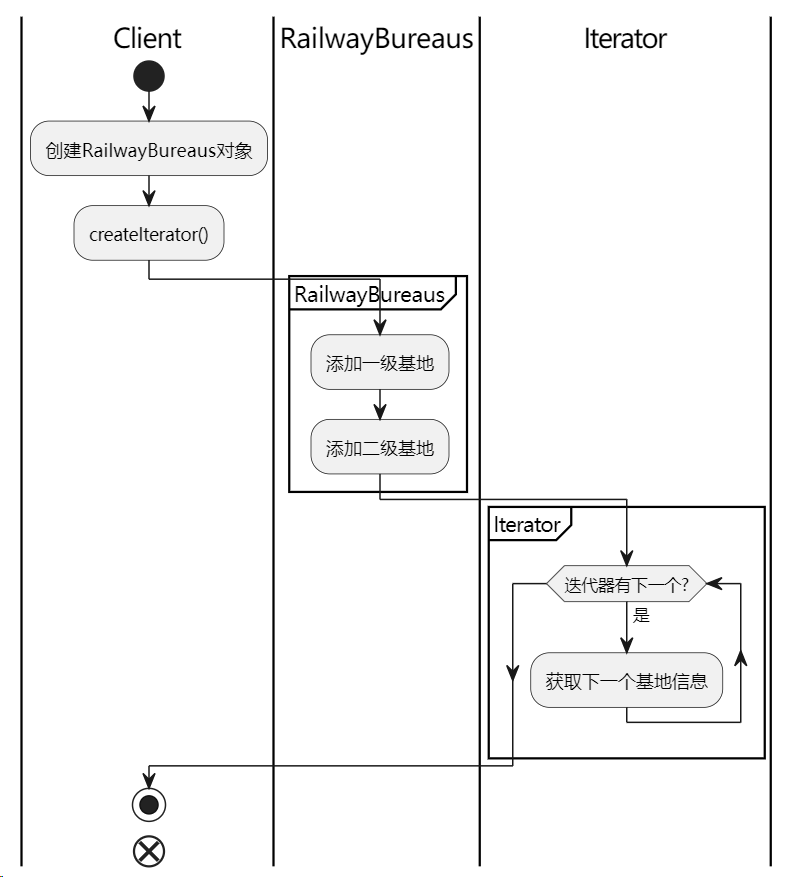
Iterator: 这是一个抽象类，定义了遍历集合的接口。其中有两个纯虚函数 hasNext() 和 next()，用于检查是否存在下一个元素以及获取下一个元素的值。它还有一个虚析构函数来确保正确释放资源。

RailwayIterator: 这个类继承自 Iterator，实现了在铁路基地集合上的迭代功能。它包含了两个 std::vector<std::string> 成员变量，分别用于存储一级和二级基地，以及一个 position 成员变量来追踪当前位置。它重写了 hasNext() 和 next() 函数来支持对一级和二级基地集合的迭代，并且提供了 reset() 函数来重置迭代器。

RailwayBureaus: 这个类用于管理铁路基地集合。它有两个私有成员变量 primaryBureaus 和 secondaryBureaus，分别代表一级和二级基地的集合。它提供了添加一级和二级基地的函数 addPrimaryBureau() 和 addSecondaryBureau()，以及创建迭代器的函数 createIterator()。此外，它还提供了获取基地数量的函数 getTotalBureausCount()、getPrimaryBureausCount() 和 getSecondaryBureausCount()。

main: 主函数演示了如何使用 RailwayBureaus 类来管理一级和二级基地，以及如何使用 Iterator 类来遍历并显示这些基地的信息。

# 流程图



客户端（Client） 开始流程，并创建了一个 RailwayBureaus 对象。

客户端调用了 createIterator() 方法创建了一个迭代器对象。

RailwayBureaus 部分显示了一级基地和二级基地的添加过程。

Iterator 部分展示了迭代器的初始化和迭代遍历过程，通过循环检查迭代器是否有下一个基地信息，并获取这些基地信息。

最后，流程结束，客户端停止。

# 代价分析

复杂性增加：

引入迭代器模式会增加代码的复杂性。虽然迭代器模式可以将遍历逻辑封装在迭代器类中，但可能会导致类的数量增加，增加系统的复杂性和理解成本。

迭代器对象生命周期管理：

客户端代码需要手动管理迭代器对象的生命周期，包括创建和销毁。在实际应用中，可能需要更好地管理迭代器对象的生命周期，以避免悬挂指针或者访问非法对象的情况。

维护成本增加：

如果集合对象的内部结构发生变化，可能需要对迭代器进行相应的调整以保持一致性。这可能会增加代码维护成本。

性能开销：

在某些情况下，使用迭代器模式可能会带来一定的性能开销。因为迭代器需要额外的函数调用和对象创建，可能在遍历集合时产生一些额外的性能开销。

灵活性限制：

虽然迭代器模式提供了一种通用的遍历方式，但有时也可能限制了对集合的灵活操作。特定类型的集合可能需要更复杂的遍历方式或迭代器，这可能不太适合迭代器模式的简单实现。

不适用于小规模集合：

对于小规模集合，引入迭代器模式可能显得过于复杂，使用简单的循环或其他直接的遍历方式更为合适。