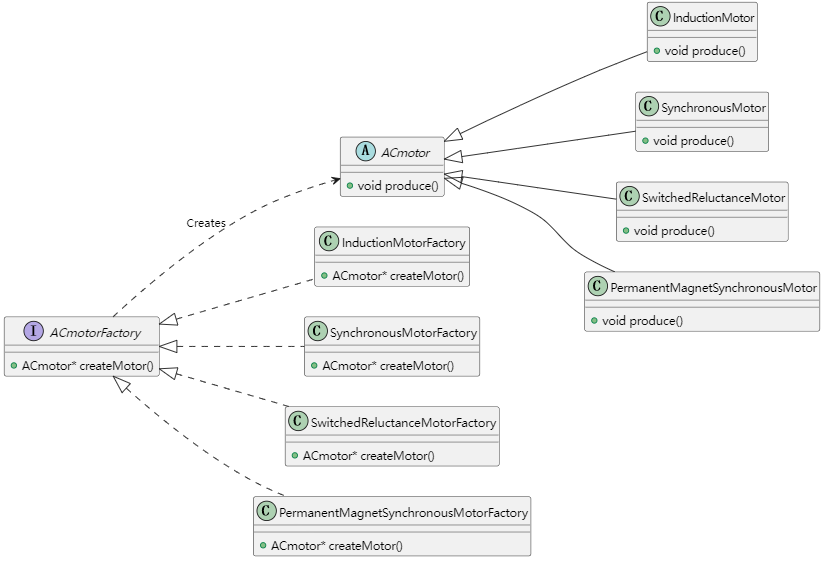
工厂模式介绍

# 实现功能

通过工厂方法模式创建不同类型的交流电机对象，并展示了在客户端如何利用这些工厂来生产所需类型的电机。通过此设计，客户端代码（main 函数）不需要直接实例化具体的产品类，而是依赖于抽象的工厂接口，通过不同的工厂类创建不同类型的产品对象。这使得客户端代码与具体产品的创建过程分离，提高了代码的灵活性和可维护性。

# 类图



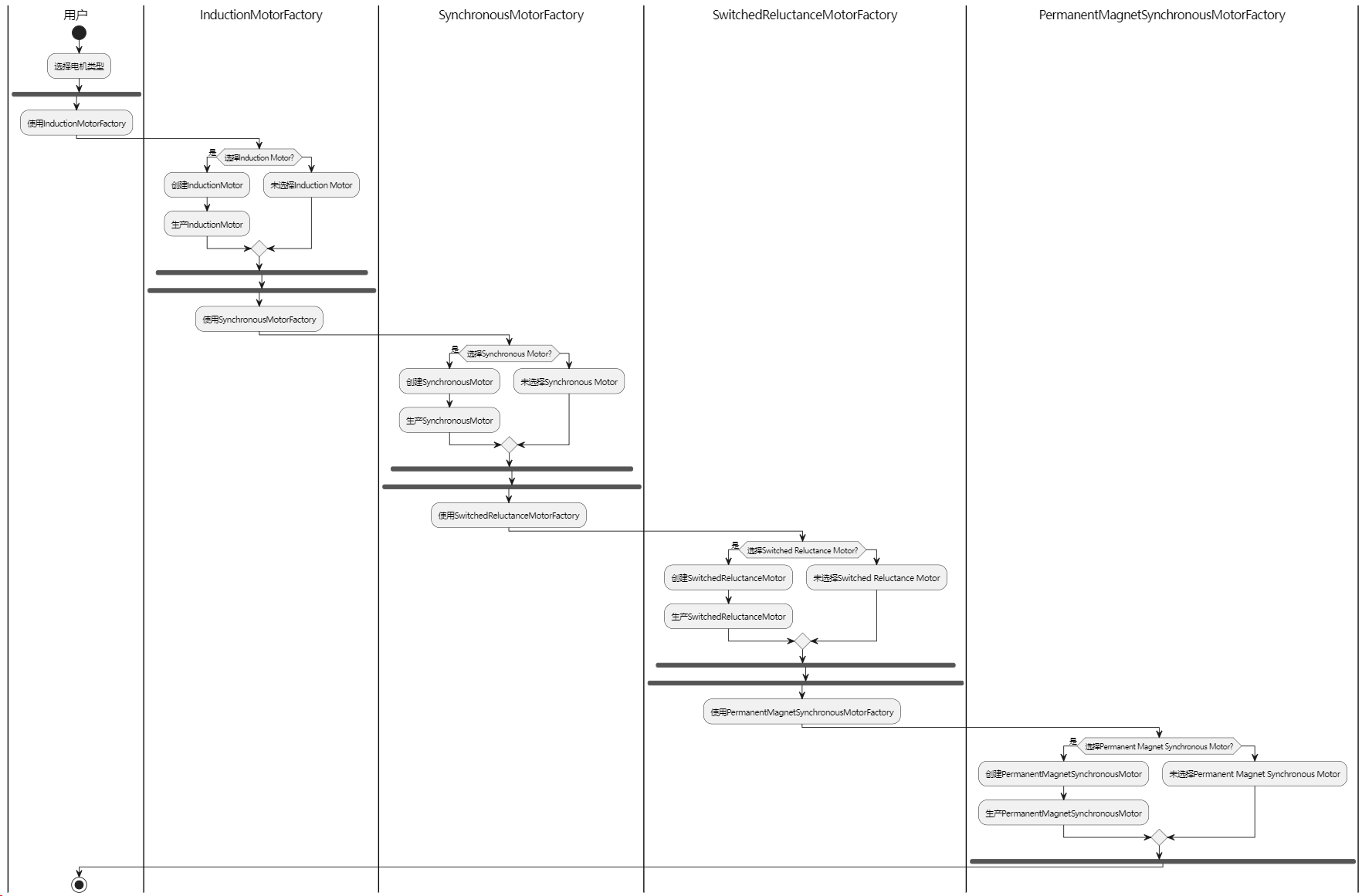
不同类型的交流电机类（InductionMotor、SynchronousMotor、SwitchedReluctanceMotor、PermanentMagnetSynchronousMotor）：派生自 ACmotor，并重写了 produce() 函数以生产不同类型的交流电机。

抽象工厂类 ACmotorFactory：定义了创建交流电机的接口 createMotor()，派生类需要实现此函数以创建具体的交流电机。

具体的交流电机工厂类（InductionMotorFactory、SynchronousMotorFactory、SwitchedReluctanceMotorFactory、PermanentMagnetSynchronousMotorFactory）：派生自 ACmotorFactory，并实现了 createMotor() 函数以创建对应类型的交流电机。

客户端代码：在 main() 函数中，创建了不同类型的具体工厂对象，并调用它们的 createMotor() 函数创建相应的电机，然后调用 produce() 函数进行生产。

# 流程图



用户开始并选择电机类型。

接下来，用户同时进行了四个选择分支：

对于每种电机类型，都有一个相应的工厂在等待用户的选择。

如果用户选择了某个特定类型的电机，则在该类型的工厂分支中创建并生产相应的电机。每个分支独立检查用户是否选择了对应的电机类型，然后创建并生产该类型的电机；否则，显示未选择该类型的提示信息。

每个分支独立地完成其任务后，活动图结束并返回给用户。

# 代价分析

内存管理问题：

在代码中使用了 new 操作符来创建对象，并且在后续使用 delete 来释放内存。在实际应用中，如果没有正确的内存管理和资源释放，可能会导致内存泄漏或者程序中出现悬空指针等问题。

耦合性问题：

在示例中，客户端代码需要直接与具体的工厂类交互，创建特定的工厂对象来生成特定类型的电机。这可能导致代码耦合度较高，如果在将来需要更改或添加新类型的电机，可能需要修改客户端代码，违反了开闭原则。

对象创建方式固定：

每个具体工厂类只能创建对应类型的电机，这种固定的创建方式可能会限制系统的灵活性。如果需要添加新类型的电机，就需要创建新的具体工厂类，并在客户端代码中进行相应的修改。

潜在的性能问题：

每次创建对象都会涉及动态内存分配和释放，这可能在运行时造成一定程度的性能开销。在频繁调用对象创建的场景下，可能需要考虑这种开销对系统性能的影响。

缺乏异常处理：

代码示例中没有包含异常处理机制，如果在对象创建或其他地方出现异常，可能会导致程序异常终止或者不可预测的行为。

未考虑对象生命周期：

客户端代码在创建电机后负责释放内存。在实际应用中，可能需要更好地管理对象的生命周期，以避免出现内存泄漏或者使用非法对象的情况。