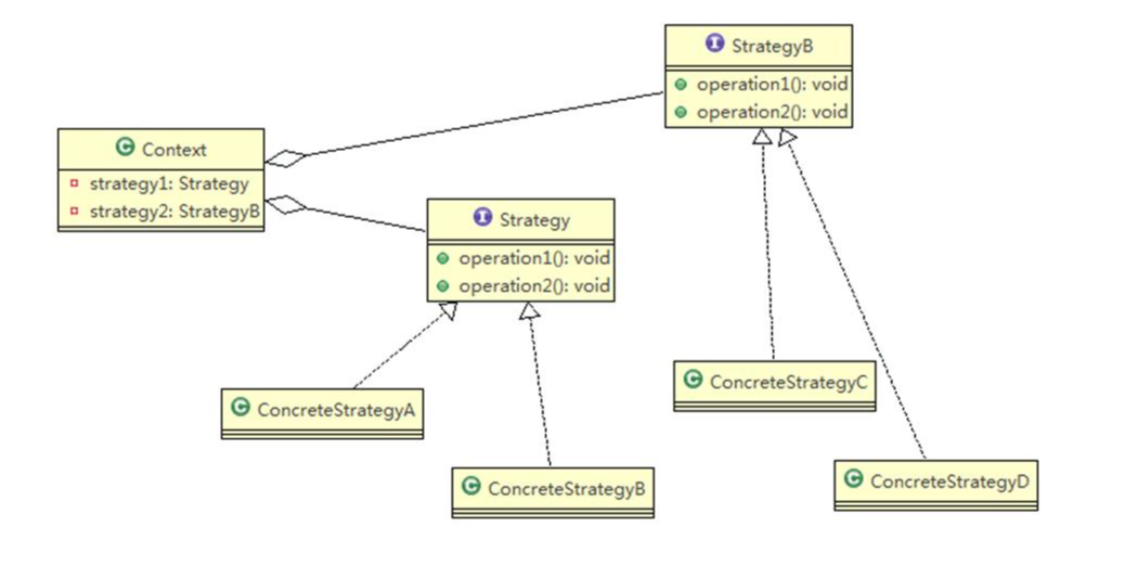
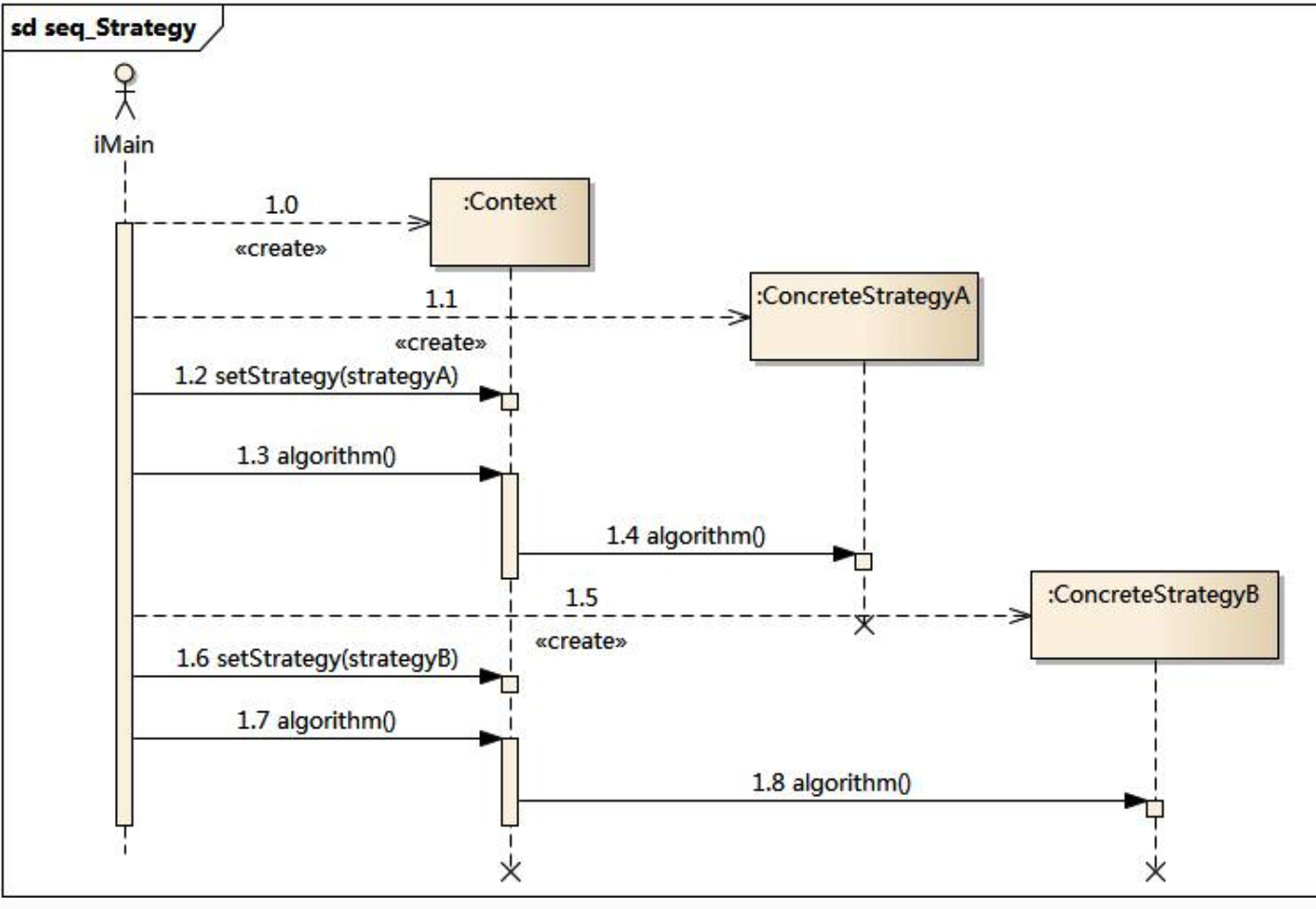
1. 策略模式(Strategy Pattern): 定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法独立于使用它的客户而变化，也称为政策模式(Policy)
2. 策略模式对应于解决某一个问题的一个算法族, 允许用户从该算法族中任选一个算法解决某一问题同时可以方便的更换算法或者增加新的算法. 并且由客户端决定调用哪个算法. 例如根据不同用户选择不同的报价策略等
3. 本质: **分离算法, 选择实现**。它把算法的责任和算法本身分割开，委派给不同的对象管理。在策略模式中，应当由客户端自己决定在什么情况下使用什么具体策略角色。这在一定程度上提高了系统的灵活性，但是客户端需要理解所有具体策略类之间的区别，以便选择合适的算法，这也是策略模式的缺点之一，在一定程度上增加了客户端的使用难度。
4. 关键是分析项目中变化部分和不变化的部分，其核心思想是 **多用组合/聚合，少用继承**，用行为类组合而不是行为的继承
5. 策略模式结构图

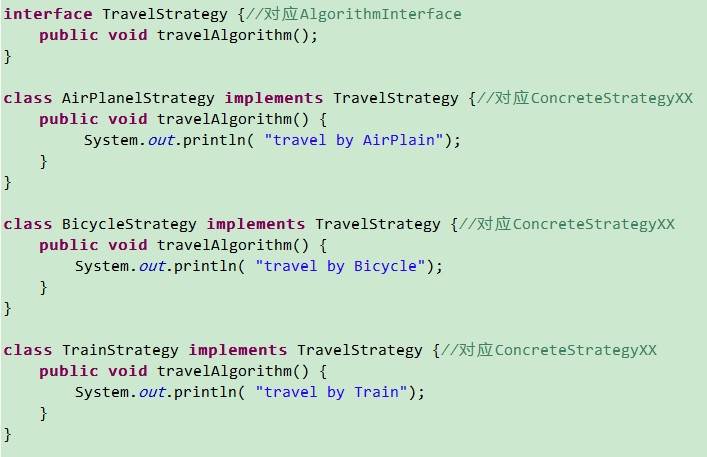


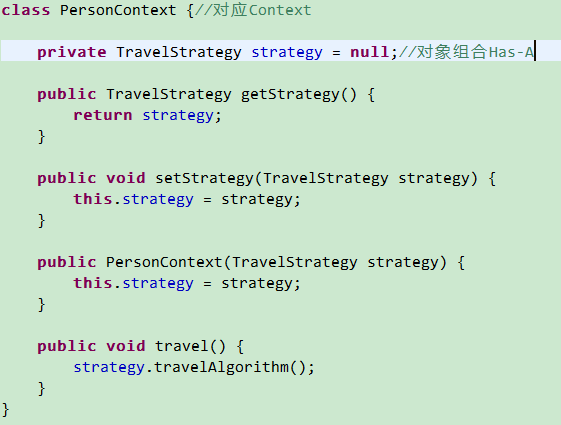
* 1. 环境类(Context): 用一个ConcreteStrategy对象来配置。**维护一个对Strategy对象的引用(对象组合Has-A)**。可定义一个接口来让Strategy访问它的数据。
  2. 抽象策略类(Strategy): 定义所有支持的算法的公共接口。 Context使用这个接口来调用某ConcreteStrategy定义的算法。
  3. 具体策略类(ConcreteStrategy): 以Strategy接口实现某具体算法。

1. 时序图

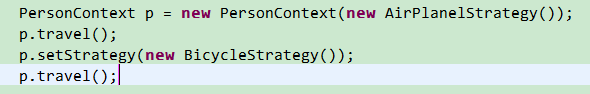


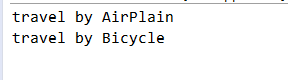
1. 例1，出行方式选择



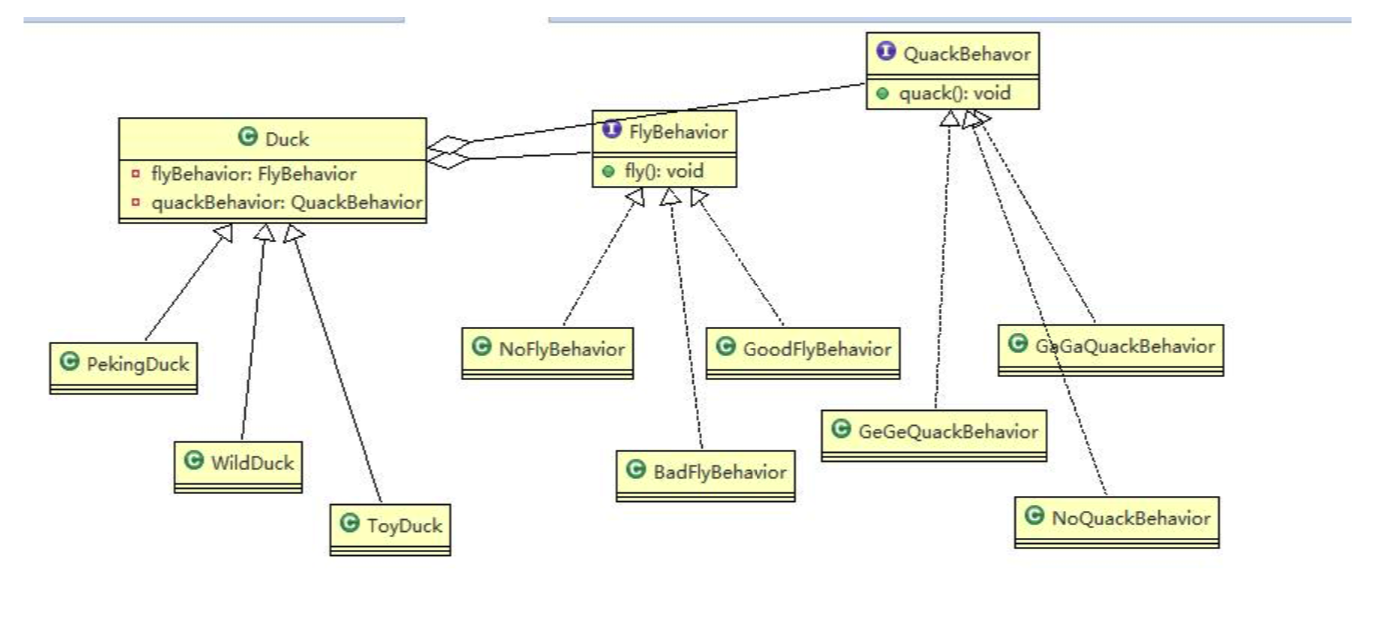


测试：



输出：

1. 例2，鸭子问题类图



1. 策略模式在JDK的java.util.Arrays#sort(T[], java.util.Comparator<? super T>)方法中就使用了策略模式

|  |
| --- |
| Integer[] arr = {1**,** 20**,** 5**,** 30**,** 15}**;** Arrays.*sort*(arr**,** Comparator.*reverseOrder*())**;** |

* 1. Comparator充当Strategy接口
  2. 使用时候传入方法的具体实现充当具体的策略

1. 在软件开发中也常常遇到类似的情况，实现某一个功能有多种算法或者策略，我们可以根据环境或者条件的不同选择不同的算法或者策略来完成该功能。如查找、排序等，一种常用的方法是硬编码(Hard Coding)。在一个类中，如需要提供多种查找算法，可以将这些算法写到一个类中，在该类中提供多个方法，每一个方法对应一个具体的查找算法；当然也可以将这些查找算法封装在一个统一的方法中，通过if…else…或者case等条件判断语句来进行选择。这两种实现方法我们都可以称之为硬编码，如果需要增加一种新的查找算法，需要修改封装算法类的源代码；更换查找算法，也需要修改客户端调用代码。在这个算法类中封装了大量查找算法，该类代码将较复杂，维护较为困难。如果我们将这些策略包含在客户端，这种做法更不可取，将导致客户端程序庞大而且难以维护，如果存在大量可供选择的算法时问题将变得更加严重
2. 优缺点：
   1. 策略模式的优点
      1. 策略模式提供了对“开闭原则”的完美支持，用户可以在不修改原有系统的基础上选择算法或行为，也可以灵活地增加新的算法或行为。
      2. 策略模式提供了管理相关的算法族的办法。
      3. 策略模式提供了可以替换继承关系的办法。
      4. 使用策略模式可以避免使用多重条件转移语句
   2. 策略模式的缺点
      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. 适用环境
   1. 如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。
   2. 一个系统需要动态地在几种算法中选择一种。
   3. 如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。
   4. 不希望客户端知道复杂的、与算法相关的数据结构，在具体策略类中封装算法和相关的数据结构，提高算法的保密性与安全性