**《软件工程》**

**实验报告七：设计模式以及构件设计**

**姓 名： 向申赤 学 号： 202210120510**

**院 系： 计算机与信息学院 专 业： 计算机科学与技术**

**实 验 室： J1-306 实验日期： 2024.12.02**

**总评成绩： 审阅教师： 杨青**

# 实验目的

1. 了解软件工程工具生成代码的方法，理解代码自动生成的过程，破除自动生成代码“黑科技”
2. 掌握构件设计方法
3. 学习Bridge设计模式，体会设计模式的作用

# 实验环境

Rational Rose Enterprise Edition, StarUML.

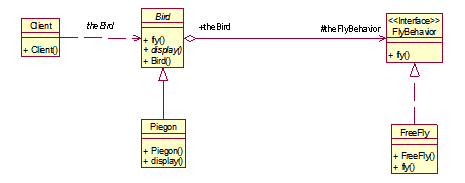
# 实验要求

1. 参考[第7章 设计概念（design principle &pattern）（4学时）](https://star.jmhui.com.cn/u/cms/www/202305/08213040s60e.pdf)，学习Bridge设计模式的使用；
2. 学习 Rose 生成代码的方法，理解代码自动生成原理
3. 文件以“学号-姓名-软件工程实验七.rar”的方式命名，把模型文件和工程实现代码打包，提交到长江雨课堂**软件工程实验七**；

# 实验内容

## 按照给出的 Bridge 设计模式 构建类图

有一个叫 HuntBird 的游戏，里面需要表示各种各样的鸟类。采用Bridge设计模式实现，针对各种鸟类，采用Bridge设计模式建模，如下图：



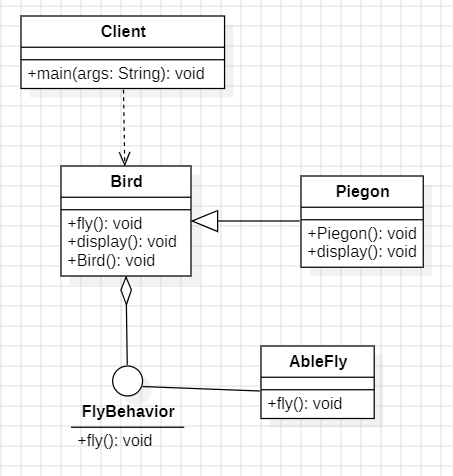
注意：

（1）绘制类之前，首先设置默认语言为Java/C++（本例以Java说明）

（2）Bird类为抽象类

（3）display方法为抽象方法

（4）Bird具有保护类型的FlyBehavior属性；



Bird类作为抽象类，定义了鸟类的基本属性和行为接口，包括一个保护类型的FlyBehavior属性和抽象的display方法。通过引入FlyBehavior接口，将飞行行为从Bird类中分离出来，实现行为的灵活扩展。这种设计方式符合面向对象设计的开放封闭原则（OCP），可以在不修改现有代码的情况下扩展新功能。

具体实现中，FlyBehavior接口作为行为的抽象定义，能够灵活地由不同的实现类扩展。例如，通过设计UnableFly类，实现无法飞翔的具体逻辑；通过设计AbleFly类，则可实现正常飞行的逻辑。类图的设计不仅体现了抽象与实现的分离，也为后续功能扩展提供了良好的基础。

## 按照以下给出的要求，扩展类图，并生成代码框架，实现代码，测试

(1)添加一个飞翔的行为的实现类UnableFly，实现fly接口方法：扑腾几下飞不起来；

(2)添加游泳的行为SwimBehavior接口，两个实现类AbleSwim，实现swim接口方法：鱼翔浅底；UnableSwim实现类实现swim接口方法：扑腾几下就淹死了：

(3)添加两个具体类：老鹰Eagle和企鹅Penguin，并实现相应代码，即构造函数和display抽象方法

(4)使用Client类，测试以上代码

图示

描述已自动生成

在完成Bridge设计模式基础类图构建后，实验进一步扩展类图以增加飞行和游泳行为，并生成代码框架进行实现与测试。飞行行为通过新增UnableFly类实现“扑腾几下飞不起来”的逻辑，游泳行为则引入SwimBehavior接口和其实现类AbleSwim（“鱼翔浅底”）与UnableSwim（“扑腾几下就淹死了”），展现了Bridge模式对行为扩展的灵活性。

具体鸟类方面，设计了Eagle（老鹰）和Penguin（企鹅）两个类。Eagle通过AbleFly和AbleSwim组合行为，体现飞行与游泳能力，而Penguin则组合UnableFly和UnableSwim，表现其无法飞翔且游泳失败的特点。每个具体类均重写了display方法，用于展示各自特性。

在实现代码框架后，通过Client类对系统进行测试。实验验证了不同鸟类实例化及其行为调用的正确性，例如Eagle类成功表现为“能够飞翔”，而Penguin则表现为“无法飞翔且游泳失败”。测试结果表明系统设计遵循开放封闭原则（OCP），行为扩展无需修改原有代码，同时动态绑定和多态性功能得到了验证。整个过程充分体现了Bridge设计模式的灵活性与实用性。

## 根据以下需求说明，设计符合面向对象设计原则的 UML 类图并实现代码。

假如我们正在开发一个酒店预订系统，针对不同的用户，我们需要计算出不同的房价。比如，普通用户是全价，金卡是 8 折，银卡是 9 折。

图示

描述已自动生成

实验扩展任务提出了酒店预订系统的需求，要求根据不同用户类型（普通用户、金卡用户、银卡用户）计算不同的房价。这部分任务采用策略模式（Strategy Pattern），设计了一个动态折扣计算的解决方案。房价的计算逻辑被封装在独立的策略类中，例如FullPriceStrategy（全价策略）、GoldCardStrategy（8折策略）和SilverCardStrategy（9折策略）。用户类通过组合具体策略对象，实现了灵活的折扣动态变化。这种设计方式体现了依赖倒置原则（DIP）和单一职责原则（SRP），使得房价计算的逻辑独立于用户类，从而提升了系统的扩展性和维护性。

# 实验结论

通过本次实验，我们深入探讨了Bridge设计模式在系统解耦和行为扩展中的重要作用，并成功完成了从类图设计到代码实现的全流程实践。实验的过程和结果都表明，该模式能够显著提升系统的灵活性和可维护性。

首先，Bridge设计模式的核心在于将抽象部分和实现部分分离，从而允许它们独立变化。在本实验中，我们通过Bird类的抽象设计以及FlyBehavior和SwimBehavior接口的实现，成功将鸟类的具体行为与其种类解耦。这种设计不仅减少了类之间的耦合度，还使得新增行为时无需修改已有代码，从而满足了开放封闭原则（OCP）。例如，通过新增UnableFly和AbleSwim类，我们仅需实现相关接口，无需改动Bird类及其子类的代码结构。

其次，实验充分体现了设计模式在代码扩展中的优势。通过新增Eagle（老鹰）和Penguin（企鹅）两个具体鸟类，我们验证了Bridge模式的多维扩展能力。在传统设计中，可能需要为每种鸟类手动实现所有行为，但在本实验中，这些行为通过接口和实现类完成了复用，显著提高了开发效率。同时，行为的实现独立于具体鸟类，这种灵活性使系统能够快速响应需求的变化。

实验过程中，工具的应用也起到了重要作用。Rational Rose和StarUML等建模工具提供了从类图设计到代码生成的自动化支持，使我们能够快速验证设计的可行性。这些工具不仅提高了开发效率，也帮助我们更直观地理解了设计模式在工程实践中的作用。

此外，本实验还强调了面向对象设计原则的贯彻落实。在设计过程中，我们严格遵循单一职责原则（SRP）和依赖倒置原则（DIP）。Bird类仅专注于抽象定义，而具体行为通过独立的接口实现；类的设计依赖于行为接口，而不是具体实现。这种分层结构不仅增强了系统的可维护性，也为后续功能扩展奠定了良好的基础。

在测试阶段，我们通过Client类验证了系统设计的正确性和稳定性。具体测试包括实例化不同的鸟类并调用其特定行为，结果表明系统能够准确区分不同种类鸟类的行为表现。此外，我们还对新增行为（如UnableSwim）的功能进行了测试，确保了代码的完整性与可靠性。

本实验的案例扩展——酒店预订系统的折扣计算设计，也进一步展示了设计模式的应用价值。通过策略模式（Strategy Pattern）实现动态折扣计算，系统能够灵活应对不同用户类型的需求。这一设计与Bridge模式在行为解耦上的思路相通，体现了设计模式的通用性。

总之，本次实验不仅加深了我对Bridge设计模式的理解，也为系统设计和代码实现提供了宝贵经验。实验中涉及的工具使用和设计方法为复杂系统开发提供了实际指导，同时，模式应用的成功实践也让我意识到设计模式在提升系统灵活性和维护性方面的重要意义。未来在开发类似系统时，我将更加注重设计模式的合理运用，以打造更高效、更稳定的软件系统。

# 仓库地址

<https://github.com/xiangshenchi/soft-engineering/tree/main/SEproject7>

<https://gitee.com/help_xsz/soft-engineering/tree/main/SEproject7>