选择题

1. （ B ）系统体系结构的最佳表示形式是一个可执行的软件原型。
   A. 真
   B. 假

2. （ A ）软件体系结构描述是不同项目相关人员之间进行沟通的使能器。
   A. 真
   B. 假

3. （ A ）良好的分层体系结构有利于系统的扩展与维护。
   A. 真
   B. 假

4. （ B ）消除两个包之间出现的循环依赖在技术上是不可行的。
   A. 真
   B. 假

5. （ A ）设计模式是从大量成功实践中总结出来且被广泛公认的实践和知识。
   A. 真
   B. 假

6. 程序编译器的体系结构适合使用（ A ）。
   A. 仓库体系结构
   B. 模型－视图－控制器结构
   C. 客户机／服务器结构
   D. 以上选项都不是

7. 网站系统是一个典型的（ C ）。
   A. 仓库体系结构
   B. 胖客户机／服务器结构
   C. 瘦客户机／服务器结构
   D. 以上选项都不是

8. 在分层体系结构中，（ D ）实现与实体对象相关的业务逻辑。
   A. 表示层
   B. 持久层
   C. 实体层
   D. 控制层

选择题

1. （A）面向对象设计是在分析模型的基础上，运用面向对象技术生成软件实现环境下的设计模型。

   A. 真

   B. 假

2. （B）系统设计的主要任务是细化分析模型，最终形成系统的设计模型。

   A. 真

   B. 假

3. （B）关系数据库可以完全支持面向对象的概念，面向对象设计中的类可以直接对应到关系数据库中的表。

   A. 真

   B. 假

4. （A）用户界面设计对于一个系统的成功是至关重要的，一个设计得很差的用户界面可能导致用户拒绝使用该系统。

   A. 真

   B. 假

5. 内聚表示一个模块（B）的程度，耦合表示一个模块（D）的程度。

   A. 可以被更加细化

   B. 仅关注在一件事情上

   C. 能够适时地完成其功能

   D. 联接其他模块和外部世界

6. 良好设计的特征是（E）

   A. 模块之间呈现高耦合

   B. 实现分析模型中的所有需求

   C. 包括所有组件的测试用例

   D. 提供软件的完整描述

   E. 选项B和D

   F. 选项B、C和D

7. （A）是选择合适的解决方案策略，并将系统划分成若干子系统，从而建立整个系统的体系结构；（B）细化原有的分析对象，确定一些新的对象、对每一个子系统接口和类进行准确详细的说明。

   A. 系统设计

B. 对象设计

C. 数据库设计

D. 用户界面设计

8. 下面的（ D ）界面设计原则不允许用户保持对计算机交互的控制。

A. 允许交互中断

B. 允许交互操作取消

C. 对临时用户隐藏技术内部信息

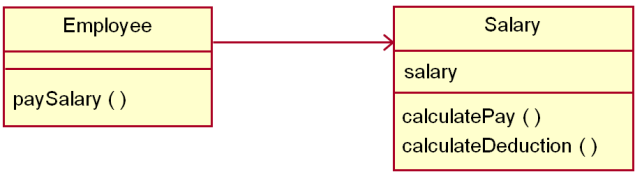
D. 只提供一种规定的方法完成任务

作业

1. 下图是某公司支付雇员薪水程序的一个简化 UML 设计类图，目前雇员薪水是按固定月薪支付的，系统需要准时支付正确的薪金，并从中扣除各种扣款。现在该公司准备增加“时薪”和 “底薪+佣金”两种支付方式，考虑到良好的可扩展性，开发人员打算使用设计模式修改原有 设计，以支持多种薪水支付方式。

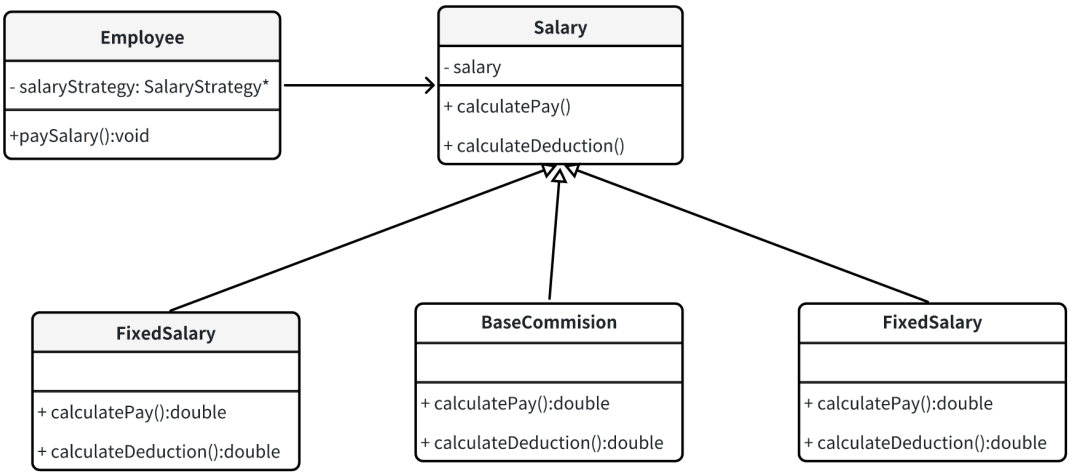
   （1） 你会选择什么设计模式？为什么？

   （2） 请画出修改后的 UML 设计类图，并用 C++语言编写实现该类图的程序。



我会选择策略模式。

考虑 Emplyee 与 Salary之间的关系，Salary 类可以看作是计算工资的策略， Emplyee可以使用不同的 Salary计算工资，这样可以更加灵活调整/更换工资计算逻辑。



```cpp
#include <iostream>

// Abstract base class for salary strategy
class Salary {
     public:
     virtual double calculatePay() const = 0;
     virtual double calculateDeduction() const = 0;
};

// Concrete class for fixed salary strategy
class FixedSalary : public Salary {
     public:
     double calculatePay() const override {
          // Calculation logic for fixed salary
          return 5000.0;
     }

     double calculateDeduction() const override {
          // Calculation logic for deduction
          return 200.0;
     }
};
// Concrete class for hourly salary strategy
class HourlySalary : public Salary {
public:
     double calculatePay() const override {
          // Calculation logic for hourly salary
          return 20.0 * 8 * 20; // $20 per hour, 8 hours per day, 20 days per month
     }

     double calculateDeduction() const override {
          // Calculation logic for deduction
          return 100.0;
     }
};

// Concrete class for base plus commission salary strategy
class BasePlusCommissionSalary : public Salary {
public:
     double calculatePay() const override {
     // Calculation logic for base plus commission salary
          return 3000.0 + (0.1 * 50000); // $3000 base salary plus 10% commission  on $50,000 sales
     }

     double calculateDeduction() const override {
          // Calculation logic for deduction
          return 300.0;
     }
};
 // Employee class
 class Employee {
 private:
      Salary* salary;

 public:
      Employee(Salary* strategy) : salary(strategy) {}

      void setSalary(Salary* strategy) {
           salary = strategy;
      }

      void paySalary() {
           double pay = salary->calculatePay();
           double deduction = salary->calculateDeduction();
           double finalPay = pay - deduction;
           std::cout << "Paid salary: $" << finalPay << std::endl;
      }
 };
```