软件体系结构与设计模式

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

---

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

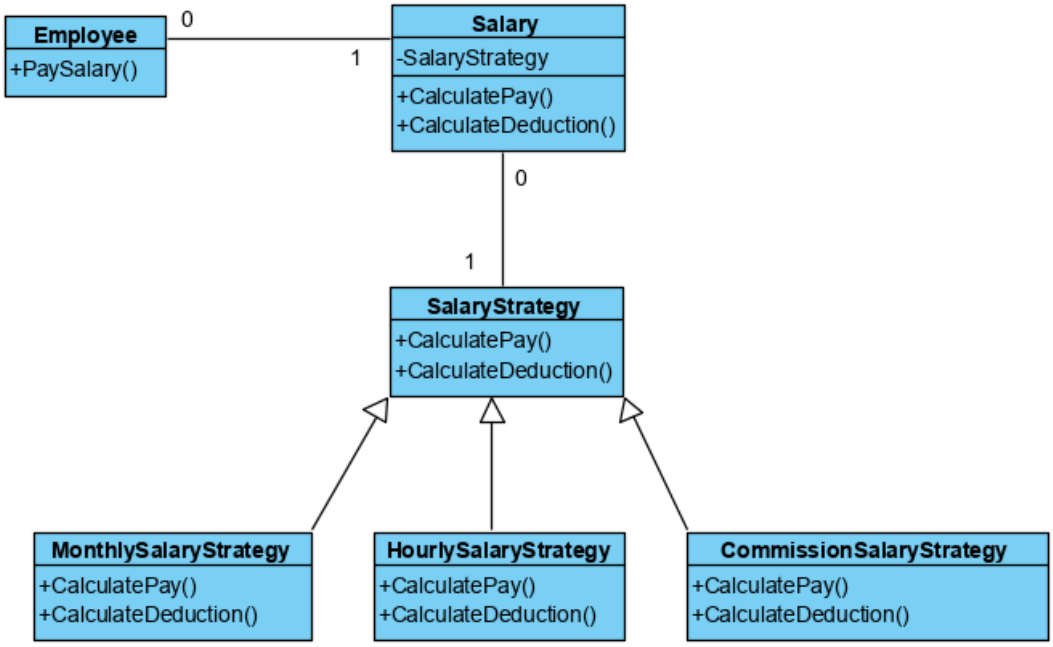
   D. 只提供一种规定的方法完成任务

9. 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？

   选择策略模式，因为策略模式有助于管理由算法过多变体而产生的复杂度，适合支持多种薪水计算方式，并且有良好的扩展性。

   （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。




```cpp
/* 基类 */
class SalaryStrategy {
public:
    virtual ~SalaryStrategy();
    virtual double CalculatePay() = 0;
    virtual double CalculateDeduction() = 0;
};

/* 固定月薪 */
class MonthlySalaryStrategy : public SalaryStrategy {
protected:
    // ...
public:
    // ...
    virtual double CalculatePay();
    virtual double CalculateDeduction();
};

/* 时薪 */
class HourlySalaryStrategy : public SalaryStrategy {
protected:
    // ...
public:
    // ...
    virtual double CalculatePay();
    virtual double CalculateDeduction();
};

/* 底薪 + 佣金 */
class CommissionSalaryStrategy : public SalaryStrategy {
protected:
    // ...
public:
    // ...
    virtual double CalculatePay();
    virtual double CalculateDeduction();
};


class Salary {
protected:
    // ...
    SalaryStrategy *strategy_;
```

```
public:
    // ...
    Salary(SalaryStrategy *strategy) : strategy_(strategy) {};

    SetStrategy(SalaryStrategy *strategy) {
        if (strategy_)
            delete strategy_;

        strategy_ = strategy;
    }

    double CalculatePay() {
        if (strategy_)
            return strategy_->CalculatePay();
        else
            return -1;
    }

    double CalculateDeduction() {
        if (strategy_)
            return strategy_->CalculateDeduction();
        else
            return -1;
    }
};

class Employee {
protected:
    // ...
    Salary salary_;
public:
    // ...
    void PaySalary() {
        // ...
        salary_.CalculatePay() - salary_.CalculateDeduction()
        // ...
    }
};
```

10. 个人独立开发一个 Game of Life 游戏，具体游戏描述如下：

> 生命游戏是英国数学家约翰·何顿·康威在 1970 年发明的细胞自动机，它包括一个二维矩形世界，这个世界中的每个方格居住着一个活着的或死亡的细胞。一个细胞在下一个时刻生死取决于相邻八个方格中活着的或死了的细胞的数量。如果相邻方格活着的细胞数量过多，这个细胞会因为资源匮乏而在下一个时刻死去；相反，如果周围活细胞过少，这个细胞会因太孤单而死去。
>
> 游戏在一个类似于围棋棋盘一样的，可以无限延伸的二维方格网中进行。例如，设想每个方格中都可放置一个生命细胞，生命细胞只有两种状态：“生”或“死”。图中，用黑色的方格表示该细胞为“死”，其它颜色表示该细胞为“生” 。游戏开始时，每个细胞可以随机地（或给定地）被设定为“生”或“死”之一的某个状态，然后，再根据如下生存定律计算下一代每个细胞的状态：
>
> 1. 每个细胞的状态由该细胞及周围 8 个细胞上一次的状态所决定；
> 2. 如果一个细胞周围有 3 个细胞为生，则该细胞为生，即该细胞若原先为死则转为生， 若原先为生则保持不变；
> 3. 如果一个细胞周围有 2 个细胞为生，则该细胞的生死状态保持不变；
> 4. 在其它情况下，该细胞为死，即该细胞若原先为生则转为死，若原先为死则保持不变。

要求：实现本游戏，语言不限，使用发布订阅模式或观察者模式。

使用观察者模式。在一般的观察者模式中，subject和observer通常是不同的，而本题中的“细胞”既是subject，又是observer，因此构建的Cell对象既有subject的方法和属性，又有observer的方法和属性。每个 `Cell` 对象表示一个细胞，调用 `Notify()` 方法来通知周围的细胞，调用 `Update()` 方法来更新自身的状态。

```
class CellBase {
protected:
    std::list<CellBase *> adj_cells_; // 邻接的8个细胞
    bool alive_; // true表示“生”，false表示“死”
public:
    virtual ~CellBase() {}
    virtual void Update() {}

    // 新增一个邻接的细胞
    void Attach(CellBase *cell) {
        adj_cells_.push_back(cell);
    }

    // 移除一个邻接的细胞
    void Detach(CellBase *cell) {
        adj_cells_.remove(cell);
    }

    // 通知邻接细胞更新状态
    void Notify() {
        for (auto &c : adj_cells_)
            c->Update();
    }

    // 设置状态
    void SetStatus(bool alive) {
        alive_ = alive;
    }

    // 返回状态
    bool GetStatus() {
        return alive_;
    }
};

class Cell : public CellBase {
protected:
    std::string id_;
public:
    Cell(std::string id) : id_(id) {}
    virtual ~Cell() {}

    // 根据周围细胞的状态来更新自身状态
    virtual void Update() {
        int count = 0;
        for (auto &c : adj_cells_)
            if (c->GetStatus())
                ++count;

        if (count == 3)
            alive_ = true;
        else if (count < 2 || count > 3)
            alive_ false;
    }
};
```