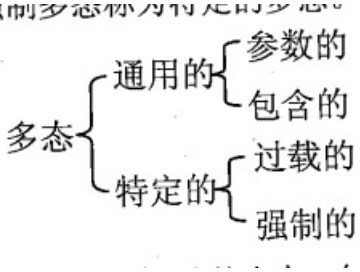
面向对象技术

面向对象基础

面向对象基本概念

- 对象
- 消息
- 类
- 继承
- 多态
- 动态绑定

多态有4种.参数多态和包含多态称为通用多态.过载多态和强制多态称为特定的多态



- 参数多态是应用最广的最纯的多态.
- 包含多态最常见的例子是子类化.
- 过载多态是同一个名字在不同的上下文中代表的含义不同
- 类属类是一种参数多态机制.强调的是成员特性中和具体类型无关的部分.而和具体类型相关的部分则用变元来表示.类属类对类库的建设提供了强有力的支持.
- 重置或者覆盖是在子类中重新定义父类的方法.是通过动态绑定的机制,使得子类在继承父类接口定义的前提下,用适合自己要求的实现取替换父类中的相应机制..在Java中,使用抽象方法来进行重置申明.通过方法查找实现重置方法体的动态绑定.

面向对象分析(OOA)

面向对象分析包含:

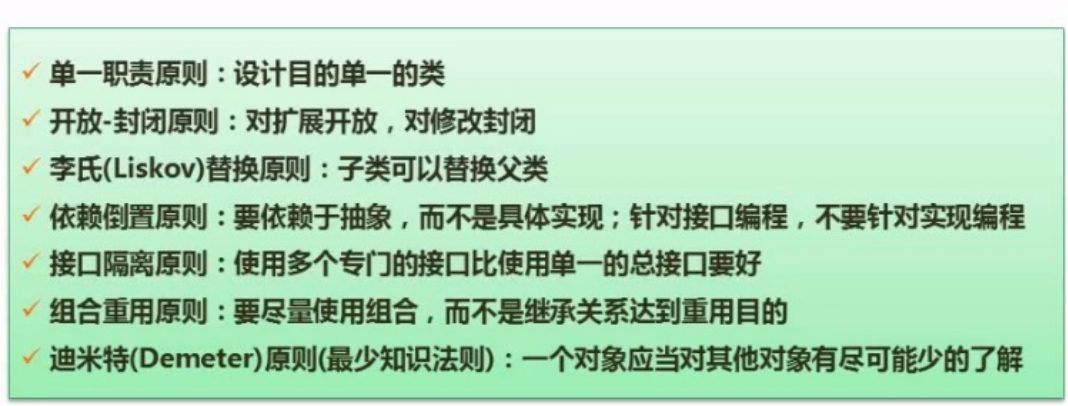
- 认定对象
- 组织对象
- 确认对象间的相互作用
- 基于对象的操作

面向对象设计(OOD Object-Oriented Design)

面向对象遵循的设计准则

- 抽象
- 信息隐蔽
- 功能独立
- 模块化

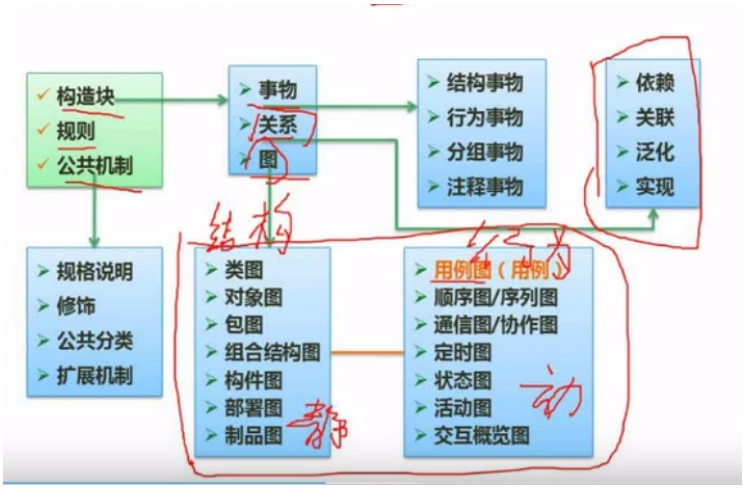
面向对象设计的原则



面向对象程序设计(OOP Objected-oriented Programming)

UML

考察频度很高, 考点是**UML中的关系和图**.其中,用例图考察比例最高



事务

- 结构事物 UML模型的静态部分,描绘概念或者物理元素,接口,用例,类等都属于结构事物
- 行为事物 UML模型的动态部分 状态,动作,消息.
- 分组事物 UML模型的组织部分.主要是包
- 注释事物 UML模型的注释部分

关系

- 依赖 箭头指向被依赖者
- 关联和聚集 关联一般是类和类之间的,比如雇主和雇员的关系.雇主是0到1个,雇员是0到*个.聚集是群体和个体的关系.这个个体属于群体,比如孩子和家庭的关系.菱形在群体侧. 实心的菱形则是组合.
- 范化 可以考虑是父类和子类的关系,箭头由子类指向父类
- 实现 可以想象是方法实现了接口,箭头由方法指向了接口

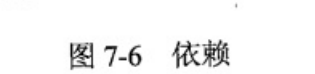


图 7-6 依赖

图 7-7 关联

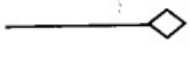


图 7-8 聚集

图 7-9　泛化

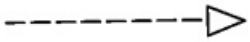


图 7-10　实现

图

图是一组元素的图形表示.一般都是顶点(代表事物)和弧(代表关系)的连通图.
UML2.0提供了13种图:

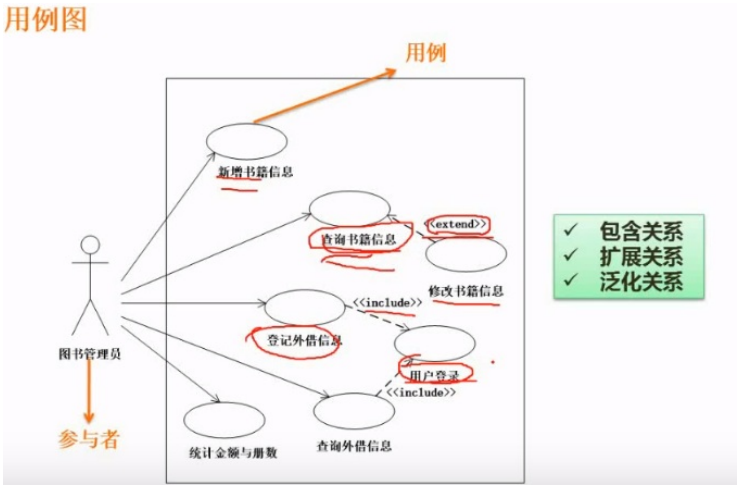
- 类图 静态 展示一组对象,接口,协作和它们之间的关系.
- 对象图 静态 描述某一时刻一组对象之间的关系
- 用例图 静态
- 序列图 动态 场景的图形化标识
- 通信图 动态 也称为协作图
- 状态图 动态
- 活动图 动态 本质上是一种特殊的状态图
- 构件图 静态 展示了构件之间的组织和依赖关系
- 组合结构图 静态
- 部署图 静态
- 包图 静态
- 交互概览图 动态
- 计时图 动态 适合实时和嵌入式系统

其中 序列图,通信图,交互概览图和计时图被称为交互图(对系统的动态方面进行建模的图称为交互图)

用例图

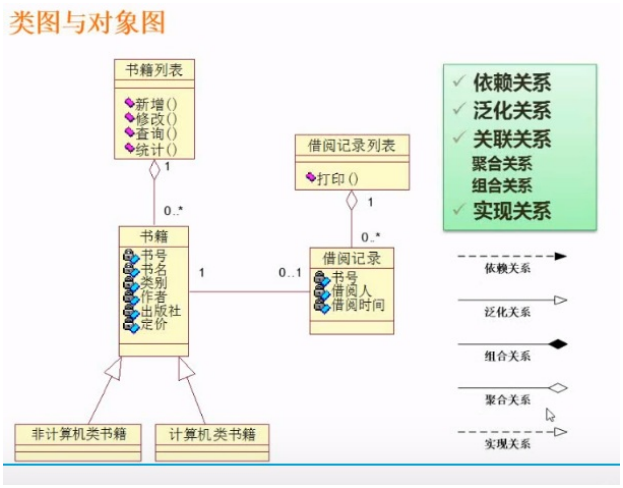
主要考察内容:

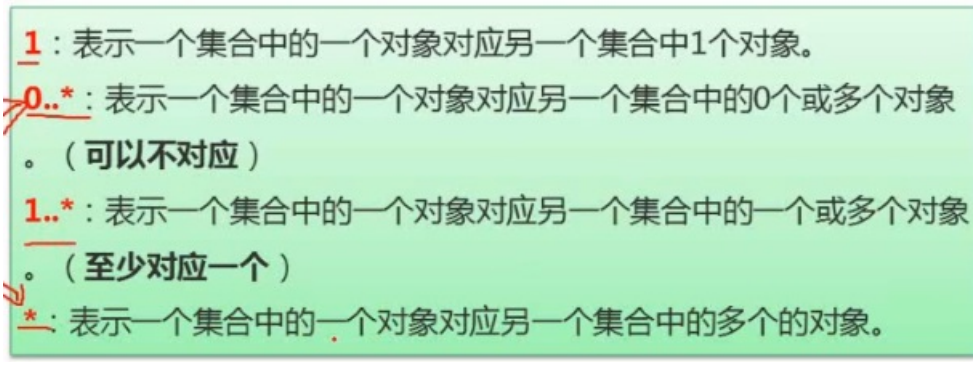
- 根据题干确认用例名称
- 用例之间的关系 extend/范化 include/包含



类图

主要考察类名,多重度.关系.类属性和方法

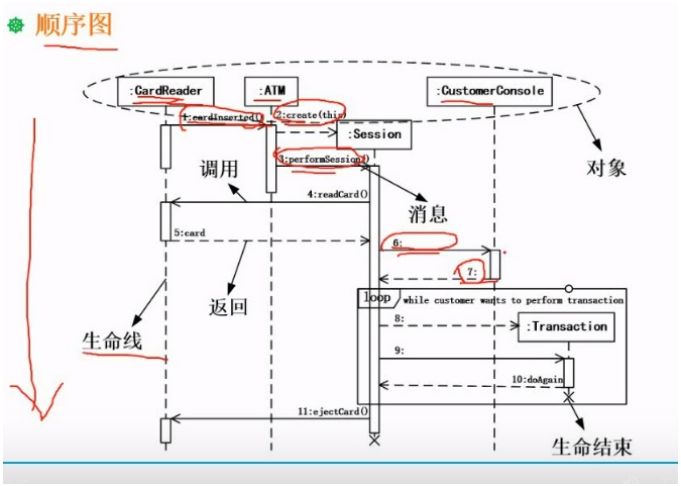




顺序图

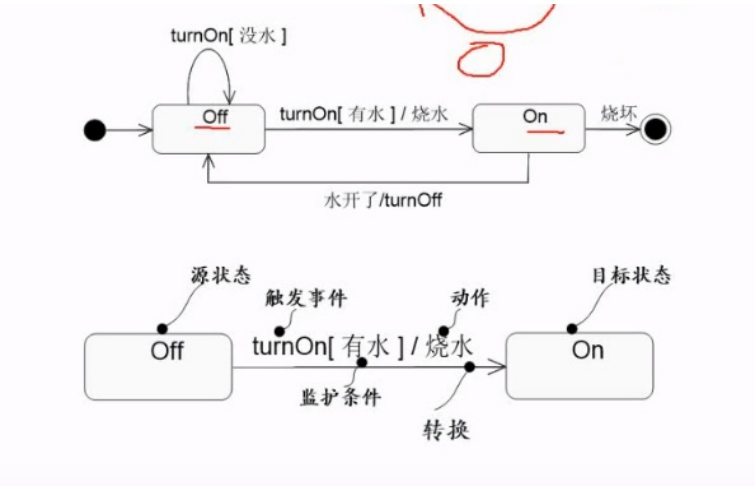
顺序图一般写在顶端,每一个对象引出一个生命线,用虚线表示.
序列图中用瘦高的矩形代表控制焦点.表示一个对象执行一个动作所需经历的时间段.它既可以是直接执行.也可以是通过下级程序执行.矩形的顶点表示动作的开始,底部表示动作的结束.
一般按照从上到下,从左至右(发起交互的对象在左边,下级对象放右边).按照箭头来执行.
箭头对应的是消息.
顺序图表达处理事务的时候的时间顺序.

顺序图主要考察的是消息.根据处理流程填上消息的名称.



状态图

状态图展现了一个状态机.由状态,转换,事件和活动组成.
箭头线表述状态. 主要查看对象为状态变迁的事件.
状态图强调行为的事件顺序



活动图

箭头表示的是消息.

活动图是特殊的状态图,展现的是一个活动到另一个活动的流程.
活动图专注于系统的动态试图.强调对象间的控制流程.

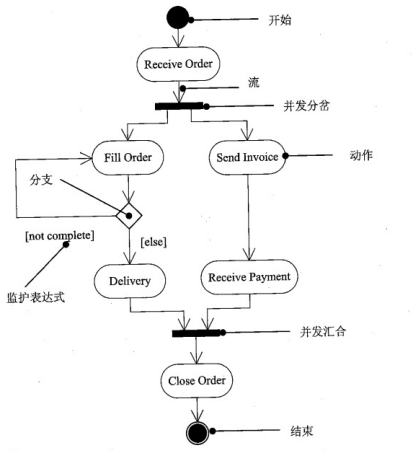


图 7-19 UML 活动图

通信图

也叫协作图.强调收发消息的对象的结构组织.顶点是参加交互的对象.传递的是消息.
通信图和序列图/顺序图是同构的,可以互相转换.

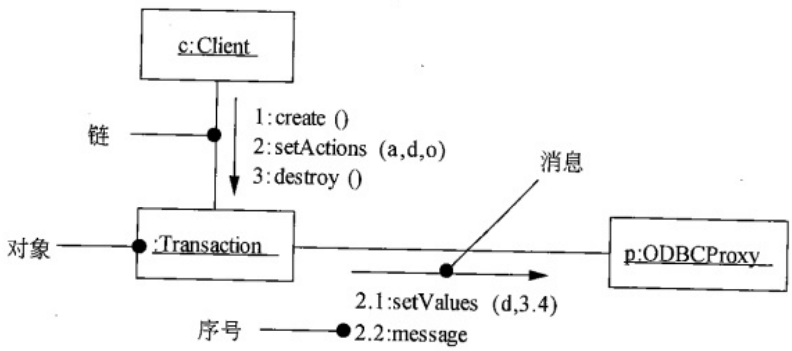


图 7-15 UML 通信图

例题

例题1

试题1

已知某唱片播放器不仅可以播放唱片，而且可以连接电脑并把电脑中的歌曲刻录到唱片上（同步歌曲）。连接电脑的过程中还可自动完成充电。

关于唱片，还有以下描述信息：

（1）每首歌曲的描述信息包括：歌曲的名字、谱写这首歌曲的艺术家以及演奏这首歌曲的艺术家。只有两首歌曲的这三部分信息完全相同时，才认为它们是同一首歌曲。艺术家可能是一名歌手或一支由2名或2名以上的歌手所组成的乐队。一名歌手可以不属于任何乐队，也可以属于一个或多个乐队。

（2）每张唱片由多条音轨构成；一条音轨中只包含一首歌曲或为空，一首歌曲可分布在多条音轨上；同一首歌曲在一张唱片中最多只能出现一次。

（3）每条音轨都有一个开始位置和持续时间。一张唱片上音轨的次序是非常重要的，因此对于任意一条音轨，播放器需要准确地知道，它的下一条音轨和上一条音轨是什么（如果存在的话）。

根据上述描述，采用面向对象方法对其进行分析与设计，得到了如表13-1所示的类列表、如图13-1所示的初始类图以及如图13-2所示的描述播放器行为的UML状态图。

试题1

【问题1】

根据题目中的描述，使用表13-1给出的类的名称，给出图13-1中的A~F所对应的类。

【问题2】

根据题目中的描述，给出图13-1中（1）~（6）处的多重度。

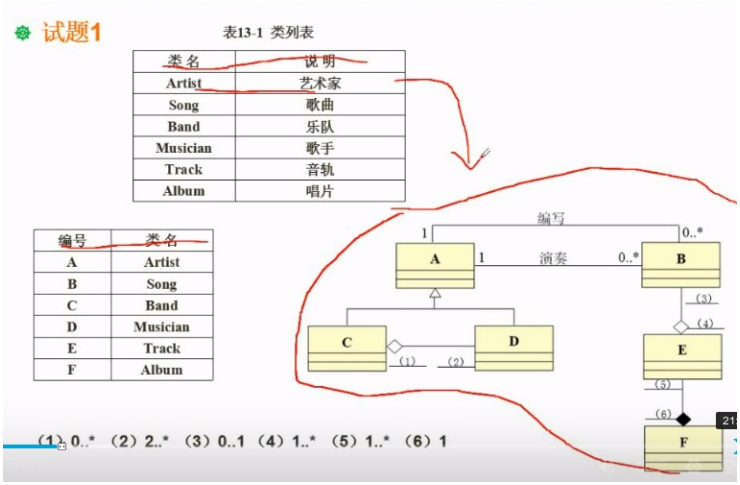
【问题3】

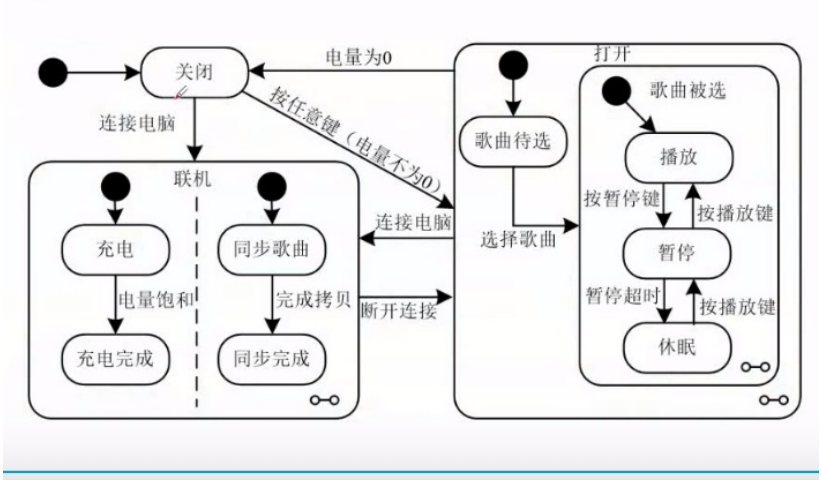
图13-1中缺少了一条关联，请指出这条关联两端所对应的类以及每一端的多重度。

类	多重度

【问题4】

根据图13-2所示的播放器行为UML状态图，给出从“关闭”状态到“播放”状态所经过的最短事件序列（假设电池一开始就是有电的）。





解题

*通过题干,我们可知的类包括:

- 歌曲(联合主键 名字,作曲和演奏).
- 艺术家, 歌手,艺术家和歌手是多对一的关系.歌手组成乐队,艺术家可以是歌手也可以是乐队.
- 唱片和音轨是一对多. 音轨和歌曲是多对一. 唱片和歌曲是一对多.
- 音轨后开始时间和持续时间,有指向上一个音轨和下一个音轨的指针
  注意图中的继承/范化,聚合和组合关系.进行分析

第一问

- A和B之间有1对多的编写和演奏的关系.可以肯定A为艺术家.B为歌曲.
- C和D是A的子类.C是D的聚合.我们通过题干分析可以知道.艺术家可以是歌手也可以是乐队.者说明歌手和乐队和艺术家有着范化的关系.艺术家是父类.歌手和乐队是子类.又因为乐队是歌手组成.是聚合关系.那么C就是乐队,D就是歌手.
- B和E有聚合关系,根据一首歌可以分布在多条音轨上,说明E是音轨.E和F有组合关系,说明F是唱片.

第二问

多重度的计算是站在对方的角度看待自己.计算B的多重度的时候就一个A对应几个B(是对应关系,不是一个A由几个B组成!)

- 由算计C的多重度.因为一个歌手对应0到n个乐队,所以1的值是0..*, 计算D的多重度,因为一个乐队至少需要2个歌手组成,歌手可以属于多个乐队.所以D的多重度(2)就是2..*
- 因为一个歌曲分布在多个音轨上.所以E的多重度(4)就是1..*,由于一个音轨最多保存一个歌曲或者空.所以B的多重度(3)的值是0..1
- 由于一个唱片对应多个音轨(至少一个音轨),所以5的值是1..*, 6的值是1.

第三问

题干3的部分没有在图中描述. 音轨有对应上一条和上一条(最少1条音轨).所以这条关键的类就是音轨.多重度就是0,1.

第四问 关闭,按任意键,选择歌曲,播放.

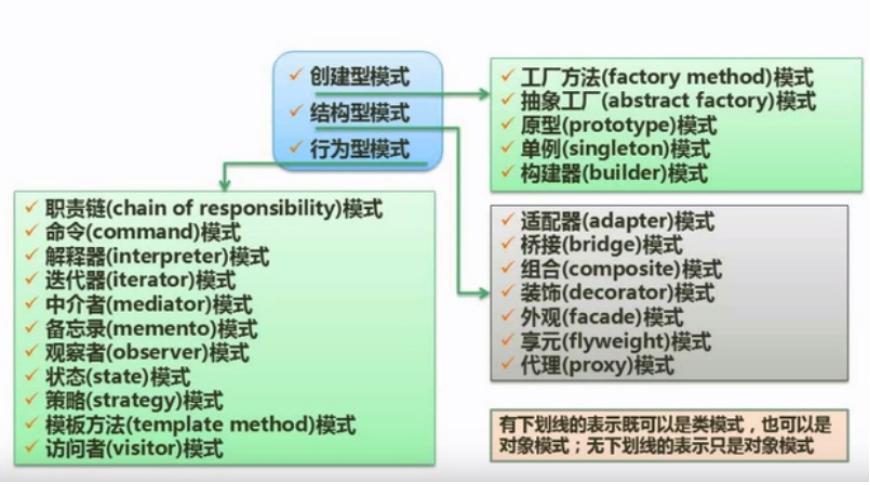
这类题的描述都是不完整的.做这类题的时候注意不要较真.模棱两可的事情假设成立,往简单的地方想.

设计模式

设计模式4要素:

- 模式名称 助记
- 问题 描述了在何时使用此模式
- 解决方案 描述了设计模式的组成,内部关系.协作和职责.
- 效果 描述了模式应用的结果.

设计模式分为 **创建型,结构型和行为型**三大类



看教材书仔细了解不同的设计模式的特点

创建型设计模式

设计模式名称	简要说明
Abstract Factory 抽象工厂模式	提供一个接口，可以创建一系列相关或相互依赖的对象，而无需指定它们具体的类
Builder 构建器模式	将一个复杂类的表示与其构造相分离，使得相同的构建过程能够得出不同的表示
Factory Method 工厂方法模式	定义一个创建对象的接口，但由子类决定需要实例化哪一个类。工厂方法使得子类实例化的过程推迟
Prototype 原型模式	用原型实例指定创建对象的类型，并且通过拷贝这个原型来创建新的对象
Singleton 单例模式	保证一个类只有一个实例，并提供一个访问它的全局访问点

结构型设计模式

设计模式名称	简要说明	速记关键字
Adapter 适配器模式	将一个类的接口转换成用户希望得到的另一种接口。它使原本不相容的接口得以协同工作	转换接口
Bridge 桥接模式	将类的抽象部分和它的实现部分分离开来，使它们可以独立地变化	继承树拆分
Composite 组合模式	将对象组合成树型结构以表示“整体-部分”的层次结构，使得用户对单个对象和组合对象的使用具有一致性	树形目录结构
Decorator 装饰模式	动态地给一个对象添加一些额外的职责。它提供了用子类扩展功能的一个灵活的替代，比派生一个子类更加灵活	附加职责
Façade 外观模式	定义一个高层接口，为子系统中的一组接口提供一个一致的外观，从而简化了该子系统的使用	对外统一接口
Flyweight 享元模式	提供支持大量细粒度对象共享的有效方法	
Proxy 代理模式	为其他对象提供一种代理以控制这个对象的访问	

行为型设计模式

设计模式名称	简要说明	速记关键字
Chain of Responsibility 职责链模式	通过给多个对象处理请求的机会，减少请求的发送者与接收者之间的耦合。将接收对象链接起来，在链中传递请求，直到有一个对象处理这个请求	传递职责
Command 命令模式	将一个请求封装为一个对象，从而可用不同的请求对客户进行参数化，将请求排队或记录请求日志，支持可撤销的操作	日志记录，可撤销
Interpreter 解释器模式	给定一种语言，定义它的文法表示，并定义一个解释器，该解释器用来根据文法表示来解释语言中的句子	
Iterator 迭代器模式	提供一种方法来顺序访问一个聚合对象中的各个元素，而不需要暴露该对象的内部表示	
Mediator 中介者模式	用一个中介对象来封装一系列的对象交互。它使各对象不需要显式地相互调用，从而达到低耦合，还可以独立地改变对象间的交互	不直接引用

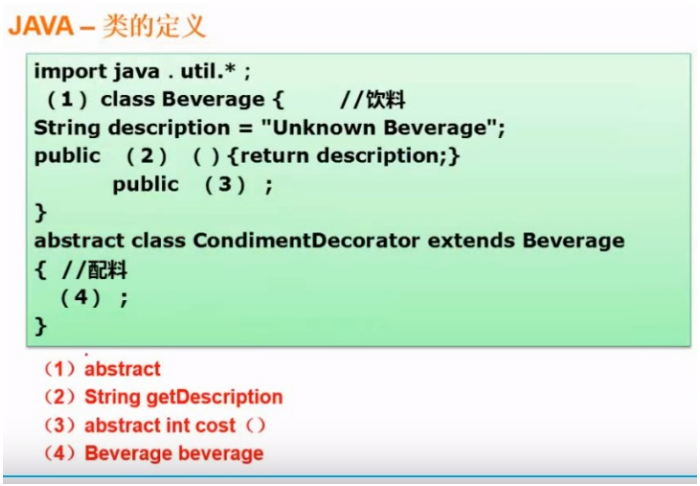
设计模式名称	简要说明	速记关键字
Memento 备忘录模式	在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，从而可以在以后将该对象恢复到原先保存的状态	
Observer 观察者模式	定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动更新	
State 状态模式	允许一个对象在其内部状态改变时改变它的行为	状态变成类
Strategy 策略模式	定义一系列算法，把它们一个个封装起来，并且使它们之间可互相替换，从而让算法可以独立于使用它的用户而变化	多方案切换
Template Method 模板方法模式	定义一个操作中的算法骨架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构即可重新定义算法的某些特定步骤	
Visitor 访问者模式	表示一个作用于某对象结构中的各元素的操作，使得在不改变各元素的类的前提下定义作用于这些元素的新操作	

面向对象之Java

知识点和简单填空题

JAVA – 类的定义

类的定义格式如下：
[import包]
[类修饰符] class xxxclass [extends超类] [implements 接口] {
    public:
        公有数据成员或公有函数成员的定义；
    protected:
        保护数据成员或保护函数成员的定义；
    private:
        私有数据成员或私有函数成员的定义；
}
说明：
·import包：引入包中的类。
·类修饰符：主要有四个修饰符，public、abstract、final、private。
·class为关键字，xxxclass为类名，命名遵循Java标识符的命名规则。
·extends为继承关键字，implements 为接口关键字。



基本定义方法

定义接口

- 接口是隐式抽象的，当声明一个接口的时候，不必使用abstract关键字。
- 接口中每一个方法也是隐式抽象的，声明时同样不需要abstract关键字。
- 接口中的方法都是公有的.

类实现接口

- 类中实现接口时，方法的名字、返回值类型、参数的个数及类型必须与接口中的完全一致.
- 一个类,如果实现了某个接口,必须要实现接口中的所有方法

抽象类

- 声明抽象类只需在声明类的时候在关键字class前面加上abstract即可 public abstract class User{....}. 包含抽象的方法不是必须的.
- 反之如果一个类包含抽象方法，那么该类必须是抽象类。也就是必须在class前面加上abstract
- 子类必须实现父类的所有抽象方法，或者声明自身为抽象类。
- 抽象方法只是声明，不包含方法体，就是不给出方法的具体实现也就是方法的具体功能。
- 构造方法，类方法（用static修饰的方法）不能声明为抽象方法。

```
interface Int1(){
    /*
    接口中的方法默认是public的,接口中的方法都是abstract的.
    也就是说实际上是 public abstract viod method1();
    public和abstract可以省略
    */
    void method1();
```

```
    int method2(float num);
}

class MyClass implements Int1{
    /*
    一个类,如果实现了某个接口,必须要实现接口中的所有方法,
    类中实现接口时，方法的名字、返回值类型、参数的个数及类型必须与接口中的完全一致
    */
    private String name;
    private int age;
    public MyClass(String name, int age){
        /*类构造器*/
        this.name = name;
        this.age = age;
    }
    public void method1(){
        System.out.println("ok");
    }
    public int method2(float num){
        return Integer(num);
    }
}

public abstract class Parent{
    /*包含抽象方法的抽象类*/
    abstract String say();
}

public class Children extends Parent{
    /*实现父类方法*/
    public String say(){
        return "hello world!";
    }
}

/*父类子类和main方法的示范*/

abstract class Parent{
    public String name;
    public int age;

    public Parent(String name, int age){
        this.name = name;
        this.age = age;
    }

    abstract public void say();
    public abstract void speak();
}

class Children extends Parent{
    public Children(String name, int age){
        super(name,age);
    }
    public void say(){
        System.out.println("ok");
    }
    public void speak(){
        System.out.println("hello");
    }
}

public class Family{
    public static void main(String[] args){
        Children child = new Children("Jack", 12);
        child.say();
        child.speak();
    }
```

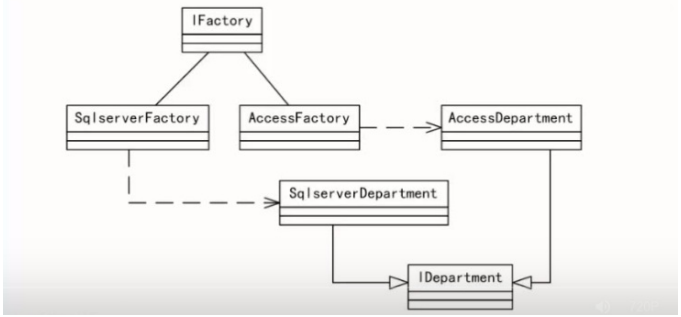
```
}
```

面向对象应用题

试题1

试题1

【说明】
现欲开发一个软件系统，要求能够同时支持多种不同的数据库，为此采用抽象工厂模式设计该系统。以SQL Server和Access两种数据库以及系统中的数据库表Department为例，其类图如图6-1所示。



【Java代码】

```
import java.util.*;
class Department{/*代码省略*/}

interface IDepartment{
   (1)   ;              (1) void Insert(Department department)
   (2)   ;              (2) Department GetDepartment(int id)
}

class SqlserverDepartment   (3)   {        (3) implements IDepartment
   public void Insert(Department department){
     System.out.println(" Insert a record into Department in SQL Server!");
     ||其余代码省略
   }
   public Department GetDepartment(int id){
     /*代码省略*/
   }
}
```

```
classAccessDepartment (4) {  implements IDepartment
   public void Insert(Department department){
     System.out.println("Insert a record into Department in ACCESS!" );
     ||其余代码省略
   }
   public Department GetDepartment(int id){
     /*代码省略*/
   }
}
   (5)   {          (5) interface IFactory
   (6)   ;          (6) Idepartment CreateDepartment()
}
```

```
class SqlServerFactory implements IFactory{
    public IDepartment CreateDepartment(){
      return new SqlserverDepartment();
    }
    //其余代码省略
}
class AccessFactory implements IFactory{
    public IDepartment CreateDepartment(){
      return new AccessDepartment();
    }
    //其余代码省略
}
```

试题2

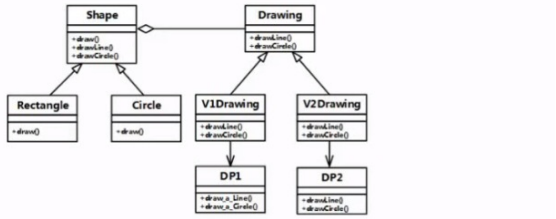
试题2

【说明】

欲开发一个绘图软件，要求使用不同的绘图程序绘制不同的图形。以绘制直线和圆形为例，对应的绘图程序如表6-1所示。

	DP1	DP2
绘制直线	draw_a_line(x1,y1,x2,y2)	drawline(x1,x2,y1,y2)
绘制圆	draw_a_circle(x,y,r)	drawcircle(x,y,r)

该绘图软件的扩展性要求，将不断扩充新的图形和新的绘图程序。为了避免出现类爆炸的情况，现采用桥接（Bridge）模式来实现上述要求，得到如图6-1所示的类图。




```
【Java代码】
  (1)  Drawing{        (1) interface
  (2)  ;               (2) void drawLine(double x1, double y1, double x2 ,double y2)
  (3)  ;               (3) void drawCircle (double x, double y, double r)
}
class  DP1{
static public void draw_a_line(double x1, double y1, double x2, double y2)
{/*代码省略*/}
static public void draw_a_circle(double x, double y, double r)
{/*代码省略*/}
}

class   DP2{
static public void drawline(double x1, double y1, double x2 ,double y2){/*代码省略*/}
static public void drawcircle (double x, double y, double r){/*代码省略*/}
}
```

```
class V1Drawing implements Drawing{
        public void drawLine(double x1, double y1, double x2 ,double y2){/*代码省略*/}
        public void drawCircle (double x, double y, double r){ ( 4 ) ;}
}                                                    DP1. draw_a_circle(x,y,r)

class V2Drawing implements Drawing{
        public void drawLine(double x1, double y1, double x2 ,double y2){/*代码省略*/}
        public void drawCircle (double x, double y, double r){ ( 5 ) ;}
}                                                    DP2. drawcircle(x,y,r)

Abstract class Shape{
        Private Drawing _dp;
        ( 6 );                              abstract public void draw()
        Shape(Drawing dp) {_dp=dp;}
        public void drawLine(double x1, double y1, double x2 ,double y2){
                _dp.drawLine(x1,y1,x2,y2);      }
        public void drawCircle (double x, double y, double r){ _dp.drawCircle(x,y,r);}
}
```

```
class Rectangle extends Shape{
        private double _x1,_x2,_y1,_y2;
        public Rectangle(Drawing dp,double x1, double y1, double x2 ,double y2)
                {/*代码省略*/}
        public void draw(){/*代码省略*/}
}

class Circle extends Shape{
        private double _x, _y, _r;
        public Circle(Drawing dp,double x, double y, double r) {/*代码省略*/}
        public void draw(){drawCircle(_x,_y,_r);}
}
```

应用题一般不会让你真的写代码.记得查看其他的代码和UML的类图.