第 6 章：程序设计与设计模式　　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

考情分析

本章主要围绕软件设计师大纲进行C++、Java语言程序设计和设计模式进行讲解，其中设计模式和软件设计师历年的真题的UML结构图全部采用Visio 2007绘制。本章知识全面，对C++、Java语言和设计模式知识进行了精炼的总结与概括，并对典型真题进行解析，考前必练选取了2007-2010年软件设计师历年的真题进行讲解与分析，是参加软件设计师考试的

C++、Java语言程序设计往往与设计模式结合出现在下午的考题中，分。根据考试大纲要求，要求考生掌握C语言和C++、Java中的一种面向对象的程序语言。题有四道题，选做题有三道题，分别为C语言题、C++题和Java题，整个卷面共七道题。许多考生在选做题中只选C语言题应答，对不会或不熟C++和Java语言的考生来说，一样可以应考，这与软件设计师考试大纲的要求是相背离的。因此，2010年上半年软件设计师下午试题中，必做题数量不变，选做题里去掉了C语言题，即选做题为二道题，整个卷面共六道题。这样，考生在选做题中就只能在C++和Java中选择一题作答，达到了考试大纲所要求的掌握一门面向对象的程序设计语言目的。在今后的软件设计师考试中，这种倾向将会维持一段时间。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

第 6 章：程序设计与设计模式　　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

考试大纲要求分析

本小节对考试大纲要求对Java程序设计和c++程序设计中容易涉及的一些知识进行了分析。

1．Java语言中流与文件的应用

对应主要考点如下：

Java I/O系统的类很多, Java中每一种流的基本功能依赖于InputStream and OutputStream，InputStream 用于读，OutputStream 用写。这部分的难点就是类比较复杂，尤其是每个类的构造方式。类的继承关系如下：

a．字节流：

InputStream

|-- FileInputStream （基本文件流）

|-- BufferedInputStream

|-- DataInputStream

|-- ObjectInputStream

OutputStream 同上图

BufferedInputStream DataInputStream ObjectInputStream 只是在 FileInputStream 上增

添了相应的功能，构造时先构造FileInputStream。

b．字符流：

Reader

|-- InputStreamReader（byte->char 桥梁）

|-- BufferedReader（常用）

Writer

|-- OutputStreamWriter（char->byte 桥梁）

|-- BufferedWriter

|-- PrintWriter（常用）

c. 随机存取文件 RandomAccessFile

2．Java语言中的多线程的开发与应用

对应主要考点如下：

① Java多线程

Java语言已经内置了多线程支持，所有实现Runnable接口的类都可被启动一个新线程，Thread类是实现了Runnable接口的一个实例，它代表一个线程的实例，并且启动线程的唯一方法就是通过Thread类实例的start( )方法。因此，有两个方法可以实现自己的线程：

方法1：自己的类继承Thread，并复写run( )方法，就可以启动新线程并执行自己定义的run( )方法。比如：

```
public class MyThread extends Thread {
   public run( ) {
     System.out.println( "MyThread.run( )" );
    }
}
```

在合适的地方启动线程：new MyThread( ).start( );

方法2：如果自己的类已经继承了另一个类，就无法直接继承Thread，此时，必须实现一个Runnable接口：

```
public class MyThread extends OtherClass implements Runnable {
   public run( ) {
     System.out.println( "MyThread.run( )" );
   }
}
```

为了启动MyThread，需要首先实例化一个Thread，并传入自己的MyThread实例：

```
MyThread myt = new MyThread( );
Thread t = new Thread( myt );
t.start( );
```

事实上，当传入一个Runnable target参数给Thread后，Thread的run( )方法就会调用target.run( )方法。

② 线程同步

由于同一进程内的多个线程共享内存空间，在Java中，就是共享实例，当多个线程试图同时修

改某个实例的内容时，就会造成冲突，因此，线程必须实现共享互斥，使多线程同步。最简单的同步是将一个方法标记为synchronized，对同一个实例来说，任一时刻只能有一个synchronized方法在执行。当一个方法正在执行某个synchronized方法时，其他线程如果想要执行这个实例的任意一个synchronized方法，都必须等待当前执行synchronized 方法的线程退出此方法后，才能依次执行。

但是，非synchronized方法不受影响，不管当前有没有执行synchronized方法，非synchronized方法都可以被多个线程同时执行。另外必须注意同一时刻，同一实例的synchronized方法只能被一个线程执行，不同实例的synchronized方法是可以并发的。比如class A定义了synchronized方法sync( )，则不同实例a1.sync( )和a2.sync( )可以同时由两个线程来执行。

3．Java语言中集合类库的应用

对应主要考点如下：

Collection是集合类的上级接口，继承与他的接口主要有Set 和List.Collections是针对集合类的一个帮助类，他提供一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。

map：映射,存储一系列名称与值的集合,数据没有顺序,不允许重复值,实现类:Hashtable

collection包括set接口和list接口。

set接口：一系列数据集合，（无序,不允许重复值,实现类:HashSet）。

List接口：一系列数据组成的列表（有序,允许重复值,实现类:数组,ArrayList,Vector）

下面是Java主要集合类的用法

① Hashtable

初始化：Hashtable table = new Hashtable( )；

添加数据：table.put(object类型的名称,object类型的数据)；

读取数据：先获得名称的枚举，再根据名称获得具体值。

```
Enumeration keys = table.keys( );
while( keys.hasMoreElements( ) )
{
    String name =( String )keys.nextElement( );
    String value = table.get( name );
}
```

删除数据：根据名称删除数据，table.remove(object key)；如果要删除所有数据，有清空操作(table.clear( ))，但是推荐重新new(初始化)。

② HashSet

初始化：HashSet set = new HashSet( );

添加数据：set.add(object类型的值);

读取数据：根据迭代器获得所有数据。

```
Iterator it = set.iterator( );
while( it.hasNext( ) )
{
    String ss = ( String )it.next( );
}
```

删除数据：set.remove(数据)；如果删除所有，推荐重新new(初始化)。

4．Java语言中网络、数据库的开发与应用

对应主要考点如下：

JDBC(Java Database Connectivity)是Java 实现数据库访问的API(Application Programming Interface)，与Microsoft 的ODBC(Open Database Connectivity)一样。JDBC API被定义在java.sql包中，其中定义了JDBC API用到的所有类、接口和方法，主要的类和接口有：

• DriverManager类——处理驱动程序的装入，为新的数据库连接提供支持。驱动程序要向该类注册后才能被使用。进行连接时，该类根据JDBC URL选择匹配的驱动程序。

• java.sql.Driver接口——驱动程序接口，负责确认URL与驱动程序的匹配、建立到数据库的连接等，其中的方法需要有相应的驱动程序实现。

• java.sql.Connection接口——表示到特定数据库的连接，其中的方法需要有相应的驱动程序实现。

• java.sql.Statement接口——为SQL语句提供一个容器，包括执行SQL语句、取得查询结果等方法。此接口有两个子类型：

①java.sql.PreparedStatement，用于执行预编译的SQL语句；

②java.sql.CallableStatement，用于执行对一个数据库内嵌过程的调用。

• java.sql.ResultSet接口——提供对结果集进行处理的手段。

一般来说，JDBC 的工作主要分为3个步骤：首先与某一关系数据库建立连接；然后向数据库发送SQL 语句，实现对数据库的操作；最后取得处理结果。

5．C++标准类库中容器库的应用

对应主要考点如下：

C++的标准模板库（Standard Template Library，简称STL）是一个容器和算法的类库。容器往往包含同一类型的数据。STL中比较常用的容器是vector，set和map，比较常用的算法有Sort等。下面以vector为例来讲解

① 声明：

一个vector类似于一个动态的一维数组。

vector<int> a; //声明一个元素为int类型的vector a

vector<MyType> a; //声明一个元素为MyType类型的vector a

这里声明的a包含0个元素，既a.size( )的值为0，但它是动态的，其大小会随着数据的插入和删除而改变。

vector<int> a(100, 0); //这里声明的是一已经存放了100个0的整数vector a

② 向量操作

常用函数：

```
size_t size( );            // 返回vector的大小，即包含的元素个数
void pop_back( );          // 删除vector末尾的元素，vector大小相应减一
void push_back( );         //用于在vector的末尾添加元素
T back( );                 // 返回vector末尾的元素
void clear( );             // 将vector清空，vector大小变为0
```

初始化：Vector vc = new Vector( );

添加数据：vc.add(object类型value);vc.add(int index,value);索引从0开始。

读取数据：for循环

```
for( int i = 0;i<vc.size( );i++ )
{
    vc.get( i );
}
```

迭代器

```
Iterator it = vc.iterator( );
while( it.hasNext( ))
{
    it.next( );
}
```

推荐使用枚举

```
Enumeration keys = vc.elements( );
while( keys.hasMoreElements( ))
{
    keys.nextElement( );
}
```

删除数据：vc.remove( );删除所有推荐重新new(初始化)。

6．C++标准类库中算法库的应用

对应主要考点如下：

算法部分主要由头文件<algorithm>，<numeric>和<functional>组成。

① <algorithm>是所有STL头文件中最大的一个，它是由一大堆模版函数组成的，可以认为每个函数在很大程度上都是独立的，其中常用到的功能范围涉及到比较、交换、查找、遍历操作、复制、修改、移除、反转、排序、合并等等。

②<numeric>体积很小，只包括几个在序列上面进行简单数学运算的模板函数，包括加法和乘法在序列上的一些操作。

③<functional>中则定义了一些模板类，用以声明函数对象。

STL的算法也是非常优秀的，它们大部分都是类属的，基本上都用到了C + +的模板来实现，这样，很多相似的函数就不用自己写了，只要用函数模板就可以了。

7．C++标准类库中的迭代器的应用

对应主要考点如下：

除了使用下标来访问vector对象的元素外，标准库还提供了另一种检测元素的方法：使用迭代器（iterator）。迭代器是一种允许程式员检查容器内元素，并实现元素遍历的数据类型。容器的iterator类型每种容器类型都定义了自己的迭代器类型。

① vector：

```
vector<int>::iterator iter;
```

这条语句定义了一个名为iter的变量，他的数据类型是由vector<int>定义的iterator类型。每个标准库容器类型都定义了一个名为iterator的成员，这里的iterator和迭代器实际类型的含义相同。

② begin和end操作

每种容器都定义了一对命名为begin和end的函数，用于返回迭代器。假如容器中有元素的话，由begin返回的迭代器指向第一个元素：

vector<int>::iterator iter = ivec.begin( );

上述语句把iter初始化为由名为begin的vector操作返回的值。假设vector不空，初始化后，iter即指该元素为ivec[0]。由end操作返回的迭代器指向vector的“末端元素的下一个”。通常称为超出末端迭代器（off-the-end iterator），表明他指向了一个不存在的元素。假如vector为空，begin返回的迭代器和end返回的迭代器相同。由end操作返回的迭代器并不指向vector中任何实际的元素，相反，他只是起一个哨兵（sentinel）的作用，表示我们已处理完vector中任何元素。例如：

```
std::vector<int> IntVector;
std::vector<int>::iterator first = IntVector.begin( );
//first指向向量第一个元素，*first即为第一个元素的值
std::vector<int>::iterator last = IntVector.end( );
//end指向向量最后一个元素，*end即为最后一个元素的值
```

8．C++标准类库中的字符串的应用

对应主要考点如下：

C++标准程序库中的string类

String类的主要构造函数和析构函数如下：

a）string s; //生成一个空字符串s

b）string s(str) //拷贝构造函数 生成str的复制品

c）string s(str,stridx) //将字符串str内"始于位置stridx"的部分当作字符串的初值

d）string s(str,stridx,strlen) //将字符串str内"始于stridx且长度顶多为strlen"的部分作为字符串的初值

字符串操作函数列举部分如下

a）=,assign( ) //赋以新值

b）swap( ) //交换两个字符串的内容

c）+=,append( ),push_back( ) //在尾部添加字符

d）insert( ) //插入字符

e）erase( ) //删除字符

f）clear( ) //删除全部字符

g）replace( ) //替换字符

h）+ //串联字符串

i）= =,!=,<,<=,>,>=,compare( ) //比较字符串

j）size( ),length( ) //返回字符数量

k）max_size( ) //返回字符的可能最大个数

l）empty( ) //判断字符串是否为空

9．C++标准类库中的流与文件的应用

对应主要考点如下：

在C++中用printf和scanf可以输出和输入标准类型（如:int，float，double，char）的数据，

但无法输出用户自己声明的类型（如数组、结构体、类）的数据。在C++中，会经常遇到对类对象的输入输出，显然无法使用printf和scanf来处理。在C++中，输入输出流被定义为类。C++的I/0库中的类称为流类（streamclass）。用流类定义的对象称为流对象。C++编译系统提供了用于输人输出的iostream类库，iostream类是从istream类和ostream类通过多重继承而派生的类。 cout和cin并不是C++语言中提供的语句，它们是iostream类的对象。此外C++对文件的输人输出需要用ifstream和ofstream类，ostream类定义了3个输出流对象，即cout，cerr，clog。分述如下。

①cout流对象

cout是console output的缩写，意为在控制台（终端显示器）的输出。

②cerr流对象

cerr流对象是标准出错流。cerr流已被指定为与显示器关联。cerr的作用是向标准出错设备（standard error device）输出有关出错信息。cerr是console error的缩写，意为“在控制台（显示器）显示出错信息”。cerr与标准输出流cout的作用和用法差不多。

③clog流对象

clog流对象也是标准出错流，它是console log的缩写。它的作用和cerr相同，都是在终端显示器上显示出错信息。它们之间只有一个微小的区别：ccrr是不经过缓冲区，直接向显示器上输出有关信息，而clog中的信息存放在缓冲区中，缓冲区满后或遇endl时向显示器输出。

10．用C++语言实现常见的设计模式及应用程序

对应主要考点如下：

主要结合23种设计模式中的一种，用C++程序实现。

11．用Java语言实现常见的设计模式及应用程序

对应主要考点如下：

主要结合23种设计模式中的一种，用Java程序实现。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

作者：希赛教育软考学院　来源：希赛网　2014年05月06日

命题特点与趋势分析

在今后的软件设计师考试中，C++、Java语言程序设计往往与设计模式结合，一般出现在下午的最后两道考题中，作为二选一试题，分值为15分。设计模式的考试范围一般就是本书所介绍的23种设计模式。经过分析可知每2年的设计模式考试一般不会重复，并且行为型模式和结构型模式出考题的几率要大于创建型模式，命题者喜欢进行交替考查。如上半年考结构型模式的话，下半年很可能会考行为型模式。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

考点精讲

软件设计师考试大纲中明确提出要求考生掌握面向对象开发概念（类、对象、属性、封装性、继承性、多态性、对象之间的引用）、面向对象开发方法的优越性以及有效领域、面向对象设计方法（体系结构、类的设计、用户接口设计）、面向对象实现方法（选择程序设计语言、类的实现、方法的实现、用户接口的实现、准备测试数据）、面向对象程序设计语言（如C++、Java）的基本机制。掌握设计模式的基本概念及其要素（创建型设计模式、结构型设计模式、行为型设计模式），同时在系统实施中也明确提出要求考生掌握C程序设计语言，以及C++、Java中任一种程序设计语言，以便能指导程序员进行编程和测试，并进行必要的优化，即结合设计模式与面向对象程序设计语言（如C++、Java）对考生进行综合测试。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

C++语法精要

本小节根据考试大纲要求对c++程序设计进行了精要的概括与总结。

1．从C到C++

C++语言是从C语言发展演变而来的，C++包含了整个C，C是建立C++的基础。C++包括C的全部特征、属性和优点，同时，C++添加了对面向对象编程的完全支持。

2．面向对象程序设计语言的概念以及C++面向对象机制

结构化程序设计语言是支持结构化方法的语言，这种语言中提供支持和实现三种基本结构（顺序、选择、循环）的机制，提供支持和实现模块的机制。面向对象程序设计语言

是支持面向对象程序设计方法的语言。从面向对象方法来看，面向对象程序设计语言应该是能提供支持实现面向对象方法中的概念、原则、对象的各种关系等机制的程序设计语言。

① 对象方法中的概念：对象、类、继承、封装、聚合、消息传送、多态性

② 原则：抽象、封装、继承、消息通信

③ 对象的各种关系：对象的分类关系、对象之间的组合关系、对象属性之间的静态关系、对象行为之间的动态关系。

④概念解释

•抽象性：抽象是指从具体的实例中抽取出共同的性质并加以描述的过程。

•封装性：所谓封装，就是将一个事物包装起来，使外界不了解它的详细内情。对象有效实现了封装的两个目标——对数据和行为的包装和信息隐藏。

•继承性：继承是软件复用的一种方式，通过继承，一个对象可以获得另一个对象的属性，并加入属于自己的一些特性。从另一个角度来看，从已有类产生新类的过是类的派生。已有的类称为基类或父类，产生的新类称为派生类或子类。

•多态性：在基类中定义的属性和操作被派生类继承之后，可能具有不同的数据类型或表现出不

同的行为，我们称之为多态性。也就是说，多态性表现为同一属性或操作在一般类及各特殊类中具有不同的语义。

•类：在面向对象的程序设计中，将彼此相关的数据和与这些数据相关的操作（函数）封装在一起，形成的一种新的具有更高的集成性和抽象性的数据类型。

3．C++ 的数据类型

① 基本类型：整型（int）、字符型char 、实型（float 、double）、bool。

② 特殊类型：空类型（void）

③ 用户定义类型：枚举类型（enum）

④ 构造类型：数组、结构和联合。

⑤ 指针类型：*

⑥ 抽象数据类型：类类型

4．输入输出流cin/cout

①cin:预定义的输入流

输入整数和实数

int i,j;

cin>>i>>j;

float x,y;

cin>>x>>y;

输入字符数据

char a,b,c;

cin>>a>>b>>c;

②cout:预定义的输出流

输出字符与字符串

cout<<”hello world!”

char ch = ’a’;

cout<<ch

char a = ’x’,b = ’y’;

cout<<”a = ”<<x<<”\n”<<”b = ”<<b;

输出整数和实数

int i = 2; float j = 12.3;

cout<<i<<j;

③namespace

C + + 标准程序库中的所有标识符都被定义于一个名为std的namespace中。由于namespace的概念，使用C + + 标准程序库的任何标识符时，可以有三种选择：

•直接指定标识符：

例如：std::ostream而不是ostream。完整语句如下： std::cout << std::hex << 3.4 << std::endl;

•使用using关键字：

例如：using std::cout; using std::endl; using std::cin; 以上程序可以写成 cout << std::hex

<< 3.4 << endl;

•最方便的就是使用using namespace std

例如：using namespace std;

这样命名空间std内定义的所有标识符都有效（曝光）。就好像它们被声明为全局变量一样。那么以上语句可以如下写: cout << hex << 3.4 << endl。

5．基本语句

① 表达式语句:表达式,函数调用。

② 空语句

③ 块语句

④ 选择语句：分支语句if、switch

⑤ 循环语句：for、while、do…while三种循环

⑥ break语句

⑦ continue语句

6．函数的定义和声明

①函数的定义

函数是一个命名的程序代码块，是程序完成其操作的场所，是将功能重复的程序段抽象出来所形成一个独立的、可重复使用的功能模块。

定义函数的一般格式为：

返回类型 函数名（数据类型1 参数1, 数据类型2 参数2,... ）

{

语句序列；

}

②C++中，函数原型声明原则

•如果函数定义在先，调用在后，调用前可以不必声明； 如果函数定义在后，调用在先，调用前必须声明。

•在程序设计中，为了使程序设计的逻辑结构清晰，一般将主要的函数放在程序的起始位置声明，这样也起到了列函数目录的作用。

声明函数原型的形式如下：

返回类型 函数名（数据类型1 参数1, 数据类型2 参数2,... ）；

例如：

int max(int x,int y) ;

int max(int,int) ;

③函数重载

各个重载函数的返回类型可以相同，也可以不同。但如果函数名相同、形参表也相同，仅仅是返回类型不同，则是非法的。在编译时会认为是语法错误。

7．类的基本问题

① 类的定义

简单讲，类是一个包含函数的结构体。因此，类的定义与结构类型的定义相似，其格式如下：

class 类名

{

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；

};

② 成员函数

成员函数描述的是类中的数据成员实施的操作。成员函数的定义、声明格式与非成员函数（全局函数）的格式相同。成员函数可以放在类中定义，也可以放在类外。放在类中定义的成员函数为内联（inline）函数。

C++可以在类内声明成员函数的原型，在类外定义函数体。这样做的好处是相当于在类内列了一个函数功能表，使我们对类的成员函数的功能一目了然，避免了在各个函数实现的大堆代码中查找函数的定义。在类中声明函数原型的方法与一般函数原型的声明一样，在类外定义函数体的格式如下：

返回值类型 类名 :: 成员函数名（形参表）

{

函数体；

}

::是类的作用域分辨符，用在此处，放在类名后成员函数前，表明后面的成员函数属于前面的那个类。

③ 对象的建立与使用

建立对象后，就可以通过对象存取对象中的数据成员及调用成员函数。存取语法如下：

对象名.属性

对象名.成员函数名（实参1, 实参2,…,）

④ 构造函数与析构函数

在定义一个对象的同时，希望能给它的数据成员赋初值¯对象的初始化。在特定对象使用结束时，还经常需要进行一些清理工作。C++程序中的初始化和清理工作分别由两个特殊的成员函数来完成，它们就是构造函数和析构函数。

▼构造函数（constructor）

构造函数是与类名相同的，在建立对象时自动调用的函数。如果在定义类时，没有为类定义构造函数，编译系统就生成一个默认形式的隐含的的构造函数，这个构造函数的函数体是空的，因此默认构造函数不具备任何功能。构造函数是类的一个成员函数，除了具有一般成员函数的特征之外，还归纳出如下特殊的性质：

•构造函数的函数名必须与定义它的类同名。

•构造函数没有返回值。如果在构造函数前加void是错误的。

•构造函数被声明定义为公有函数。

•构造函数在建立对象时由系统自动调用。

▼析构函数（destructor）

也译作拆构函数，是在对象消失之前的瞬间自动调用的函数，其形式是：

~构造函数名( );

析构函数具有以下特点：

•析构函数没有任何参数，不能被重载，但可以是虚函数，一个类只有一个析构函数。

•析构函数没有返回值。

•析构函数名与类名相同，但在类名前加上一个逻辑非运算符“~”，以示与构造函数对比区别。

•析构函数一般由用户自己定义，在对象消失时由系统自动调用，如果用户没有定义析构函数，系统将自动生成一个不做任何事的默认析构函数。

⑤拷贝构造函数

拷贝构造函数是与类名相同，形参是本类的对象的引用的函数，在用已存在对象初始化新建立对象时调用。

类的拷贝构造函数一般由用户定义，如果用户没有定义拷贝构造函数，系统就会自动生成一个默认函数，这个默认拷贝构造函数的功能是把初始值对象的每个数据成员的值依次复制到新建立的对象中。因此，也可以说是完成了同类对象的克隆（Clone）。这样得到的对象和原对象具有完全相同的数据成员，即完全相同的属性。

定义一个拷贝构造函数的一般形式为：

类名（类名& 对象名）

{

…

}；

注意：

•在重新定义拷贝构造函数后，默认拷贝构造函数与默认构造函数就不存在了，

•如果在此时调用默认构造函数就会出错。

•在重新定义构造函数后，默认构造函数就不存在了，但默认拷贝构造函数还存在。

•在对象进行赋值时，拷贝构造函数不被调用。此时进行的是结构式的拷贝。

⑥ 对象指针

对象如同一般变量，占用一块连续的内存区域，因此可以使用一个指向对象的指针来访问对象，即对象指针，它指向存放该对象的地址。可用类来定义对象指针变量，通过对象指针来访问对象的成员。

•对象指针语法定义

对象指针遵循一般指针变量的各种规则，其语法定义形式如下：

类名 *对象指针名；

如同通过对象名访问对象的成员一样，使用对象指针也只能访问该类的公有数据成员和函数成员，但与前者使用“.”运算符不同，对象指针采用“->”运算符访问公有数对象指针名->数据成员名，或：对象指针名->成员函数名（参数表）.

•对象指针应用

```
#include <iostream.h>
class M {
    int x,y;
public:
     M( ) { x = y = 0;}
     M( int i, int j ) { x = i; y = j;}
void Copy( M *m );
void SetXY( int i, int j ) { x = i;y = j;}
void Print( ) { cout<<"x="<<x<<",y="<<y<<endl; }
};
void M::Copy( M *m )
{ x = m->x; y = m->y; }
void fun( M m1, M *m2 )
{
  m1.SetXY( 12,15 ); m2->SetXY( 22,25 );
}
void main( )
{
  M p( 5,7 ),q;
  q.Copy( &p );
fun( p, &q );
p.Print( );
q.Print( );
}
```

程序运行结果为

x = 5,y = 7

x = 22,y = 25

⑦ 对象引用

对象引用就是对某类对象定义一个引用，其实质是通过将被引用对象的地址赋给引用对象，使二者指向同一内存空间，这样引用对象就成为了被引用对象的“别名”。

•对象引用的定义

对象引用的定义方法与基本数据类型变量引用的定义是一样的。定义一个对象引用，并同时指向一个对象的格式为：

类名 & 对象引用名=被引用对象；

•对象引用的使用格式为：

对象引用名.数据成员名 或：对象引用名.成员函数名（参数表）

•对象引用的应用

```
#include <iostream.h>
class M {
```

```
    int x,y;
public:
    M ( ) { x = y = 0;}
    M ( int i, int j ) { x = i; y = j;}
void Copy( M &m);
void SetXY( int i, int j ) { x = i;y = j;}
void Print( ) { cout<<"x = "<<x<<",y = "<<y<<endl; }
};
void M::Copy( M &m )
{ x = m.x; y = m.y; }
void fun( M m1, M &m2 )
{
  m1.SetXY( 12,15 ); m2.SetXY( 22,25 );
}
void main( )
{
  M p( 5,7 ),q;
  q.Copy( p );
fun( p, q );
p.Print( );
q.Print( );
}
```

程序运行结果为

```
x = 5,y = 7
x = 22,y = 25
```

⑧ 静态数据成员与静态成员函数

▼静态数据成员

静态数据成员是类的数据成员的一种特例，采用static关键字来定义，属于类属性，每个类只有一个拷贝，由该类的所有对象共同维护和使用，从而实现了同类的不同对象之间的数据共享。

▼静态成员函数

静态成员函数为类的全体对象而不是部分对象服务，与类相联系而不与类的对象联系，因此访问静态函数成员时，可以直接使用类名。

格式如下：

•通过类名调用静态成员函数

类名：：静态成员函数;

•通过对象调用静态成员函数，格式为：对象. 静态成员函数

8．派生类的概念和定义

① 定义格式

派生类定义的语法为：

class 派生类名：继承方式1 基类名1, 继承方式2 基类名2,…

{

private:

派生类的私有数据和函数

public:

派生类的公有数据和函数

protected:

派生类的保护数据和函数

};

说明：

•继承方式1 基类名1, 继承方式2 基类名2,…”为基类名表, 表示当前定义的派生类的各个基类。

•如果基类名表中只有一个基类，表示定义的是单继承；如果基类名表中有多个基类，表示定义的是多继承。

•继承方式指定了派生类成员以及类外对象对于从基类继承来的成员的访问权限。继承方式有三种：public：公有继承；private：私有继承；protected：保护继承。不管是哪种继承方式，基类的私有成员在派生类的作用域内不可见。派生类的成员函数不能直接访问基类的私有成员，但可直接访问基类的公有和保护成员，见图6-1。

•在派生类的定义中，每一种继承方式只限定紧跟其后的那个基类。如果不显式给出继承方式，系统默认为私有继承。

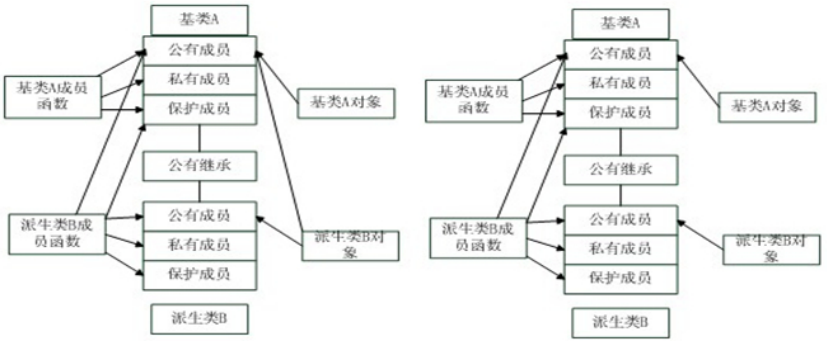

图6-1 公有继承与私有继承的访问方式

② 继承分类

•单一继承时的构造函数

派生类名::派生类名（基类所需的形参，本类成员所需的形参）:基类名（参数）

{

本类成员初始化赋值语句；

};

•多继承：允许派生类同时具有多个基类，基类和派生构成有向图的层次结构。多继承时的构造函数

派生类名::派生类名（基类1形参，基类2形参，...基类n形参，本类形参）:基类名1（参数），基类名2（参数），...基类名n（参数）

{

本类成员初始化赋值语句；

};

③ 二义性问题

一般来说，在派生类中对于基类成员的访问应该是唯一的，但是，由于多继承中派生类拥有多个基类，如果多个基类中拥有同名的成员，那么，派生类在继承各个基类的成员之后，当我们调用该派生类成员时，由于该成员标识符不唯一，出现二义性，编译器无法确定到底应该选择派生类中的哪一个成员，这种由于多继承而引起的对类的某个成员访问出现不唯一的情况就称为二义性问题。

▼二义性问题举例

```
class B
{   public:
        int b;
}
class B1 : public B
{
    private:
        int b1;
}
class B2 : public B
{
    private:
        int b2;
};
class C : public B1,public B2
{
    public:
        int f( );
    private:
        int d;
}
```

派生类C的对象的存储结构示意图见图6-2所示。

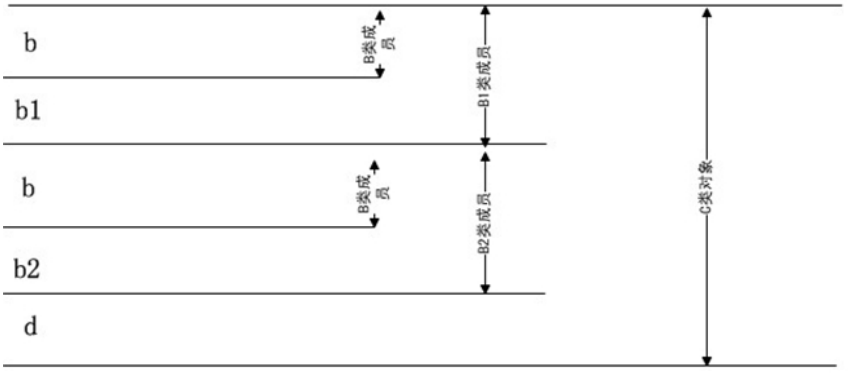

图6-2 派生类C的对象的存储结构

下面的访问是二义性的：

c.b

c.B::b

下面是正确的：

c.B1::b

c.B2::b

▼二义性的解决方法

•解决方法一：用类名来限定，通过类的作用域分辨符明确限定出现歧义的成员是继承自哪一个基类。

•解决方法二：同名覆盖，在派生类中新增一个与基类中成员相同的成员，由于同名覆盖，程序将自动选择派生类新增的成员。

④ 虚基类

为了解决从不同途径继承来的同名的数据成员在内存中有不同的拷贝造成数据不一致问题，将共同基类设置为虚基类。这时从不同的路径继承过来的同名数据成员在内存中就只有一个拷贝，同一个函数名也只有一个映射。这样不仅就解决了二义性问题，也节省了内存，避免了数据不一致的问题。

•虚基类的定义

虚基类的定义是在融合在派生类的定义过程中的，其定义格式如下：class 派生类名：virtual 继承方式 基类名

•虚基类举例

class B{ private: int b;};

class B1 : virtual public B { private: int b1;};

class B2 : virtual public B { private: int b2;};

class C : public B1, public B2{ private: float d;}

上面虚基类的派生类对象继承结构示意图见图6-3所示。

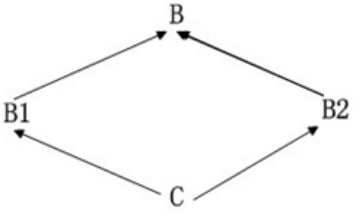

图6-3 派生类对象继承结构

派生类C的对象的存储结构示意图见图6-4所示。

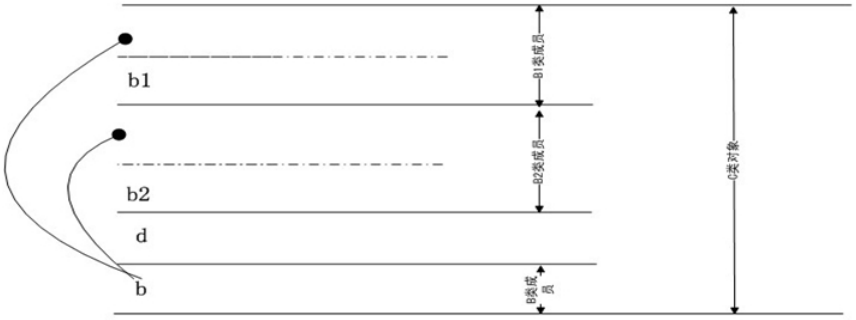

图6-4 派生类C的对象的存储结构

下面的访问是正确的：

C cobj;

cobj.b;

9．多态性与虚函数

① 多态性

多态性（polymorphism）是面向对象程序设计的重要特性之一。多态是指同样的消息被不同类型的对象接收时导致完全不同的行为。

② 虚函数的定义

虚函数定义的一般语法形式如下：

virtual 函数类型 函数表（形参表）

{

函数体；

}

10．类模板

类模板定义的语法为：

template <模板参数表>

class 类名

{

成员名；

};

其中：

• template为模板关键字。

•模板参数表中的类型为参数化（parameterized）类型，也称可变类型，类型名为class（或typename）；模板参数表中的类型也可包含普通类型，普通类型的参数用来为类的成员提供初值。

•类模板中的成员函数可以是函数模板，也可以是普通函数。

版权方授权希赛网发布，侵权必究

上一节 本书简介 下一节

作者：希赛教育软考学院 来源：希赛网 2014年05月06日

Java语法精要

本小节根据考试大纲要求对Java程序设计进行了精要的概括与总结。

1．Java程序的构成

Java语言的源程序代码由一个或多个编译单元组成，每个编译单元可包含三个要素：

① 一个包声明（package statement，可选）；

② 任意数量引入语句（import statements）；

③ 类的声明（class declarations）和接口声明（interface declarations）。

该三要素必须以上述顺序出现。也就是说任何引入语句出现在所有类定义之前；如果使用包声明，则包声明必须出现在类和引入语句之前。

2．关键字、标识符、数据类型、常量与变量

① 关键字：（具体见表6-2）

关键字对Java编译器有特殊的含义，它们可标识数据类型名或程序构造（construct）名。有关关键字值得我们注意的地方：

•true、false和null为小写，而不是象在C++语言中那样为大写。严格地讲，它们不是关键字，而是文字。然而，这种区别是理论上的。

•无sizeof运算符，因为所有数据类型的长度和表示是固定的，与平台无关，不是象在C语言中那样数据类型的长度根据不同的平台而变化。这正是Java语言的一大特点。

•goto和const不是Java编程语言中使用的关键字。

表 6-2 Java关键字

abstract	default	if	private	this
boolean	do	int	protected	throw
break	double	import	public	throws
byte	else	implements	package	try
case	extends	instanceof	return	transient
catch	final	interface	short	void
char	finally	native	strictfp	volatile
class	for	new	static	switch
continue	float	null	super	while
cast	false	long	synchronized	

② 标识符

在Java编程语言中，标识符是赋予变量、类或方法的名称。变量、函数、类和对象的名称都是标识符，程序员需要标识和使用的东西都需要标识符。标识符可从一个字母、下划线（_）或美元符号（$）开始，随后也可跟数字、字母、下划线或美元符号。标识符是区分大小写，没有长度限制，可以为标识符取任意长度的名字。标识符不能是关键字，但是它可以包含关键字作为它的名字的一部分。例如，thisone是一个有效标识符，但this却不是，因为this是一个Java关键字。

③ 数据类型

Java编程语言有八个原始数据类型，可分为以下四种：

•逻辑类（boolean）

boolean数据类型有两种文字值：true和false。

注意在Java编程语言中boolean类型只允许使用boolean值，在整数类型和boolean类型之间无转换计算。

在C语言中允许将数字值转换成逻辑值，这在Java编程语言中是不允许的。

•字符类（char）

•整数类（byte，short，int，long）

在Java编程语言中有四种整数类型，每种类型可使用关键字byte, short, int和long中的任意一个进行声明。所有Java编程语言中的整数类型都是带符号的数字，不存在无符号整数。整数类型的文字可使用十进制、八进制和十六进制表示。首位为“0”表示八进制的数值；首位为“0x”表示16进制的数值。

•浮点类（double，float）

在Java编程语言中有两种浮点类型：float和double。如果一个数包括小数点或指数部分，或者在数字后带有字母F或f（float）、D或d（double），则该数为浮点数。如果不明确指明浮点数的类型，浮点数缺省为double。

④ 常量与变量

在Java中，不同类型的数据既可以以常量的形式出现，也可以以变量的形式出现。常量就是指在程序执行期间其值不能发生变化的数据，常量是固定的。变量的值则是可以变化的，它的定义包括变量名、变量类型和作用域几个部分。

3．运算符

按照运算符功能来分，基本的运算符包括算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符、条件运算符等。

① 算术运算

算术运算符包括加号（+）、减号（-）、乘号（*）、除号（/）、取模（%）、自增运算符（++）、自减运算符（--）等。

② 关系运算

关系运算符用来比较两个值，返回布尔类型的值true或false。

③ 位运算

在Java语言中，位运算符有按位与运算符（&）、按位或运算符（|）、按位异或运算符（^）、按位取反运算符（~）、左移位运算符（<<）和右移位运算符（>>）。

④ 三目条件运算符（？：）

在Java语言中，三目条件运算符（？：）与C语言中的使用规则是完全一致的，使用的形式是：x？y：z;上面的三目条件运算的规则是：先计算表达式x的值，若x为真，则整个三目运算的结果是表达式y的值；若x为假，则整个三目运算的结果是表达式z的值。

⑤ 对象运算符（instanceof）

对象运算符instanceof用来判断一个对象是否是某一个类或者其子类的实例。如果对象是该类或者其子类的实例，返回ture；否则返回flase。

⑥ ‘.’运算符

‘.’运算符用于访问对象实例或者类的类成员函数。

⑦ new运算符

new运算符用于创建一个新的对象或者新的数组。

4．流程控制语句

① 选择结构

if—else语句,switch语句。

② 循环结构

循环结构是程序中一种重要的基本结构，是指在一定的条件下反复执行某段程序，被反复执行的这段程序称为“循环体”。Java中有三种语句来实现循环结构，分别是while，do-while和for语句。

③ 跳转语句

跳转语句用来实现循环执行过程中的流程转移。在switch语句中使用过的break语句就是一种跳转语句。在Java语言中，有两种跳转语句：break语句和continue语句。在Java语言中，可用break和continue控制循环的流程。其中，break用于强行退出循环，不执行循环中剩余的语句。而continue则停止执行当前的循环，开始新的循环。break语句和continue语句都有两种使用的形式：一种是不带标号的break语句和continue语句；一种是带标号的break语句和continue语句。标号应该定义在某一个循环语句之前，紧靠在循环语句的前方，用来标志这个循环结构，在标号和

循环之间置入任何语句都是不明智的行为。

5．类与对象

类与对象的概念与C++相同，见上。

① 类的实现

类作为复合数据类型，其实现包括两部分内容：类声明和类体。

格式如下：

[import包]

[类修饰符] class xxxclass [extends超类]〔implements 接口〕 //类声明

｛ //从这里开始为类体部分

// 类变量

// 实例变量

// 构造函数

// 类方法

// 实例方法

//类体部分结束

}

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。

•类体是类声明中用大括号所括起来的部分，它包括变量和方法，是类的主体部分。

② 类的修饰符

▼类修饰符：public

public提供给其他类完全的存取权限。也就是说在同一包中的类可自由取用此类，而别的包中的类可通过import关键词来引入此类所属的包加以运用。一个用public修饰符修饰的类具有以下几个特性：

•一个程序里只能有一个类被修饰为public，否则编译会出错。

•存储源文件时，必须是用public修饰的类名（xxxClass）来命名。

•若程序中没有任何public类，则文件名可任取。而如果文件名是程序中的一个类名，则该类被视为public。

▼类修饰符：final

final表示此类为“最终”类，别的类不能继承此类，其方法也不能被覆盖。例如System类即为final类。final类可被import来引用，但不能被继承。

▼类修饰符：private

private修饰的类只能被同一个包中的类来访问。

▼类修饰符：abstract

如果一个类用abstract来修饰，则此类为抽象类。

③ 成员变量与类变量

▼成员变量

成员变量是在类的内部、方法定义的外部所定义的变量，其作用域是整个类，也就是说，同一个类中的所有方法都可以对其访问。然而在方法内部定义的变量为局部变量，它的作用域仅仅只能在方法体内。成员变量定义的格式为：

[修饰符] 数据类型 变量名

如：public int x; //声明一个整型变量x，而且修饰符为public。其中，修饰符包括：public、protected、private、static、final。

▼类变量

用static修饰的变量称为类变量。

▼实例变量

没有用static修饰的变量称为实例变量。

▼最终变量

用final修饰的变量称为最终变量。最终变量一般在声明时进行初始化。一旦最终变量被赋值，则它在整个程序执行过程中不能改变。

④ 实例方法与类方法

类方法和实例方法统称为成员方法。

•实例方法、实例变量是指那些没有用static修饰的方法、变量。

•实例方法可以使用类中所有的变量与方法，也就是类中所有的成员，不管它们是否被static修饰。

•使用类方法有一点要特别注意的，那就是在类方法中只能使用类变量与类方法，也就是只能使用static修饰的变量与方法，而不能使用其余的实例变量。

•如果要在类方法中使用实例变量或者实例方法，就需要使用〔对象〕.〔数据〕的方式。这一点是写Java程序容易常犯的错误。

⑤ 构造方法

构造方法是一种特殊的方法。构造方法名必须与其类名相同，而且大小写都要一致。另外，构造方法没有返回值，不用void修饰。

构造方法用于创建对象时被调用。主要用于对对象内容做一些初值设置（初始化）。

在类的声明中，没有声明构造方法时，Java使用系统默认的构造方法。

构造函数的声明格式：

[修饰符] 类名（[参数行]）{

//构造函数主体（初始化操作）

}

⑥ 面向对象特性

▼封装性

所谓封装就是说类的设计者只是为使用者提供类对象可以访问的部分（包括类的成员变量和方法）。而对于类中其他成员变量和方法则隐藏起来，用户不能访问。Java为对象提供四种访问权限：public 、protected、private、friendly。这些修饰符的作用是对类的成员变量和成员方法施以一定的访问权限的限定，实现类中成员在一定范围内的信息隐藏，从而达到封装的目的。格式如下：

[public][protected][private][friendly]成员变量；

[public][protected][private][friendly]成员方法；

修饰符的作用比较（★表示可访问）见表6-4

表6-4：修饰符的作用比较

	同一个类	同一个包	不同包的子类	不同包非子类
private	★			
friendly	★	★		
protected	★	★	★	
public	★	★	★	★

类中不加任何范围权限限定的成员属于缺省的访问状态，这时候通常称此成员变量或方法为friendly。由表可知，在同一个类中、同一个包中的对象可以访问被friendly修饰的成员变量和调用friendly的成员方法。

▼继承性

继承的实现。通常在类的声明中加入关键字extends来创建一个类的子类，从而实现类的继承。格式如下：

[修饰符] class 子类名 [extends 父类名]{

…… //类体

}

在使用继承的特性时，我们需要注意下面的几个问题：

•子类可以继承父类的中访问权限设定为public、protected、friendly的成员变量和方法。

•当子类中和父类中的成员变量的名称相同的时候，子类会覆盖隐藏父类的成员变量。也就是说，子类在引用这个变量时，默认为是引用它自己定义的成员变量，而将从父类那里继承而来的成员变量”隐藏”。

•当子类中和父类中的方法的名称相同的时候，子类会覆盖隐藏父类的方法。也就是说，子类在引用调用方法时，默认为是引用它自己定义的方法，而将从父类那里继承而来的方法“隐藏”。所以在引用方法的时候需要指明引用的是父类的方法还是子类的方法。

▼多态性

多态性是由封装性和继承性引出的面向对象的程序设计的另一特性。在Java语言中多态性体现在两个方面：方法重载和方法重写（方法覆盖）。

•方法重载

所谓的方法重载，是指调用一个类中多个方法享有相同的方法名，但在执行时期可以根据其参数数量与类型来判断要调用此方法的哪一种操作。这些同名方法可按需要自行定义。

•方法重写（方法覆盖）

继承是指子类可以继承父类的方法，但是由于子类有时具有自己特有的特征，造成由父类继承而来的方法在子类中不能使用，这样Java允许子类对父类的同名方法进行重写，也就是子类与父类中已经定义的方法具有相同的名称，但是方法的内容不同，这种多态性称为方法重写（方法覆盖），即子类的方法“隐藏”父类的同名方法。

⑦ 抽象类

Java语言中，用关键字abstract修饰一个类时，这个类就是抽象类。用关键字abstract修饰一个方法时，这个方法就是抽象方法。抽象类是专门设计让子类来继承的，抽象方法必须被子类重写。对于抽象类，我们需要注意以下几个问题：

•一个抽象类里可以没有定义抽象方法。但只要类中有一个方法是被声明为abstract，则该类必须为abstract。

•抽象类不能被实例化，即不能被new成一个实例对象。 如：new 抽象类（ ）；则编译时会出现这样的错误：abstract class cannot be instantiated。

•若一个子类继承一个抽象类，则子类需要用覆盖的方式来重写该抽象超类中的所有抽象方法。若没有完全重写所有的抽象方法，则子类仍是抽象的。

•抽象方法可再与public、protected复合使用，但不能与final、private和static复合使用。

定义一个抽象类的格式如下：

abstract　class　类名

{

成员变量；

方法（ ）；//定义一般成员方法

abstract 方法（ ）；　//定义抽象方法

}

⑧ this与super

this与super分别指着有继承关系的子类和父类（超类）。this出现在程序代码中，指的是所在的该类当前对象。super指的是所继承的超类对象。this与super 均不用先声明即可直接使用。super的使用有三种情况：

•用来访问父类被隐藏的成员变量，如：super.变量名;

•用来调用父类中被重写的方法，如：super.方法名（参数）;

•用来调用父类的构造函数，如：super（参数）;

⑨ 接口

在Java语言中只支持单一继承，接口主要是为了解决多重继承的问题。

•接口的定义

接口的定义包括两部分：接口声明和接口体，定义格式如下：

[访问权限]　interface　接口名　[extends 父接口名1，父接口名2，…]{

// 接口体

}

•接口的实现

接口的实现是指在一个类的声明中使用关键字“implements”，这样的子句来表示该类使用某个已经定义的接口，然后就可以在类体中使用接口中定义的常量，而且必须实现接口中定义的所有方法。

⑩ 包

由于Java编译器编译Java源文件时，生成与文件名同名的字节码文件（类文件），因此同名的类有可能发生冲突。为了解决这样的问题，Java提供包来管理类。Java将其API中相关的类及接口组织成一个包（package）。当需要包中的类时，只要用import关键字引入一个包，便可将此包中所有的接口及类都引用进来。

•package语句

package语句是Java源文件的第一条语句，它指明了该文件中所有定义的类所在的包。格式如

下：package pakg1[.pakg2][.pakg3…];

•import语句

为了能够使用Java中已经提供的类，或者能够在其他类中使用自己定义的包中的类，我们需要使用import语句引入所需要的类。import语句格式如下：import pack1[.pack2][.pack3…].（classname|*）;

6．异常处理

在Java程序设计中，针对异常（例外）处理的方法有两种：一种方法是使用try…….catch……finally语句对异常进行捕获和处理。第二种方法是通过throws和throw抛弃异常。

① try……catch……finally语句

try---catch---finally语句语句的格式如下：

```
try{
……    //可能出现的异常的程序代码
}catch( ExceptionName1 e ){
……      //处理例外事件1
}catch( ExceptionName2 e ){
…… //处理例外事件2
}
……
}finally{
……
}
```

• try

捕获例外的第一步是用try{…}选定捕获例外的范围，由try所限定的代码块中的语句在执行过程中可能会生成例外对象并抛弃。

• catch

每个try代码块可以伴随一个或多个catch语句，用于处理try代码块中所生成的例外事件。catch语句只需要一个形式参数指明它所能够捕获的例外类型,这个类必须是Throwable的子类,运行时系统通过参数值把被抛弃的例外对象传递给catch块。

• finally

捕获例外的最后一步是通过finally语句为例外处理提供一个统一的出口，使得在控制流程到程序的其它部分以前，能够对程序的状态作统一的管理。不论在try代码块中是否发生了异常事件，finally块中的语句都会被执行。

② throws与throw

• throws子句

声明抛弃例外是在一个方法声明中的throws子句中指明的。格式如下：

```
方法声明( )throws 例外类型1，例外类型2，……
{
……
}
```

throws子句中同时可以指明多个例外，说明该方法将不对这些例外进行处理，而是声明抛弃它们。

• throw语句

抛弃例外首先要生成例外对象，生成例外对象也可以在程序中生成，可以通过throw语句实现。例如：

```
IOException e = new IOException( );
throw e ;
```

7．JDBC

假设数据源名:student　数据库名:student ，利用JDBC实现对ACCESS数据库操作，其操作过程如下。

```
import java.sql.*;//引入SQL包
public class DBM
{
   ResultSet rs; //定义返回结果值
   String strurl = "jdbc:odbc:student";//创建指定数据库的URL
   String strdriver = "sun.jdbc.odbc.JdbcOdbcDriver";//创建驱动程序
   public Connection con = null;
   public Statement st = null;
   //查询方法,返回查询结果集
   public ResultSet getResult( String sql )
   {
     try
     {
       Class.forName( strdriver );
       con = DriverManager.getConnection( strurl );
       st = con.createStatement( );
       rs = st.executeQuery( sql );
       return rs;
      }//end try
     catch( Exception e )
     {
       System.out.println( "getResult  "+e.toString( ) );
       return null;
     }//end catch
   }//end getResult
    //执行更新、删除语句方法
   public void executeSql( String sql )
{
     try{
```

```
            Class.forName( strdriver );
            con = DriverManager.getConnection( strurl );
            st = con.createStatement( );
            st.executeUpdate( sql );
            con.commit( );
          }//end try
        catch( Exception e )
        {
            System.out.println( "getResult  " + e.toString( ) );
        }//end catch
      }//end executeSql
    public void close( ) throws SQLException{
        if( con! = null ) con.close( );
     }//end close
}//end DBM
```

8．AWT、Swing用户界面与事件处理机制

① AWT、Swing用户界面

Swing组件是用Java实现的轻量级（light-weight）组件，没有本地代码，不依赖操作系统的支持，这是它与AWT组件的最大区别。

按功能划分：

• 顶层容器：JFrame,JApplet,JDialog,JWindow

• 中间容器：JPanel,JScrollPane,JSplitPane,JToolBar

• 特殊容器：在GUI上起特殊作用的中间层，如JInternalFrame,JLayeredPane,JRootPane

• 基本控件：实现人际交互的组件，如Jbutton, JComboBox, JList, JMenu, JSlider, JtextField

• 不可编辑信息的显示：JLabel, JProgressBar, ToolTip。

• 可编辑信息的格式化显示:JColorChooser, JFileChoose, JFileChooser, Jtable, JtextArea

对JFrame添加组件有两种方式：

• 用getContentPane( )方法获得JFrame的内容面板，再对其加入组件：

```
frame.getContentPane( ).add( childComponent )
```

• 建立一个Jpanel或 JDesktopPane之类的中间容器，把组件添加到容器中，用setContentPane( )方法把该容器置为JFrame的内容面板：

```
Jpanel contentPane = new Jpanel( );
        ……//把其它组件添加到Jpanel中;
frame.setContentPane( contentPane );
/把contentPane对象设置成为frame的内容面板
```

② 事件处理机制

事件监听器类应该包括以下两部分内容：

▼在事件监听器类的声明中指定要实现的监听器接口名，如:

```
public class MyListener implements XxxListener {
```

…

}

▼实现监听器接口中的事件处理方法，如：

public void 事件处理方法名（XxxEvent e）{

…//处理某个事件的代码…

}

然后，在一个或多个组件上可以进行监听器类的实例的注册。如：组件对象.addXxxListener（MyListener对象）;

•事件监听器类实例

```
import java awt.event.* ;
public class ButtonHandler implements ActionListener
{
public void actionPerformed( ActionEvent e )
{
System.out.println( "Action occurred" );
System.out.println( "Button's label is:"+
e.getActionCommand( ) );
}
}
```

•使用事件监听器类

```
import java.awt.*;
public class TestButton
{
public static void main( String args[ ] )
{
Frame f = new Frame( "Test" );
Button b = new Button( "Press Me!" );
b.addActionListener( new ButtonHandler( ) );
f.add( b, "Center" );
f.pack( );
f.setVisible( true ) ;
}
}
```

版权方授权希赛网发布，侵权必究

设计模式（一）

本节对设计模式进行了精要的概括与总结，并给出了23种设计模式的UML结构图，并根据理解需要，对少数模式给出了实现代码，读者可在后面小节的典型真题解析、考前必练中结合本节相关知识，加深对某些模式的理解。

1．设计模式定义

模式是一种问题的解决思路，它已经适用于一个实践环境，并且可以适用于其它环境。设计模式通常是对于某一类软件设计问题的可重用的解决方案，将设计模式引入软件设计和开发过程，其目的就在于要重用软件开发经验。一般模式有4个基本要素：模式名称（pattern name）、问题（problem）、解决方案（solution）、效果（consequences）。

2．设计模式的作用

设计模式主要有以下作用：

（1）重用设计，重用设计比重用代码更有意义，它会自动带来代码重用。

（2）为设计提供共同的词汇，每个模式名就是一个设计词汇，其概念是的程序员间的交流更加方便。

（3）在开发文档中采用模式词汇可以让其他人更容易理解你的想法和做法，编写开发文档也更加容易。

（4）应用设计模式可以让重构系统变得容易，可确保开发正确的代码，并降低在设计或实现中出现错误的可能。

（5）支持变化，可以为重写其他应用程序提供很好的系统架构。

（6）正确使用设计模式，可以节省大量时间。

3．组织编目

目前流行的面向对象设计模式，仅1995年“Gang of four”四个人：Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides描述的模式就有23种，这些模式称作GOF模式。GOF模式在力度和抽象层次上各不相同，分类很明显。

① 根据目的准备分类：

Creational 创建型 ：和对象创建有关。

Structural 结构型 ：处理类或对象的组合。

Behavioral 行为型 ：描述类或对象如何交互以及如何分配职责。

② 根据范围准则分类：即制定的模式是用于类还是用于对象，分为两种：

•类模式：用于处理类和类之间的关系，这些关系通过继承建立，是静态的，在编译时就已经确定下来了。从某种意义上说，几乎所有模式都是使用继承机制，，只有很少部分模式属于这一类。

•对象模式：用于处理对象间的关系，这些关系具有动态性，在运行期间是可以变化的。

由以上分类标准可以把23种GOF模式进行类别划分，结果见表6-4如下：

表6-4 设计模式的划分

目的 范围	创建型	结构型	行为型
类	Factory Method（工厂模式）	Adapter（适配器模式）	Interpreter（解释器模式） Template Method（模板方法模式）
对象	Abstract Factory（抽象工厂模式） Builder（生成器模式） Singleton（单件模式） Prototype（原型模式）	Adapter（适配器模式） Bridge（桥接模式） Composite（组合模式） Decorator（装饰模式） Facade（外观模式，门面模式） Flyweight（享元模式） Proxy（代理模式）	Chain of Responsibility（职责链模式） Command（命令模式） Iterator（迭代器模式） Mediator（中介者模式） Memento（备忘录模式） Observer（观察者模式） State（状态模式） Strategy（策略模式） Visitor（访问者模式）

表6-5 各设计模式所支持的设计可变方面

目的	设计模式	可变方面
创建型	Abstract Factory（抽象工厂模式）	产品对象家族
	Builder（生成器模式）	如何创建一个组合对象
	Factory Method（工厂模式）	被实例化的子类
	Singleton（单件模式）	一个类的唯一实例
	Prototype（原型模式）	被实例化的类
结构型	Adapter（适配器模式）	对象的接口
	Bridge（桥接模式）	对象的实现
	Composite（组合模式）	一个对象的结构和组成
	Decorator（装饰模式）	对象的职责，不生成子类
	Facade（外观模式，门面模式）	一个子系统的接口
	Flyweight（享元模式）	对象的
	Proxy（代理模式）	如访问一个对象，该对象的位置
行为型	Chain of Responsibility（职责链模式）	满足一个请求的对象
	Command（命令模式）	何时、怎样满足一个请求
	Interpreter（解释器模式）	一个语言的文法及解释
	Iterator（迭代器模式）	如何遍历、访问聚合的各元素
	Mediator（中介者模式）	对象间怎样交互、和谁交互
	Memento（备忘录模式）	一个对象中哪些私有信息存放在该对象之外，以及何时存储
	Observer（观察者模式）	多个对象依赖另外一个对象，而这些对象如何保持一致
	State（状态模式）	对象的状态
	Strategy（策略模式）	算法
	Template Method（模板方法模式）	算法中的某些步骤
	Visitor（访问者模式）	作用于一个（组）对象上的操作，但不修改这些对象的类

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

作者：希赛教育软考学院　来源：希赛网　2014年05月06日

设计模式（二）

4．模式讲解

下面对这些模式分三类（创建型、结构型、行为型）进行简要介绍：

（1）创建型模式

在OO中，创建对象没有什么困难，但是如果基于不同的情况创建不同的对象，这个过程就不容易了。创建型模式对类的实例化过程进行了抽象，能够使软件模块做到与对象的创建和组织无关。

1）Factory Method（工厂模式）

工厂模式的意图是：不直接通过对象的具体实现类，而是通过使用专门的类来负责一组相关联的对象的创建。即定义一个用于创建对象的接口，让子类决定将哪一个类实例化，这样做的目的是将类的实例化操作延迟到子类中完成，即由子类来决定究竟应该实例化（创建）哪一个类。Factory Method 使一个类的实例化延迟到其子类。

工厂模式的适应性分别如下：

·当一个类不知道它所必须创建的对象的类的时候。

·当一个类希望由它的子类来指定它所创建的对象的时候。

·客户需要清楚创建了哪个对象候。

工厂模式的结构（UML类图）如图6-5所示：

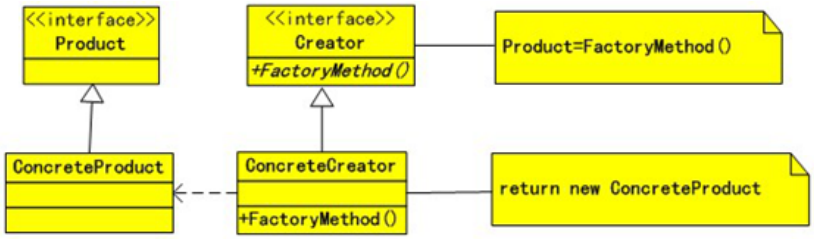

图6-5 工厂模式类图

其中：

·Product：产品角色定义产品的接口。

·ConcreteProduct：真实的产品,实现接口Product的类。

·Creator：工厂角色声明工厂方法（Factory Method），返回一个产品。

·ConcreteCreator：真实的工厂实现Factory Method工厂方法，由客户调用，返回一个产品的实例。

工厂模式的优势和缺陷分别如下：

·在工厂方法模式中，工厂方法用来创建客户需要的产品，同时还向客户隐藏了哪种具体产品类将被实例化这一细节。工厂方法模式的核心是一个抽象工厂类，各种具体工厂类通过抽象工厂类将工厂方法继承下来。使得客户可以只关心抽象产品和抽象工厂，不用关心它是如何被具体工厂创建的。基于工厂角色和产品角色的多态性是设计工厂方法模式的关键。它能够使工厂可以自主确定创建何种产品对象，而如何创建这个对象的细节则完全封装在具体工厂内部。工厂方法模式因而又被叫做多态工厂模式，就是因为所有的具体工厂类都具有同一个抽象父类。使用工厂方法模式的另外一个优点是在系统中加入新产品时，无需修改抽象工厂和抽象产品提供的接口，无须修改客户端，也无需修改其他的具体工厂和具体产品，而只要添加一个具体工厂和具体产品就可以了，这样，系统的可扩展性非常好。优秀的面向对象设计鼓励使用封装和委托来构造软件系统，工厂方法模式正是使用了封装和委托的典型例子，其中封装是通过抽象工厂来完成的，而委托则是通过抽象工厂将创建对象的负责完全交给具体工厂来实现的 ，也就是利用多态。

·使用工厂方法模式的缺点是在添加新产品是，需要编写新的具体产品类，而且要提供相应的具体工厂类，当两者都比较简单时，系统会有相对额外的开销。

2）Abstract Factory（抽象工厂模式）

抽象工厂模式的意图是：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

抽象工厂模式的适用性分别如下：

·一个系统要独立于它的产品的创建、组合和表示时。

·一个系统要由多个产品系列中的一个来配置时。

·当你要强调一系列相关的产品对象的设计以便进行联合使用时。

·想提供一组对象而不显示他们的实现过程，只显示他们的接口时。

抽象工厂模式的结构（UML类图）如图6-6所示。

如果一开始所有的对象都是直接创建，例如通过new实例化的，而之后想重构为Abstract Factory模式，那么，很自然的我们需要替换所有直接的new实例化代码为对工厂类对象创建方法的调用。抽象工厂负责创建不同的有联系的多个产品，不同的抽象工厂创建的产品不同，但是产品之间的关系相同。

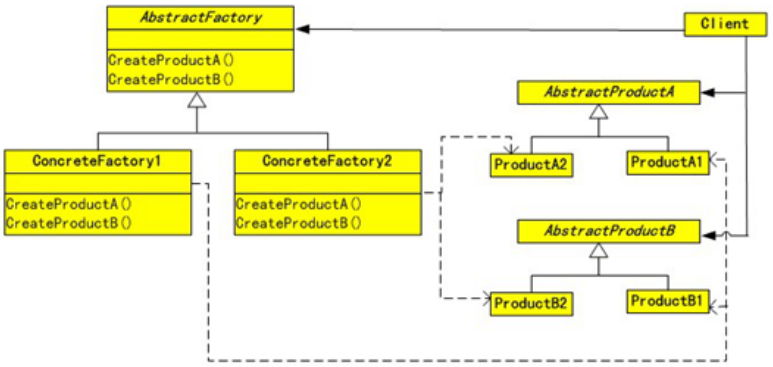

图6-6 抽象工厂模式的类图

其中：

·AbstractFactory：抽象工厂，声明抽象产品的方法。

·ConcreteFactory：具体工厂，执行生成抽象产品的方法，生成一个具体的产品。

·AbstractProduct：抽象产品，为一种产品声明接口。

·Product：具体产品，定义具体工厂生成的具体产品的对象，实现产品接口。

·Client：客户，我们的应用程序使用抽象产品和抽象工厂生成对象。

抽象工厂模式的优势和缺陷分别如下：

·抽象工厂模式的主要优点是隔离了具体类的生成，使得客户不需要知道什么被创建，由于这种隔离，更换一个具体工厂就变得相对容易。所有的具体工厂都实现了抽象工厂中定义的那些公共接口，因此只需要改变具体工厂的实例，就可以在某种程度上改变整个软件系统的行为.使用工厂模式的最大好处是：当同一个产品族中的多个对象被设计成一起工作时，他能够保证客户端始终只使用同一个产品族中的对象。这对于一些需要根据当前环境来决定其行为的软件来说是一种非常使实用的设计模式。

·抽象工厂的缺点是：在添加新产品对象时，难以扩展抽象工厂以便产生新的产品。这是因为Abstract Factory接口规定了所有可能被创建的产品集合，要支持新种类的产品就意味着要对该接口进行扩展，而这涉及到对Abstract Factory及其所有子类的修改，有些不方便。

3）Builder（生成器模式/建造者模式）

生成器模式的意图是：将一个复杂对象的构建和他的表示分离，使得同样的构建过程可以创建不同的表示。建造者模式是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建他们。用户不知道内部的具体构建细节。

生成器模式的适用性分别如下：

·当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。

·当构造过程必须允许被构造的对象有不同的表示时。

生成器模式的结构（UML类图）如图6-7所示。

建造者模式，听名字就应该知道和工厂模式一样，是用来创造对象的。但是建造者和工厂模式的区别就是工厂模式只关注最终的产品，它往往是简单的调用被创建者的构造函数；而建造者更关心细节，它定义了创建一个复杂对象所需的步骤，而创建者具体的实现类可根据具体的需求，调整创建细节。比如建造一个人: 抽象建造者把造人的过程分解为: 建造头部,躯杆，四肢, 那么不同的具体建造者可以根据客户需要, 建造出不同的人来: 有的人躯体胖, 有的人手臂长，这是构造函数无法提供的弹性。

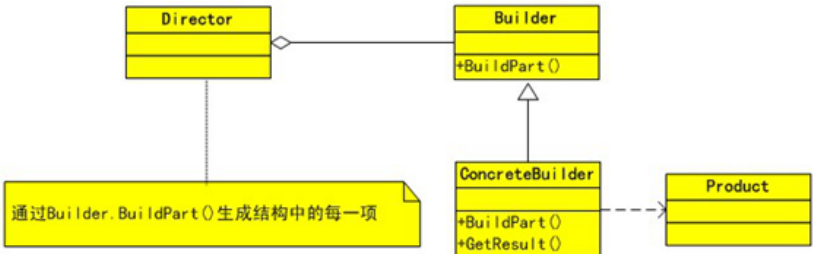

图6-7 Builder（生成器模式）的类图

其中：

·Builder：抽象建造者，为创建一个Product对象各个部件指定抽象接口，把产品的生产过程分解为不同的步骤，从而使具体建造者在具体的建造步骤上具有更对弹性，从而创造出不同表示的产品。

·ConcreteBuilder：具体建造者，实现Builder接口，构造和装配产品的各个部件定义并明确它所创建的表示，提供一个返回这个产品的接口。

·Director：指挥者，构建一个是使用Builder接口的对象。

·Product：产品角色，被构建的复杂对象，具体产品建造者，创建该产品的内部表示并定义他的装配过程。

建造者模式的优势和缺陷分别如下：

该模式的重构成本非常低的，因为一般来讲，创建过程的代码本来也就应该在原来的类的构造函数中，把它Extract出来就好了。如果发现多个类的创建过程有比较多的代码重复或类似，那么就可以重用这些提取出来的Builder类或者Builder类中的某些阶段。另外如果产品的内部变化复杂。Builder的每一个子类都需要对应到不同的产品去做构建的动作方法，这就需要定义很多个具体构造类来实现这种变化。

4）Singleton（单件模式/单例模式）

单件模式/单例模式的意图是：确保某个类只有一个实例，且自行实例化，并向整个系统提供这个实例。

单件模式/单例模式的适用性如下：

·当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。

·当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。

Singleton模式的结构（UML类图）见图6-8所示。

图6-8 Singleton模式的类图

其中：

Singleton：单例,提供一个instance的方法，让客户可以使用它的唯一实例。内部实现只生成一个实例。

Singleton单例模式为一个面向对象的应用程序提供了对象唯一的访问点，不管它实现何种功能，这种模式都未涉及以及开发团队提供了共享的概念，然而，Singleton对象派生子类就有很大困难，只有在父类没有被实例化的时候才能实现。

5）Prototype（原型模式）

原型模式的意图是：原型模式指定创建对象的种类，并且通过拷贝这些原型创建新的对象。Prototype模式允许一个对象再次创建另外一个可定制的对象，根本无须知道任何创建的细节。工作原理是：通过将一个原型对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型对象拷贝原型自己来实施创建过程。

原型模式的适用性如下：

·类的实例化是动态的。

·需要避免使用分层次的工厂类来创建分层次的对象。

·类的实例对象只有一个或者很少的几个状态组合。

原型模式的结构（UML类图）如图6-9所示。

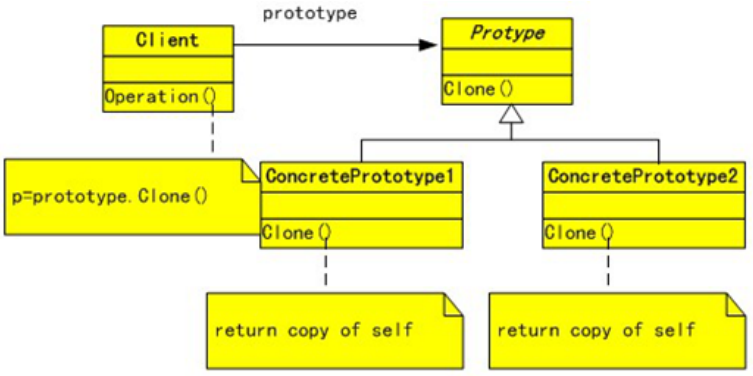

图6-9 Prototype（原型模式）的类图

其中：

·Prototype：抽象原型类,定义具有克隆自己的方法的接口。

·ConcretePrototype：具体原型类,实现具体的克隆方法。

·Client：客户。

原型模式得到了广泛的应用，特别是在创建对象成本较大的情况下（初始化需要占用较长时间，占用太多CPU资源或者网络资源，或者创建对象需要装在大文件），系统如果需要重复利用，新的对象可以通过原型模式对已经存在对象的属性进行复制并且稍作修改获得。另外，如果系统要保存对象的状态，而对象的状态变化很小，或者对象本身占用内存不多的时候，也可以用原型模式配合备忘录模式来使用。相反地，如果对象状态变化很大，或者对象占用内存很大那么采用状态模式比原型模式更好，原型模式的缺点是在实现深层复制的时候需要编写复杂的代码。

（2）结构型模式

结构型模式描述类和对象之间如何进行有效的组织，以形成良好的软件体系结构，主要的方法是使用继承关系来组织各个类，一个最容易的例子就是如何用多个继承组织两个以上的类，结果产生的类结合了父类所有的属性，结构型模式特别适用于和独立的类库一起工作。

1）Adapter（适配器模式）

适配器模式的意图是：将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

适配器模式的适用性如下：

·对象需要利用现存的并且接口不兼容的类。

·你需要创建可重用的类以协作其他接口不一定兼容的类。

·你需要使用若干个现存的子类，但又不想派生这些子类的每一个接口。

适配器模式的结构（UML类图）如图6-10所示。

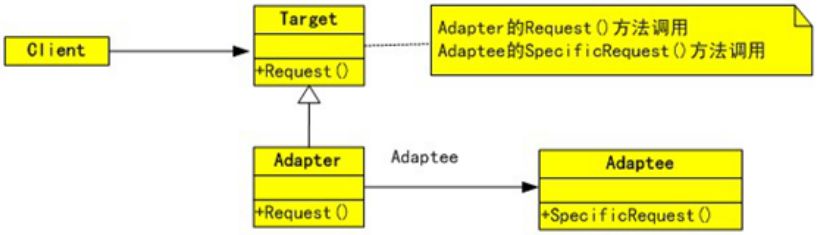

图6-10：Adapter（适配器模式）的类图

在Java中实现Adapter的方式有两种，分别是：组合（composition）和继承（inheritance）。

假设我们要打桩，有两种类：方形桩、圆形桩。

```
public class SquarePeg{
    public void insert( String str ){
        System.out.println( "SquarePeg insert( ):"+str );
    }
}
public class RoundPeg{
    public void insertIntohole( String msg ){
        System.out.println( "RoundPeg insertIntoHole( ):"+msg );
}
}
```

现在有一个应用，需要既打方形桩，又打圆形桩。那么我们需要将这两个没有关系的类综合应用，假设RoundPeg我们没有源代码,或源代码我们不想修改，那么我们使用Adapter来实现这个应用：

```
public class PegAdapter extends SquarePeg{
    private RoundPeg roundPeg;
    public PegAdapter( RoundPeg peg )( this.roundPeg = peg; )
    public void insert( String str ){ roundPeg.insertIntoHole( str );}
}
```

在上面代码中，RoundPeg属于Adaptee，是被适配者。PegAdapter是Adapter，将Adaptee（被适配者RoundPeg）和Target（目标SquarePeg）进行适配，实际上这是将组合方法（composition）和继承（inheritance）方法综合运用。PegAdapter首先继承SquarePeg，然后使用new的组合生成对象方式，生成RoundPeg的对象roundPeg，再重载父类insert( )方法。从这里,你也了解使用new生成对象和使用extends继承生成对象的不同,前者无需对原来的类修改,甚至无

需要知道其内部结构和源代码。

适配器模式可以将一个类的接口和另一个类的接口匹配起来，适用的前提是你不能或不想修改原来的适配器母接口（Adaptee）。比如，你向第三方购买了一些类、控件，但是没有元程序，这时，适用适配器模式，你就可以统一对象访问接口。

2）Bridge（桥接模式）

桥接模式的意图是：将抽象部分与它的实现部分分离，使它们都可以独立地变化。

桥接模式的适用性如下：

·避免抽象方法和实现方法邦定在一起。

·类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。

·对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。

·想在多个对象间共享实现（可能使用引用计数），但同时要求客户并不知道这一点。

桥接模式的结构（UML类图）如图6-11所示。

桥接模式是把继承关系变成合成/聚合关系。手机可以按照品牌来分类，则有手机品牌M，手机品牌N之分，现在的每个手机都有很多软件，比如通讯录，手机游戏等等，运用桥接模式，可把手机系统划分为品牌和软件，使他们可以独立的变化。

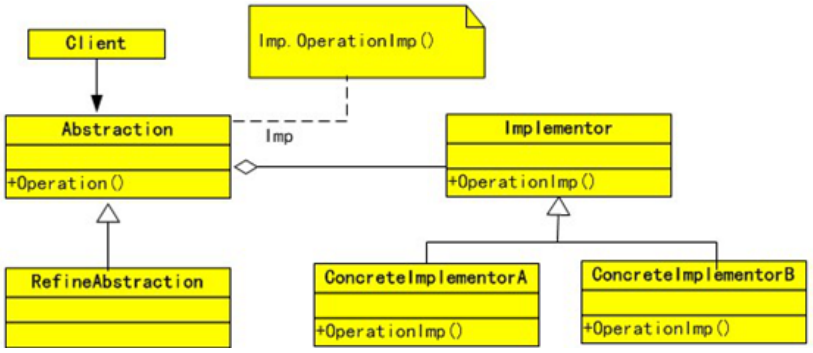

图6-11 Bridge（桥接模式）的类图

其中：

·Abstraction：抽象类定义抽象类的接口。维护一个Implementor（实现抽象类）的对象

·RefinedAbstraction：扩充的抽象类,扩充由Abstraction定义的接口。

·Implementor：实现类接口,定义实现类的接口，这个接口不一定要与Abstraction的接口完全一致，事实上这两个接口可以完全不同，一般的讲Implementor接口仅仅给出基本操作，而Abstraction接口则会给出很多更复杂的操作。

·ConcreteImplementor：具体实现类,实现Implementor定义的接口并且具体实现它。

桥接模式可以从接口中分离实现功能，使得设计更具有扩展性，这样，客户调用方法时根本不需要知道实现的细节。桥接模式的缺陷是：抽象类和实现类的双向连接使得运行速度减慢。

3）Composite（组合模式）

组合模式的意图是将对象组合成树形结构以表示“部分-整体”的层次结构,组合模式对单个对象和组合对象的使用具有一致性。

组合模式的适用性如下：

·想表示对象的部分-整体层次结构。

·希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

组合模式的结构（UML类图）如图6-12所示。

组合模式使得用户对单个对象和组合对象的适用具有一致性，可让客户可以一致的适用组合结

构和单个对象。比如文件系统中有文件，也有文件夹（其实是特殊的文件），文件和文件夹组合在一起，可以成为一个更大的文件夹。在用户看来，不管是文件，还是文件夹，还是更目录，对它们的操作都是一致的（当然试图把一个文件夹放到文件下面是不容许的）。这就是说整体和部分具有一致的接口。

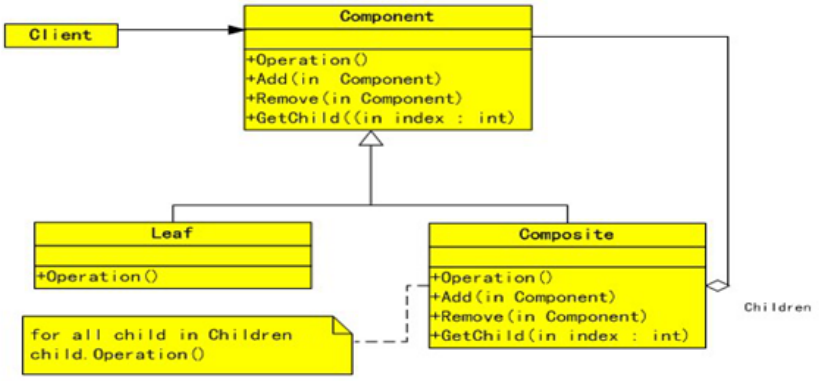

图6-12 Composite（组合模式）的类图

其中：

·Leaf：叶部件,在组合中表示叶节点对象，叶节点没有子节点。定义组合中原有接口对象的行为。

·Composite：组合类,定义有子节点（子部件）的部件的行为。

在Component接口中实现与子部件相关的操作。

·Client：客户应用程序,通过Component接口控制组合部件的对象。

组合模式可以清楚地定义分层次的复杂对象。表示对象的全部或者部分层次，使得增加新部件也更容易，并且它的结构也是动态的，提供了对象管理的灵活接口。

组合模式的缺陷是使得设计变得更加抽象，对象的商业规则如果很复杂，则实现组合模式有很大的挑战性，并不是所有的方法都与叶部件子类有关联。

4）Decorator（装饰模式）

装饰模式的意图是：动态的给一个对象增加职责，即在不必改变原类文件和使用继承的情况下，动态的扩展一个对象的功能。它是通过创建一个包装对象，也就是装饰来包裹真实的对象。

装饰模式的适用性如下：

·在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。处理那些可以撤消的职责。

·当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。

装饰模式的结构（UML类图）如图6-13所示。

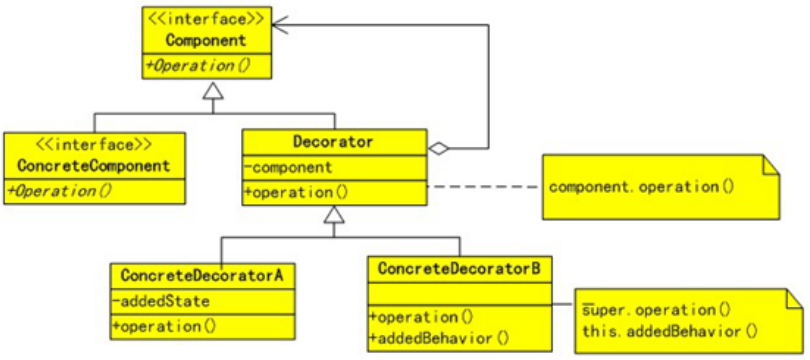

图6-13 Decorator（装饰模式）的类图

其中：

·Component：部件,定义对象的接口，可以给这些对象动态的增加职责（方法）。

·Concrete Component：具体部件,定义具体的对象，Decorator可以给他增加额外的职责（方法）。

·Decorator：装饰抽象类,维护一个内有的Component，并且定义一个与Component接口一致的接口。

·Concrete Decorator：具体装饰对象，给内在的具体部件对象增加具体的职责（方法）。

装饰模式在拓展类的功能上比继承拥有更大的灵活性，它允许开发一系列的功能类来代替增加对象的行为，这既不会污染原来对象的代码，还能使得代码更容易编写，使类更具有扩展性，因为变化都是由新的装饰类来完成的。还可以建立连接的装饰对象关系链。需要注意的是，装饰链不应该过长。装饰链太长会使系统花费较长时间用于初始化对象，同时信息在链中的传递也会浪费太多的时间。这种情况好比物品包装，包了一层又一层，大包小包。另外，如果原来的对象接口发生变化，它所有的装饰类都要修改已匹配它的变化。派生子类会影响对象的内部，而一个Decorator只会影响对象的外表。

5）Facade（外观模式，门面模式）

外观模式的意图是：为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

外观模式的适用性如下：

·当要为一个复杂子系统提供一个简单接口时

·客户程序与抽象类的实现部分之间存在着很大的依赖性。

·当需要构建一个层次结构的子系统时，使用外观模式定义子系统中每层的入口点。

外观模式的结构（UML类图）如图6-14所示。

门面模式为多个子系统提供统一的高层接口，以降低客户对子系统的依赖, 减少客户使用子系统的难度.比如汽车分好多子系统: 发动机, 刹车系统, 变速系统, 电子系统等，如果司机在开车的时候要直接和这些子系统打交道, 那这个车就很难开。但有驾驶室这个操作平台, 由驾驶室里面的油门, 刹车, 换档把手, 手札, 开关按钮来和这些子系统打交道. 而驾驶员只需要坐在驾驶室里就可以操控整个车，这个驾驶室就是这整个汽车全部系统的门面。

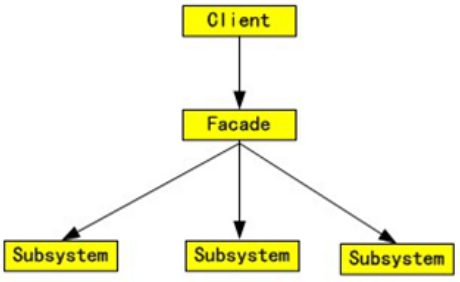

图6-14 Facade（外观模式）的类图

其中：

·Facade：外形类，知道哪些子系统负责处理那些请求，将客户的请求传递给相应得子系统对象处理。

·Subsystem：子系统类，实现子系统的功能，处理由Façade传过来的任务。子系统不用知道Façade，在任何地方也没有引用Facade。

外观模式为子系统提供了一个更高层次、更简单的接口，从而降低了子系统的复杂度和依赖。外观对象隔离了客户和子系统对象，从而降低了耦合度。当子系统中的类进行改变时，客户端不会像以前一样受到影响。这使得子系统更易于使用和管理。外观模式的劣势就是限制了客户的自由，减少了可变性。

6）Flyweight（享元模式,轻量级模式）

享元模式/轻量级模式的意图是：运用共享技术有效的支持大量细粒度的对象。系统只是用少量的对象，而这些对象都相近，状态变化很小，对象使用次数增多。

享元模式/轻量级模式的适用性如下：

·一个应用程序使用了大量的对象。

·完全由于使用大量的对象，造成很大的存储开销。

·对象的大多数状态都可变为外部状态。

·如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。

·应用程序不依赖于对象标识。由于Flyweight对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值。

享元模式的结构（UML类图）如图6-15所示。

享元模式能做到共享的关键是区分内蕴状态和外蕴状态。内蕴状态存储在享元内部，不会随环境的改变而有所不同。外蕴状态是随环境的改变而改变的。外蕴状态不能影响内蕴状态，它们是相互独立的。将可以共享的状态和不可以共享的状态从常规类中区分开来。客户端不可以直接创建被共享的对象，而应当使用一个工厂对象负责创建被共享的对象。

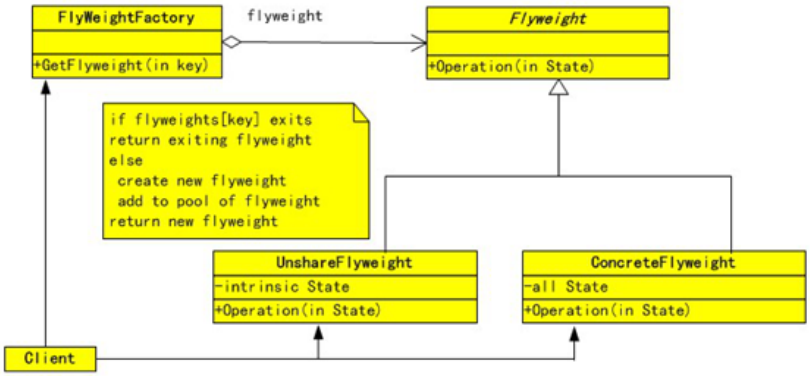

图6-15 Flyweight（享元模式）的类图

其中：

·Flyweight：抽象轻量级类,声明一个接口，通过它可以接受外来的参数（状态），并对新状态作出处理（作用）。

·ConcreteFlyweight：具体轻量级类,实现Flyweight接口，并且为内部状态（如果有）增加存储空间。ConcreteFlyweight对象必须是可以共享的。它所存储的状态必须是内部的，即它独立存在于自己的环境中。

·UnsharedConcreteFlyweight：不共享的具体轻量级类,不是所有的Flyweight子类都需要被共享，Flyweight的共享不是强制的。在某些Flyweight的结构层次中，UnsharedFlyweight对象常常将ConcreteFlyweight对象作为子节点。

·FlyweightFactory：轻量级类工厂,创建并且管理flyweight对象确保享用flyweight。当用户请求一个flyweight时，FlyweightFactory提供一个已创建的实例或者创建实例（如果不存在）。

·Client：客户应用程序.

Flyweight模式需要认真考虑如何能细化对象，以减少对象的数量，从而减少存留对象在内存或其他存储设备中的占用量。然而，此模式需要维护大量对象的外部状态，如果外部对象的数据量大，传递查找计算这些数据会变得非常复杂。当外部和内部状态很难分清时，不宜采用flyweight模式。

7）Proxy（代理模式）

代理模式的意图是：为其他对象提供一个代理或地方以控制对这个对象的访问。当客户向Proxy对象第一次提出请求时，Proxy实例化真实的对象，并且将请求传给它，以后所有客户的请求都由Proxy传递给封装了的对象。

在需要用比较通用和复杂的对象指针代替简单的指针的时候，使用Proxy模式。下面是一些可以使用Proxy模式的常见情况：

·远程代理（Remote Proxy）：为一个对象在不同的地址空间提供局部代表。

·虚代理（Virtual Proxy）：根据需要创建开销很大的对象。

·保护代理（Protection Proxy）：控制对原始对象的访问。保护代理用于对象应该有不同的访问权限的时候。

·智能指引（Smart Reference）：取代了简单的指针，它在访问对象时执行一些附加操作。它的典型用途包括：对指向实际对象的引用计数，这样当该对象没有引用时，可以自动释放它；当第一次引用一个持久对象时，将它装入内存；在访问一个实际对象前，检查是否已经锁定了它，以确保其他对象不能改变它。

代理模式的结构（UML类图）如图6-16所示。

某些情况下，客户不想或者不能够直接引用一个对象，代理对象可以在客户和目标对象直接起到中介的作用。

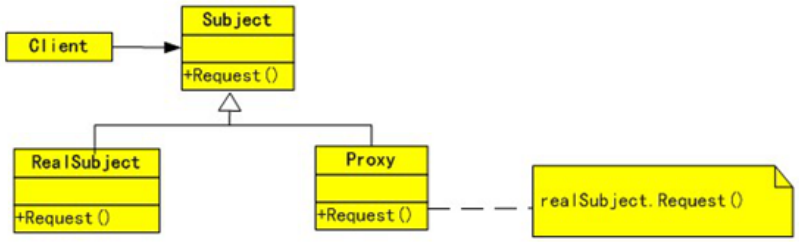

图6-16 Proxy（代理模式）的类图

其中：

·Proxy：代理维护一个引用使得代理可以访问实体，如果RealSubject和Subject的接口相同，则Proxy会引用Subject。其他功能取决于Proxy的类型。

·远程代理：负责对请求及其参数编码，向不同地址空间中的实体发送已编码的请求。

·虚拟代理：可以缓存实体的其他信息，以便延迟对它的访问。

·保护代理：检查调用者的请求是不是有所需的权限。

·Subject：抽象实体接口,为RealSubject实体和Proxy代理定义相同的接口，使得RealSubject在任何地方都可以使用Proxy来访问。

·RealSubject：真实对象，定义Proxy代理的实体。

当对象在远程机器上，要通过网络来生成时，速度可能会比较慢，这时，用代理模式（Remote Proxy）可以掩蔽对象由网络生成的过程，系统速度会加快，对于大图片的加载，Virtual Proxy可以让加载在后台进行，前台用的Proxy对象使得整体运行速度得到优化。Protection Proxy可以验证对真实对象的引用权限。

代理模式的缺陷是请求的处理速度会变慢，并且实现Proxy需要额外的工作。

（3）行为型设计模式

行为型设计模式描述类和对象之间如何交互以及如何分配职责，实际上它所牵涉的不仅仅是类或者对象的设计模式，还有他们之间的通信模式。

1）Chain of Responsibility（职责链模式）

职责链模式的意图是：为解除请求的发送者和接收者之间耦合，而使多个对象都有机会处理这个请求。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它。

职责链模式的适用性如下：

·有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。

·你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。可处理一个请求的对象集合应被动态指定。

职责链模式的结构（UML类图）如图6-17所示。

责任链模式的关键点在于把请求的处理者连成一条链,一个处理者可以处理当前请求,也有权决定是否沿着链朝上传递请求。

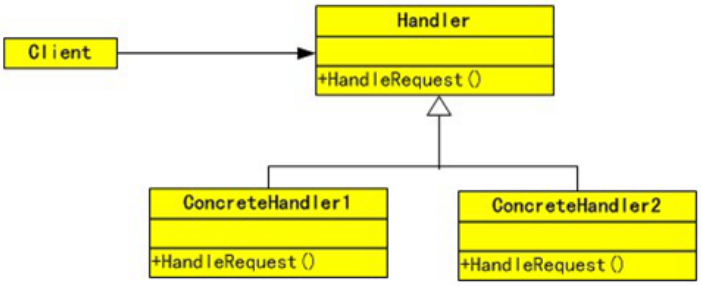

图6-17 Chain of Responsibility（职责链模式）的类图

其中：

·Handler：传递者接口，定义一个处理请求的接口。

·ConcreteHandler：具体传递者，处理它所负责的请求。可以访问链中下一个对象，如果可以处理请求，就处理它，否则将请求转发给后继者。

·Client：客户应用程序，向链中的对象提出最初的请求。

责任链模式可以减少对象的连接，为对象责任分配增加了很大的灵活性。该模式可以简化对象之间的连接，他们只需要知道一个后继者就行了，可以很方便的增加或修改处理者。树状责任链能够提供更灵活的技巧，但缺点是信息在树中容易迷失。

2）Command（命令模式）

命令模式的意图是：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。

命令模式的适用性如下：

·当你需要与动作有关的对象作为参数。

·在不同的时刻指定、排列和执行请求。

·你需要支持取消，修改和保存日志或处理事务（事务包括大量修改的数据）功能。

·支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。

·用构建在原语操作上的高层操作构造一个系统。这样一种结构在支持事务（transaction）的信息系统中很常见。一个事务封装了对数据的一组变动。模式提供了对事务进行建模的方法。Command有一个公共的接口，使得你可以用同一种方式调用所有的事务。同时使用该模式也易于

添加新事务以扩展系统。

命令模式的结构（UML类图）如图6-18所示

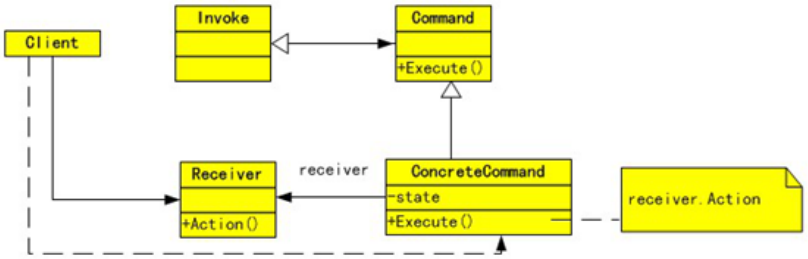

图6-18 Command（命令模式）的类图

其中

·Command：抽象命令类，声明执行操作的一个接口。

·ConcreteCommand：具体命令类将一个接收者对象绑定于一个动作。实现execute方法，以调用接收者的相关操作（Action）。

·Invoker：调用者，要求一个命令对象执行一个请求。

·Receiver：接收者，知道如何执行关联请求的相关操作。

·Client：客户应用程序，创建一个具体命令类对象，并且设定它的接收者。

命令模式分离了接收请求的对象和实现处理请求工作的对象，这样，已经存在的类可以保持不变。是的增加新类的工作变得简单。此外命令模式还可以分离用户界面和业务对象，降低系统的耦合度。

命令模式的最主要的缺点是：类的数量增加了，系统变得更复杂，程序的调试工作也相应的变得困难。

3）Interpreter（解释器模式）

解释器模式的意图是：给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

解释器模式的适用性如下：

·当有一个语言需要解释执行，并且你可将该语言中的句子表示为一个抽象语法树时，可使用解释器模式。而当存在以下情况时该模式效果最好：

·该文法简单对于复杂的文法，文法的类层次变得庞大而无法管理。

·效率不是一个关键问题最高效的解释器通常不是通过直接解释语法分析树实现的，而是首先将它们转换成另一种形式。

解释器模式的结构（UML类图）如图6-19所示。

解释器模式将描述怎样在有了一个简单的文法后，使用模式设计解释这些语句。银行经常要用复杂而变化的模型来计算数据，比如（1+r）n*c，（r = Rate, n = Years, c = Capital）这个公示就是计算一笔存款存n年之后的的总额.如果随着银行产品种类的增多，业务种类的增多，公式将会多种多样，如果可以有一个公式编辑器,程序通过解析公式（运算元素解析、运算符号解析），然后给公式的参数赋值就可以应对需求。那么这个程序就是一个解释器。

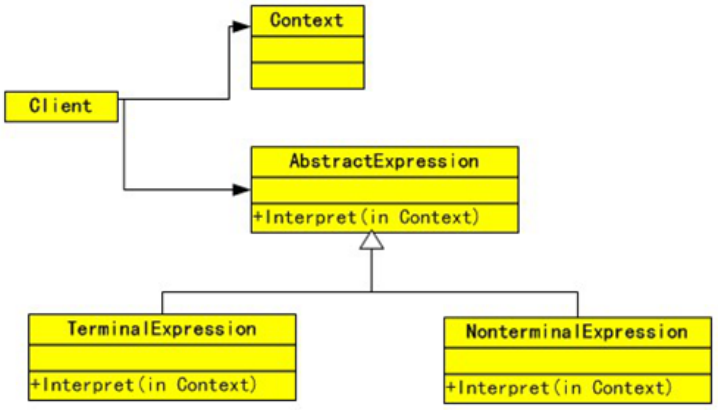

图6-19 Interpreter（解释器模式）的类图

其中：

·AbstractExpression：抽象表达式类，定义一个接口来执行解释操作，实现与文法中的元素相关联的解释操作。

·TerminalExpression：终结符表达式，实现文法中关联终结符的解释操作。文句中的每个终结符都需要一个实例。

·NonTerminalExpression：非终结符表达式

·Context：场景，包含解释器的所有全局信息。

·Client：客户程序，建造（或被给定）由这种语言表示的句子的抽象文法树，文法树由终结符表达式或非终结表达式的实例组成。

解释器模式的作用很强大，它使得改变和扩展文法很容易，实现文法也变得简单明了，很多编译器，包括文本编辑器，网页浏览器都用到了解释器模式。

解释器模式的缺陷在于：因为文句分析成树结构，解释器需要递归访问它，所以效率会受到影响，编译整个工程耗时比较长。

4）Iterator（迭代器模式）

迭代器模式的意图是：提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。

迭代器模式的适用性如下：

·需要遍历访问聚集中的对象而不能暴露聚集的内部结构。

·允许对聚集的多级遍历访问而不会相互受到影响。

·提供一个一致的接口来遍历访问集中不同的结构。

迭代器模式的结构（UML类图）如图6-20所示。

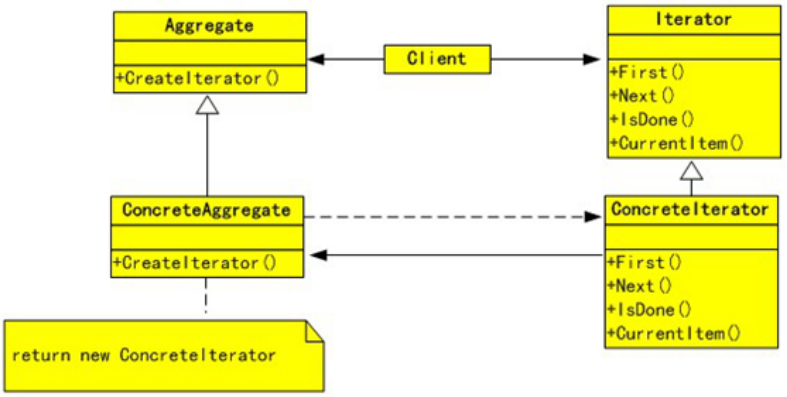

图6-20 Iterator（迭代器模式）的类图

其中：

·Iterator：迭代器，迭代器定义访问和遍历元素的接口。

·ConcreteIterator：具体迭代器，实现迭代器的借口，在遍历时跟踪当前聚合对象中的位置。

·Aggregate：聚合，定义一个创建迭代器对象的接口。

·ConcreteAggregate：具体聚合，实现创建迭代器对象，返回一个具体迭代器的实例。

迭代器支持在聚集中移动游标，使得访问聚合中的元素变得简单，简化了聚集的接口，分装了聚合的对象。迭代器模式还可以应用于树形结构的访问，程序不需要从头行代码查找相应的位置，可控制到从子集开始查找，对于加快程序的运行速度有很重要的作用。缺点是聚合密切相关，增加了耦合，但是将这种耦合定义在抽象类，可以解决这种问题。

5）Mediator（中介者模式）

中介者模式的意图是：用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

中介者模式的适用性如下：

·一组对象以定义良好但是复杂的方式进行通信。产生的相互依赖关系结构混乱且难以理解。

·一个对象引用其他很多对象并且直接与这些对象通信,导致难以复用该对象。想定制一个分布在多个类中的行为，而又不想生成太多的子类。

中介者模式的结构（UML类图）如图6-21所示。中介者模式使得各个对象之间不需要直接打交道，从而使他们松散耦合。可以独立的改变他们之间的交互。

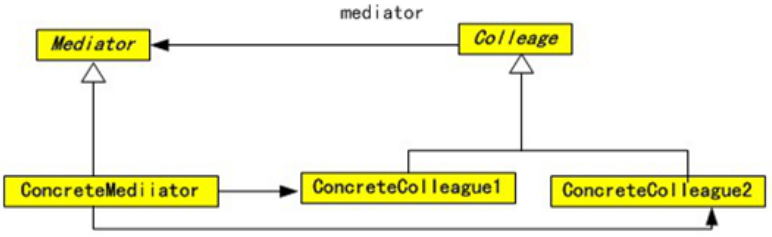

图6-21 Mediator（中介者模式）的结构类图

其中类和对象的关系如下：

·Mediator：抽象中介者角色定义统一的接口用于各同事角色之间的通信。

·ConcreteMediator：具体中介者，协调各个同事对象之间实现协作的行为，掌握并且维护它的各个同事对象引用。

·Colleague：同事类，每一个同事角色都知道对应的具体中介者角色，而且与其他的同事角色通信的时候，一定要通过中介者角色协作。

中介者模式的优点：把中介者作为一个独立的概念，可使对象之间的关系转为对象和中介者之间的关系，可站在一个宏观的角度看问题。

中介者模式的缺点：中介者集中控制关系，使得中介者变得复杂。

6）Memento（备忘录模式）

备忘录模式的意图是：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。这样以后就可将该对象恢复到原先保存的状态。

备忘录模式的适用性如下：

·必须保存一个对象在某一个时刻的（部分）状态, 这样以后需要时它才能恢复到先前的状态。

·如果一个用接口来让其它对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。

备忘录模式的结构（UML类图）如图6-22所示。备忘录对象是一个用来存储另外一个对象内部

状态的快照的对象。

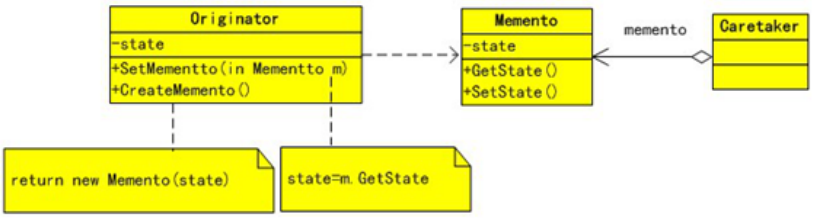

图6-22 Memento（备忘录模式）的类图

其中类和对象的关系如下：

·Memento：备忘录对象，保持Originator（原发器）的内部状态，根据原发器来决定保存哪些内部的状态。保存原发器之外的对象访问备忘录。备忘录可以有效地利用两个接口。看管者只能调用狭窄（功能有限）的接口——它只能传递备忘录对象给其他对象。而原发器能调用一个宽接口（功能强大的接口），通过这个接口可以访问所有需要的数据，使原发器可以返回原先的状态。理想的情况是，只允许生成本备忘录那个原发器访问本备忘录的状态。

·Originator：原发器，通常是需要备忘的对象自己，创建一个备忘录，记录它的当前内部状态。可以利用一个备忘录来恢复它的内部状态。

·CareTaker：备忘录管理者，只负责看管备忘录，不可以对备忘录的内容操作或者检察。

Memento保存了封装的边界，一个Memento对象是另一种原发器对象的表示，不会被其他对象改动。同时将特定状态的行为局部化，并将不同的状态的行为分割开，把状态的变迁分散转移到各个子类中去。

7）Observer（观察者模式）

观察者模式的意图是：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被自动更新。

观察者模式的适用性如下：

·当一个抽象模型有两个方面，其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。

·当对一个对象的改变需要同时改变其它对象, 而不知道具体有多少对象有待改变。

·当一个对象必须通知其它对象，而它又不能假定其它对象是谁。换言之，你不希望这些对象是紧密耦合的。

观察者模式的结构（UML类图）如图6-23所示。观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使他们能够自动更新自己。

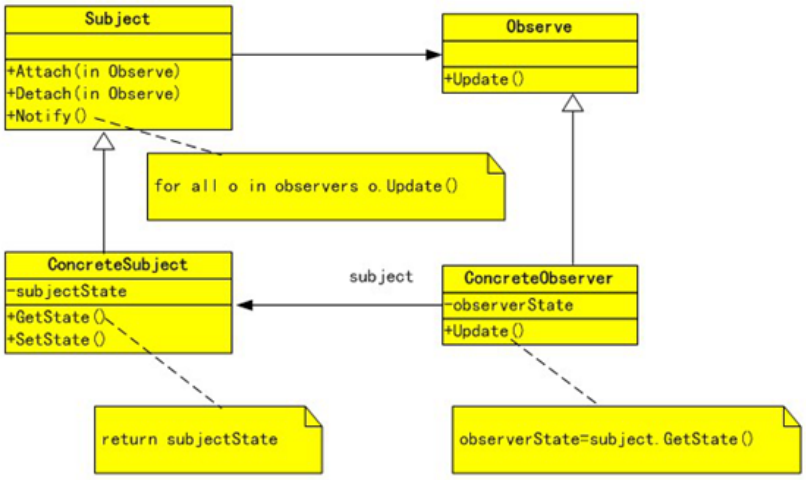

图6-23 Observer（观察者模式）的类图

其中类和对象的关系如下：

·Subject：被观察对象，了解其多个观察者，任意数量的观察者可以观察一个对象，提供一个接口用来绑定以及分离观察者对象。

·Concrete Subject：具体被观察对象，存储具体观察者，Concrete Observer有兴趣的状态。当其状态改变时，发送一个通知给其所有的观察者对象。

·Observer：观察者，定义一个更新接口，在一个被观察对象改变时应被通知。

·Concrete Observer：具体观察者，维护一个对Concrete Subject对象的引用。

观察者模式抽象了被观察者和观察者对象的连接，作用在于解耦， 就是让耦合的双方都依赖于抽象而不是具体，从而使得各部分的变化都不会影响到对方，容易增加新的观察者对象。观察者模式的缺陷是：对象间的关系难以理解，在某种情况下会表现低效能。

8）State（状态模式）

状态模式的意图是：允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

状态模式的适用性如下：

·对象的行为依赖于它的状态（属性），并且它必须可以根据它的状态而改变它的行为。

·操作很多部分都带有与对象状态有关的大量条件语句。

状态模式的结构（UML类图）如图6-24所示。状态模式把所研究的对象的行为包装在不同的状态对象里，每一个状态对象都属于一个抽象状态类的一个子类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。状态模式需要对每一个系统可能取得的状态创立一个状态类的子类。当系统的状态变化时，系统便改变所选的子类。

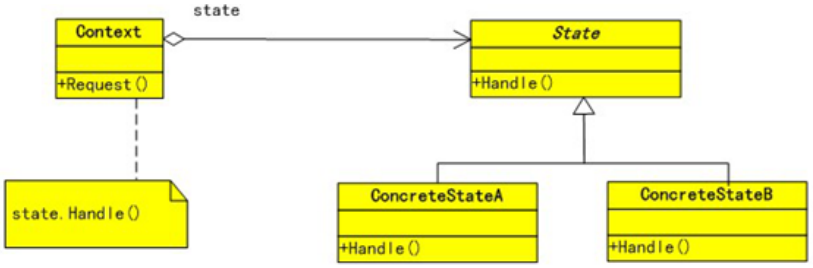

图6-24 State（状态模式）的类图

其中类和对象的关系为：

·Context：情景类,定义客户应用程序有兴趣的接口,维护一个Concrete State（具体状态）子类的实例对象。

·State：抽象状态类,定义一个接口用来封装与Context的一个特别状态（State）相关的行为。

·Concrete State：具体状态类,每一个具体状态类（Concrete State）实现了一个Context的状态（State）相关的行为。

状态模式在对象内保存特定的状态，并且就不同的状态履行不同的行为，它使状态的变化显得清晰了，也很容易创建对象的新状态。状态模式在工作流或游戏等各种系统中有大量的应用。

9）Strategy（策略模式）

策略模式的意图是：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

策略模式的适用性如下：

·多个类的分别只在于行为不同

·需要对行为算法做很多变动

·客户不需要知道算法使用的数据

策略模式的结构（UML类图）如图6-25所示。

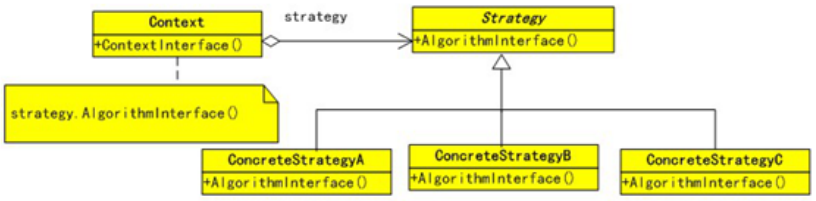

图6-25 Strategy（策略模式）的类图

其中类和对象的关系为：

·Strategy：抽象策略类,定义一个借口给所有支持的算法，Context使用这个接口调用ConcreteStrategy定义的算法。

·ConcreteStrategy：具体策略类,用ConcreteStrategy对象配置其执行环境。

策略模式提供了替代派生的子类，并定义类的每个行为，把容易发生变化的算法独立出来，易于扩展，系统变得更加灵活。应用策略模式会产生很多子类，这符合高内聚的责任分配模式，类增多了，使系统难度加大。

10）Template Method（模板方法模式）

模板方法模式的意图是：定义一个操作中算法的骨架，以将一些操作延缓到子类中实现，模板方法让子类重新定义一个算法的某些步骤，而无需改变算法结构。

模板方法模式的适用性如下：

·一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。

·控制子类扩展。

模板方法模式的结构（UML类图）如图6-26所示。

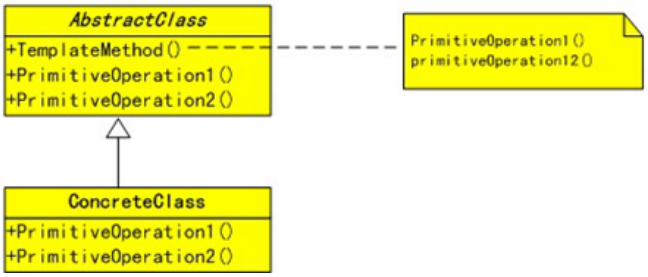

图6-26：Template Method（模板方法模式）的类图

其中类和对象的关系如下：

·AbstractClass：抽象类,定义一个抽象原始的操作，其子类可以重新定义它实现一个算法的各个步骤。实现一个模板方法定义一个算法的骨架，此模板方法不仅可以调用原始的操作，还可以调用定义于AbstractClass中的方法或者其他对象中的方法。

·ConcreteClass：具体子类,实现原始的操作以完成子类特定算法的步骤。

模板方法模式在一个类中形式化的定义算法，而由他的子类实现细节的处理，模板方法模式的优势是把公共的代码或不变的行为放在超类/模版中， 以隔离变化和不变，使不便的部分可复用。模板方法的特点在于：每个不同的实现都需要定义一个子类，这也符合高内聚的设计模式。

11）Visitor（访问者模式）

访问者模式的意图是：表示一个作用于某对象结构中的各元素的操作。它使你可以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。

访问者模式的适用性如下：

·一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作。

·需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而你想避免让这些操作“污染”这些对象的类。Visitor使得你可以将相关的操作集中起来定义在一个类中。当该对象结构被很多应用共享时，用Visitor模式让每个应用仅包含需要用到的操作。

·定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。改变对象结构类需要重定义对所有访问者的接口，这可能需要很大的代价。如果对象结构类经常改变，那么可能还是在这些类中定义这些操作较好。

访问者模式的结构（UML类图）如图6-27所示。访问者模式在不改变一个已存在的类的层次结构的情况下，为这个层次结构中的某些类定义一个新的操作，这个新的操作作用于（操作）已存在的类对象，也即新的操作需要访问被作用的对象。

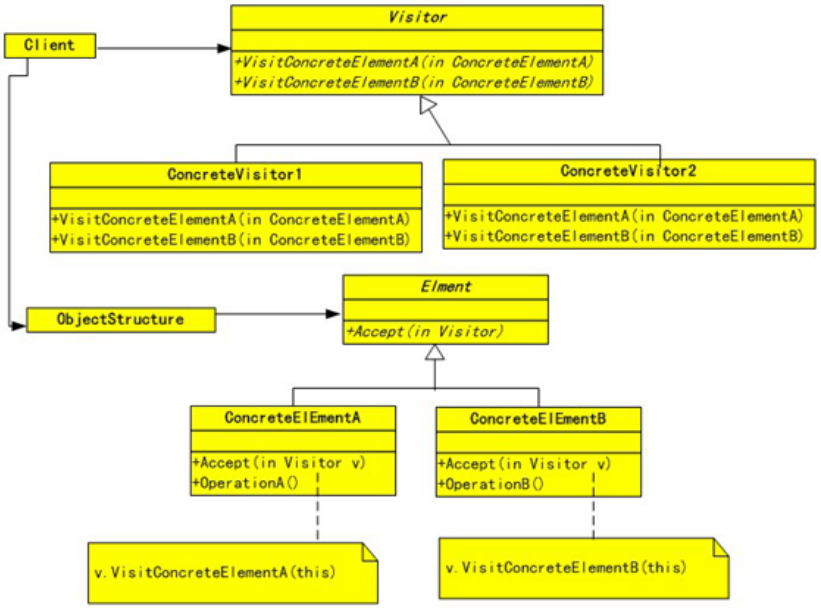

图6-27 Visitor（访问者模式）的类图

其中类和对象的关系为：

·Visitor：抽象访问者，为对象结构类中每一个ConcreteElement的类声明一个Visit操作。这个操作的名称以及标志识别传出Visit请求给访问者的类。这就使得访问者可以解定正要被访问的元素的具体类，这样访问这就可以直接经由其特有接口访问到元素。

·ConcreteVisitor：具体访问者，实现每个由Visitor声明的操作。每个操作实现本算法的一部分，而该算法片断乃是对应于结构中对象的类。ConcreteVisitor为该算法提供了场景并存储他的局部状态。这一状态常常在遍历该结构的过程中积累结果。

·Element：元素，定义一个Accept操作，它一个访问者为参数。

·ConcreteElement：抽象具体元素，实现Accept操作，该操作以一个访问者为参数。

·ObjectStructure：对象结构类，能枚举他的元素，可以提供一个高层的接口以允许访问者访问它的元素，可以是一个组合模式或者是一个集合，如一个列表或一个无序集合。

Java实现访问者模式代码如下。

//访问者接口

```
public interface Visitor {
    public void isitConcreteElementA( ConcreteElementA e );
    public void visitConcreteElementB( ConcreteElementB e );
}
//具体访问者1
public class ConcreteVisitor1 implements Visitor{
    public void visitConcreteElementA( ConcreteElementA e ) {
        System.out.println( "访问者1访问具体元素A" );
    }
    public void visitConcreteElementB( ConcreteElementB e ) {
        System.out.println( "访问者1访问具体元素B." );
    }
}
//具体访问者2
public class ConcreteVisitor2 implements Visitor{
    public void visitConcreteElementA( ConcreteElementA e ) {
        System.out.println( "访问者2访问具体元素A" );
    }
      public void visitConcreteElementB( ConcreteElementB e ) {
        System.out.println( "访问者2访问具体元素B" );
    }
}
//类结构借口
public interface Element {
    public void accept( Visitor v );
}
//实体类A
public class ConcreteElementA implements Element{
    public void accept( Visitor v ) {
        v.OperationA( this );
    }
}
//实体类B
public class ConcreteElementB implements Element{
    public void accept( Visitor v ) {
        v.OperationB( this );
    }
}
//对象结构
```

```
public class ObjectsStructure {
    List<Element> list = new ArrayList( );
    public ObjectsStructure( ){
        list.add( new ConcreteElementA( ) );
        list.add( new ConcreteElementB( ) );
    }
    public void process( Visitor v ){
        for( Element e : list ){
            e.accept( v );
        }
    }
    public static void main( String[] args ){
        ObjectsStructure o = new ObjectsStructure( );
        o.process( new ConcreteVisitor1( ) );
        o.process( new ConcreteVisitor2( ) );
    }
}
```

Visitor模式使得增加新的操作变得容易，它可以收集有关联的方法，而分离没有关联的方法，特别适用于分离因为不同原因而变化的事物，但是Visitor模式常常要打破对象的封装，visitor同element需要达成某些共识。

版权方授权希赛网发布，侵权必究

第 6 章：程序设计与设计模式　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

典型真题解析

本节从历年考试真题中，精选出5道典型的试题进行分析，这5道试题所考查的知识点基本上覆盖了本章的所有内容，非常具有代表性。

版权方授权希赛网发布，侵权必究

第 6 章：程序设计与设计模式　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

例题1

现欲构造一文件/目录树，采用组合（Composite）设计模式来设计，得到的类图如6-28所示：

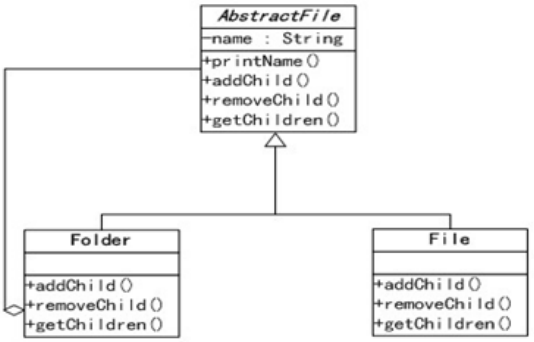

图6-28 类图

【Java代码】

```
import Java.util.ArrayList;
import java.util.List;
 （1） class AbstractFile {
    protected String name;
    public void printName( ) { System.out.println( name ); }
    public abstract boolean addChild( AbstractFile file );
    public abstract boolean removeChild( AbstractF ile file );
    public abstract List<AbstractFile> getChildren( );
 }
class File extends AbstractFile {
    public File( String name ) { this.name = name; }
    public boolean addChild( AbstractFile file ) { return false; }
    public boolean removeChild( AbstractFile file ) { return false; }
    public List<AbstractFile> getChildren( ) { return （2） ; }
 }
class Folder extends AbstractFile {
    private List <AbslractFile> childList;
    public Folder( String name ) {
          this.name = name;
        this.childList = new ArrayList<AbstractFile>( );
       }
    public boolean addChild( AbstractFile file ) { return childList . add( file ); }
    public boolean removeChild( AbstractFile file ) { return childList . remove( file ); }
    public （3） <AbstractFile> getChildren( ) { return （4） ; }
 }
public class Client {
    public static void main( String[] args ) {
        //构造一个树形的文件 / 目录结构
        AbstractFile rootFolder =  new Folder( "c:\\ " );
        AbstractFile compositeFolder = new Folder( "composite" );
```

```
        AbstractFile windowsFolder = new Folder( "windows" );
        AbstractFile file = new File( "TestComposite.java" );
        rootFolder.addChild( compositeFolder ) ;
        rootFolder.addChild( windowsFolder );
        compositeFolder.addChild( file ) ;
        //打印目录文件树
        printTree( rootFolder );
    }
    private static void printTree( AbslractFile ifile ) {
        ifile.printName( );
        List <AbslractFile> children = ifile . getChildreno :
        if( children = =null ) return;
        for ( AbstractFile ..file:children ) {
            （5）;
        }
    }
}
```

该程序运行后输出结果为：

c:\

composite

TestComposite.java

Windows

例题1分析

本题考查基本面向对象设计模式的运用能力。

组合设计模式主要是表达整体和部分的关系，并且对整体和部分对象的使用无差别。由UML结构图知AbstractFile是File类和Folder类的父类，它抽象了两个类的共有属性和行为。在使用中，不论是File对象还是Folder对象，都可被当作AbstractFile对象来使用。另外由UML结构图知，类Folder和AbstractFile之间存在共享关系，Folder对象可以共享其它的Folder对象和File对象。题目中AbstractFile类应该为抽象类，因此空（1）应填abstract.

由于File对象没有孩子节点，所以空（2）应填null。getChildren( )方法继承自AbstractFile类，因此其返回类型也应保持一致性，故空（3）应填List。对于空（4），返回Folder对象的孩子节点，因此返回其成员childList的地址，应填childList. 该程序的运行能够打印出文件目录树，故空（5）应该为打印方法调用，应填printTree(file)。

例题1参考答案

（1）abstract

（2）null

（3）List

（4）childList

（5）printTree(file)

版权方授权希赛网发布，侵权必究

例题2

现欲实现一个图像浏览系统，要求该系统能够显示BMP、JPEG 和GIF三种格式的文件，并且能够在Windows和Linux两种操作系统上运行。系统首先将BMP、JPEG 和GIF三种格式的文件解析为像素矩阵，然后将像素矩阵显示在屏幕上。系统需具有较好的扩展性以支持新的文件格式和操作系统。为满足上述需求并减少所需生成的子类数目，采用桥接（Bridge）设计模式进行设计所得类图如图6-29所示。

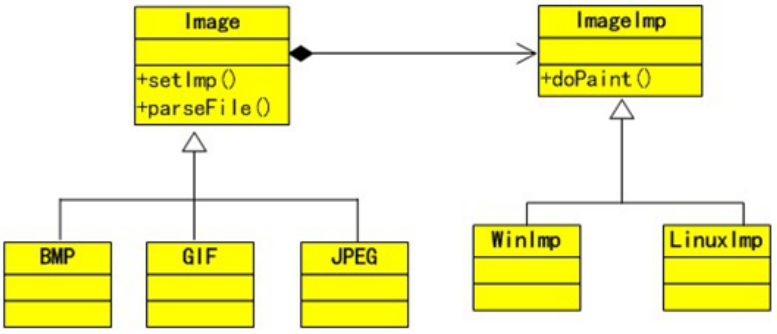

图6-29 类图

采用该设计模式的原因在于：系统解析BMP、GIF与JPEG文件的代码仅与文件格式相关，而在屏幕上显示像素矩阵的代码则仅与操作系统相关。

【C++代码】

```
class Matrix{   //各种格式的文件最终都被转化为像素矩阵
   //此处代码省略
   };
class ImageImp{
   public:
   virtual void doPaint( Matrix m ) = 0;    //显示像素矩阵m
   };
class WinImp : public ImageImp{
   public:
   void doPaint( Matrix m ){ /*调用windows系统的绘制函数绘制像素矩阵*/ }
   };
class LinuxImp : public ImageImp{
   public:
   void doPaint( Matrix m ){ /*调用Linux系统的绘制函数绘制像素矩阵*/  }
   };
class Image {
   public:
```

```
    void setImp( ImageImp *imp ){  （1）     = imp;}
    virtual void parseFile( string fileName ) = 0;
    protected:
    （2）    *imp;
    };
class BMP : public Image{
    public:
    void parseFile( string fileName ){
    //此处解析BMP 文件并获得一个像素矩阵对象m
    （3）  ;// 显示像素矩阵m
    }
    };
class GIF : public Image{
    //此处代码省略
    };
class JPEG : public Image{
    //此处代码省略
    };
void main( ){
    //在windows操作系统上查看demo.bmp图像文件
    Image *image1 =    （4）  ;
    ImageImp *imageImp1 =   （5）  ;
     （6）  ;
    image1->parseFile( "demo.bmp" );
    }
```

现假设该系统需要支持10种格式的图像文件和5种操作系统，不考虑类Matrix，若采用桥接设计模式则至少需要设计 （7） 个类。

例题2分析

本题考查基本面向对象设计模式的运用能力。

由文字描述和UML结构图知BMP、GIF与JPEG是Image的子类，分别负责读取不同格式的文件。ImageImp的主要任务是将像素矩阵显示在屏幕上，它的两个子类WinImp 、 LinuxImp分别实现windows系统和Linux系统上的图像显示代码。空（1）处主要设置在哪个平台上进行实现，由于该类的成员变量也是imp,与参数相同，因此应填this->imp。同理，该成员变量的类型和参数的类型应保持相同，故空（2）处应填ImageImp。空（3）处需要根据imp成员变量存储的实现对象来显示图像，应填imp->doPaint(m);在空（4）处需要生成一个BMP对象，故应填new BMP()，在空（5）处需要生成一个WinImp对象,故应填new WinImp()，空（6）处应填image1->setImp(ImageImp1)，采用Bridge（桥接模式）能够将文件分析代码和图像显示代码分解在不同的类层结构中，如果不考虑Matrix等类，那么最后需要设计的类包括2个父类，分别为文件格式子类和操作系统平台类，故系统需要支持10种格式的图像文件和5种操作系统至少需要17个类。

例题2参考答案

（1）this->imp

（2）ImageImp

（3）imp->doPaint(m)

（4）new BMP()

（5）new WinImp()

（6）image1->setImp(ImageImp1)

（7）17

版权方授权希赛网发布，侵权必究

例题3

已知某类库开发商提供了一套类库，类库中定义了Application类和Document类，它们之间的关系如图6-30所示，其中，Application 类表示应用程序自身，而Document类则表示应用程序打开的文档。Application类负责打开一个已有的以外部形式存储的文档，如一个文件，一旦从该文件中读出信息后，它就由一个Document对象表示

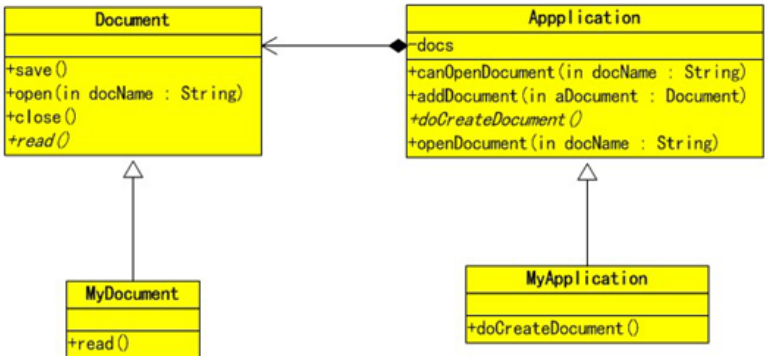

图 6-30　Application与Document关系图

当开发一个具体的应用程序时，开发者需要分别创建自己的Application和Document子类，例如图6-30中的类MyApplication和类MyDocument，并分别实现Application和Document类中的某些方法。

已知Application类中的openDocument方法采用了模板方法（Template Method）设计模式，该方法定义了打开文档的每一个主要步骤，如下所示：

1．首先检查文档是否能够被打开，若不能打开，则给出出错信息并返回；

2．创建文档对象；

3．通过文档对象打开文档；

4．通过文档对象读取文档信息；

5．将文档对象加入到Application的文档对象集合中。

【Java 代码】

```
abstract class Document{
```

```
    public void save( ){ /*存储文档数据，此处代码省略*/ }
    public void open(String docName){    /* 打开文档，此处代码省略  */ }
    public void close( ){ /* 关闭文档，此处代码省略*/    }
    public abstract void read(String docName );
  };
  abstract class Appplication{
    private  Vector <   ( 1 )   >    docs; /*文档对象集合*/
    public  boolean canOpenDocument( String docName ){
    /*判断是否可以打开指定文档，返回真值时表示可以打开，返回假值表示不可打开，此处代码省略*/
    }
  public void addDocument( Document aDocument ){
      /*将文档对象添加到文档对象集合中*/
      docs.add(   ( 2 )  );
    }
    public abstract Document doCreateDocument( ); /*创建一个文档对象*/
  public void openDocument( String docName ){ /*打开文档*/
      if (  ( 3 )  ){
        System.out.println( "文档无法打开!" );
        return;
      }
       ( 4 )   adoc =    ( 5 )   ;
      ( 6 )   ;
    ( 7 )   ;
        ( 8 )   ;
    }
  };
```

例题3分析

本题考查了Java语言的应用能力和模板方法设计模式。空（1）考查了Java库中Vector模板类的使用，由于Vector模板类可以存储任意类型，在定义时需要指定其存储类型，根据后面的代码，能够加入到该文档集合对象的类型为文档类型，因此空（1）处的类型应该为Document。空（2）处将文档对象加入文档集合对象中。从空（3）开始的代码属于图中Application类的OpenDocument方法，该方法是模板方法，因此，需根据题目给出的步骤一一对应填空。空（3）处判断能否打开文档，需要调用父类自己的方法canOpen- Document。空（4）与空（5）所在的语句需要创建文档对象，调用doCreateDocument方法，接着通过文档对象打开和读取文档，最后通过addDocument方法将该文档对象加入到文档对象集合中。所有这些方法都是在父类或文档对象中进行定义，不涉及到具体的子类。而子类负责要实现这些模板方法中需要调用的方法以便运行时被调用。

例题3参考答案

（1）Document

（2）aDocument

（3）!canOpenDocument(docName)

（4）Document

（5）doCreateDocument( )

（6）adoc.open(docName)

（7）adoc.read(docName)

（8）addDocument(adoc)

版权方授权希赛网发布，侵权必究

例题4

已知某企业欲开发一家用电器遥控系统，即用户使用一个遥控器即可控制某些家用电器的开与关。遥控器如图 6-31 所示。该遥控器共有 4 个按钮，编号分别是 0 至 3，按钮 0和 2 能够遥控打开电器 1 和电器 2，按钮 1 和 3 则能遥控关闭电器 1 和电器 2。由于遥控系统需要支持形式多样的电器，因此，该系统的设计要求具有较高的扩展性。现假设需要控制客厅电视和卧室电灯，对该遥控系统进行设计所得类图如图6-32所示。

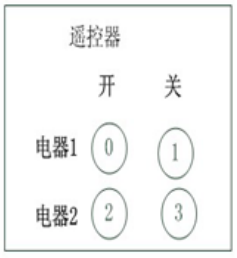

图 6-31遥控器

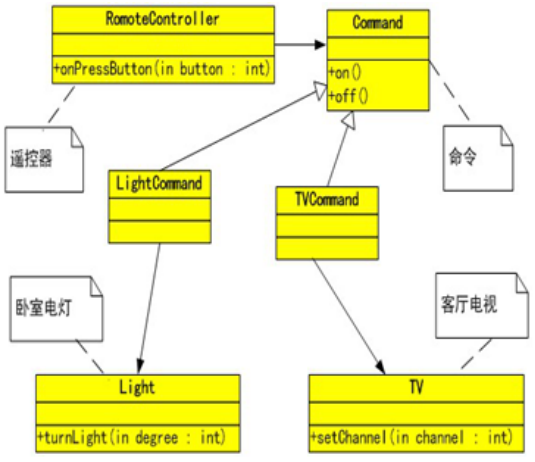

图6-32 设计类图

在图6-32中，类RomoteController的方法onPressButton(int button)表示当遥控器按键按下时调用的方法，参数为按键的编号；Command 接口中 on 和 off 方法分别用于控制电器的开与关；Light中turnLight(int degree)方法用于调整电灯灯光的强弱，参数degree值为0时表示关灯，

值为100时表示开灯并且将灯光亮度调整到最大；TV 中setChannel(int channel)方法表示设置电视播放的频道，参数 channel 值为 0 时表示关闭电视，为 1 时表示开机并将频道切换为第1频道。

【C++代码】

```
class Light{    //电灯类
public:
void turnLight( int degree ){ //调整灯光亮度，0表示关灯，100表示亮度最大};
};
class TV{ //电视机类
public:
void setChannel( int channel ){//调整频道，0表示关机，1表示开机并切换到1频道};
};
class Command{ //抽象命令类
public:
virtual void on( )= 0;
virtual void off( )= 0;
};
class RemoteController{    //遥控器类
protected:
Command *commands[4]; //遥控器有4个按钮，按照编号分别对应4个Command对象
public:
void onPressButton( int button ){    //按钮被按下时执行命令对象中的命令
  if( button % 2 = = 0 )commands[button]->on( );
  else commands[button]->off( );
}
void setCommand( int button,Command * command ){
（1）    = command; //设置每个按钮对应的命令对象
}
};
class LightCommand : public Command{    //电灯命令类
protected:  Light *light;    //指向要控制的电灯对象
public:
void on( ){light->turnLight( 100 );};
void off( ){light-> （2）  ;};
LightCommand( Light * light ){this->light = light;};
};
class TVCommand : public Command{ //电视机命令类
protected:  TV * tv;      //指向要控制的电视机对象
public:
void on( ){tv->  （3）  ;};
```

```
void off( ){tv->setChannel( 0 );};
TVCommand( TV * tv ){ this->tv = tv; };
};
void main( ){
Light light;  TV tv; //创建电灯和电视对象
LightCommand lightCommand( &light );
TVCommand tvCommand( &tv );
RemoteController remoteController;
remoteController.setCommand( 0, （4）  );   //设置按钮0的命令对象
…//此处省略设置按钮1、按钮2和按钮3的命令对象代码
}
```

本题中，应用命令模式能够有效让类（5）、类（6）和类（7）之间的耦合性降至最小。

例题4分析

本题考查的是设计模式中的命令模式。

设计时，为了保证遥控器和家用电器之间的独立性，定义了Command类，当用户按下遥控器上的按钮时，触发Command上的On或者Off方法，因此，一对按钮分别对应一个Command对象。题目中的LightCommand以及与TVCommand分别为Command的子类，该子类用于控制实际的Light以及TV对象，将On与Off方法委托给Light以及TV实现。空（1）表示要设置遥控器上按钮控制的对象，其参数传递的是某一个命令对象，因此只需将该命令对象存储下来即可；空（2）表示关闭电灯，根据说明，关闭电灯的方法为turnLight（0）；空（3）表示打开电视机，因此需要调用打开电视的方法。空（4）表示将按钮0和相应的Command对象相关联，根据题目描述，按钮0用于控制灯或者电视，因此，应该设置灯或者电视的命令对象。本题中应用命令模式的目的是为了使为了让类遥控器和类Light与TV之间的耦合性降至最低。

例题1参考答案

（1）commands[button]

（2）trunLight(0)

（3）setChannel(1)

（4）&lightCommand

（5）RemoteController

（6）Light

（7）TV

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

例题5

已知某企业的采购审批是分级进行的，即根据采购金额的不同由不同层次的主管人员来审批，主任可以审批5万元以下（不包括5万元）的采购单，副董事长可以审批5万元至10万元（不包括10万元）的采购单，董事长可以审批10万元至50万元（不包括50万元）的采购单，50万元及以上的采购单就需要开会讨论决定。

采用责任链设计模式（Chain of Responsibility）对上述过程进行设计后得到的类图如图6-33所示。

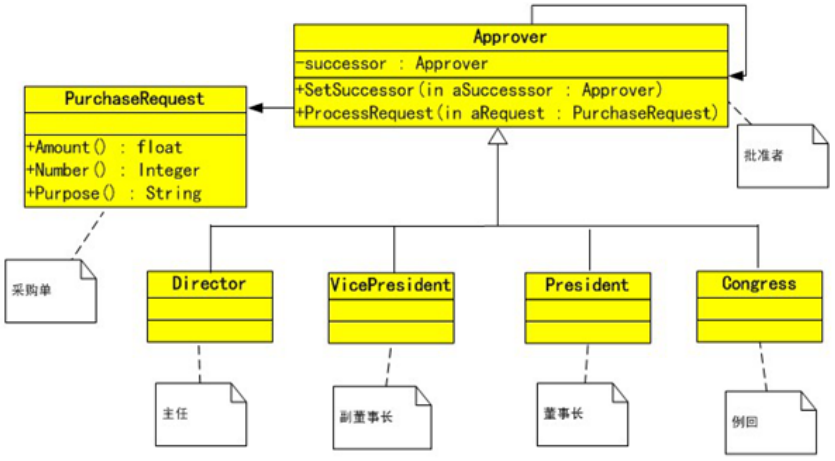

图6-33 设计类图

【Java代码】

```
class PurchaseRequest {
public  double  Amount;   // 一个采购的金额
    public   int  Number;      // 采购单编号
    public   String  Purpose;   // 采购目的
 };
class Approver {      // 审批者类
    public Approver( ){  successor = null;    }
    public void ProcessRequest( PurchaseRequest aRequest ){
          if( successor != null ){   successor.  （1）  ;   }
     }
    public void SetSuccessor( Approver aSuccesssor ){ successor = aSuccesssor; }
    private  （2）   successor;
 };
class Congress extends Approver {
    public void ProcessRequest( PurchaseRequest aRequest ){
           if( aRequest.Amount>= 500000 ){ /* 决定是否审批的代码省略 */  }
           else   （3）  .ProcessRequest( aRequest );
     }
};
class Director extends Approver {
    public void ProcessRequest( PurchaseRequest aRequest ){ /* 此处代码省略*/ }
 };
```

```
class President extends Approver {
    public void ProcessRequest( PurchaseRequest aRequest ){ /* 此处代码省略*/ }
};
class VicePresident extends Approver {
    public void ProcessRequest( PurchaseRequest aRequest ){ /* 此处代码省略*/ }
};
public class rs {
    public static void main( String[] args ) throws IOException {
        CongressMeeting = new Congress( );
        VicePresidentSam = new VicePresident( );
        DirectorLarry = new Director( );
        PresidentTammy = new President( );
        //构造责任链
        Meeting.SetSuccessor( null );
Sam.SetSuccessor( （4） );
        Tammy.SetSuccessor( （5） );
Larry.SetSuccessor( （6） );
        // 构造一采购审批请求
        PurchaseRequestaRequest = new PurchaseRequest( );
        BufferedReader br = new BufferedReader( newInputStreamReader( System.in ) );
        aRequest.Amount = Double.parseDouble( br.readLine( ) );
            （7） .ProcessRequest( aRequest );       // 开始审批
        return;
    }
}
```

例题5分析

本题考查的是责任链设计模式（Chain of Responsibility）的应用，责任链设计模式属于行为型模式。其目的是为了将一个请求发送给一个对象集合，对象被组织成一条链，而负责处理该请求的对象将获取请求消息并处理，其余对象则仅仅负责将该请求消息按照责任链的顺序传递到下一个对象。故责任链设计模式的关键在于组织不同的对象成为一条链并传递消息。

代码中空（1）处位于条件判断if(successor != null)内，其含义是判断当前对象是否存在后继对象，如果存在，按照责任链设计模式，可以把请求消息进行传递，即调用ProcessRequest方法，应填ProcessRequest(aRequest)。空（2）处要求填写successor的类型，代码中voidSetSuccessor(Approver *aSuccesssor){ successor = aSuccesssor; }给出了successor的类型为Approver，故应填Approver。空（3）处位于Congress类的ProcessRequest方法中，该方法表示处理外界的请求，else块的含义表明Congress对象不处理审批金额大于50万元的请求，会将请求转发到下一个对象进行处理，即直接调用父类的ProcessRequest方法，应填super。空（4）、（5）、（6）主要用来将各种对象串成一个链，根据题目的要求，对象在责任链中的顺序应该为Director Larry 、VicePresident Sam、President Tammy、Congress Meeting，而审批的请求应

该从Larry开始。而SetSuccessor方法的参数应为Approver对象或继承Approver的子对象，

例题5参考答案

（1）ProcessRequest(aRequest)

（2）Approver

（3）super

（4）Tammy

（5）Meeting

（6）Sam

（7）Larry

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

考前必练

近年来，下午试题中的面向对象考题模式非常稳定，主要就是考的设计模式+程序语言基本语法。这种类型的题需要对所考查的设计模式有一定的了解。本小节从历年考试试题中，精选出6道典型的试题（3道Java，3道C++）作为考生考前练习，后面给出了解题分析和答案。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

试题1

考前必做的练习题

某软件公司现欲开发一款飞机飞行模拟系统，该系统主要模拟不同种类飞机的飞行特征与起飞特征。需要模拟的飞机种类及其特征如表6-6所示。

表6-6 模拟的飞机种类及其特征表

飞机种类	起飞特征	飞行特征
直升机（Helicoper）	垂直起飞（VerticalTakeOff）	亚音速飞行（SubSonicFly）
客机（ AirPlane）	长距离起飞（LongDistanceTakeOff）	亚音速飞行（SubSonicFly）
歼击机（ Fighter）	长距离起飞（LongDistanceTakeOff）	超音速飞行（SuperSonicFly）
鹞式战斗机（Harrier）	垂直起飞（VerticalTakeOff）	超音速飞行（SuperSonicFly）

为支持将来模拟更多种类的飞机，采用策略设计模式（Strategy）设计的类图如图6-34所示。

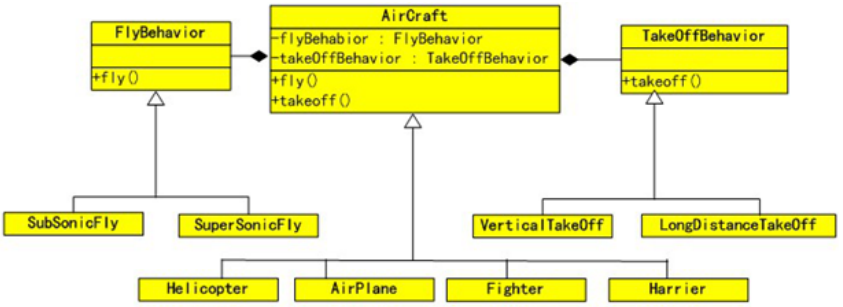

图6-34 类图

在图6-34中，AirCraft为抽象类，描述了抽象的飞机，而类Helicopter、AirPlane、Fighter和Harrier分别描述具体的飞机种类，方法fly( )和takeOff( )分别表示不同飞机都具有的飞行特征和起飞特征，类FlyBehavior与TakeOffBehavior为抽象类，分别用于表示抽象的飞行行为与起飞行为；类SubSonicFly与SuperSonicFly分别描述亚音速飞行和超音速飞行的行为；类VerticalTakeOff与LongDistanceTakeOff分别描述垂直起飞与长距离起飞的行为。

【Java 代码】

```
interface FlyBehavior {
    public void fly( );
};
class SubSonicFly implements FlyBehavior{
    public void fly( ){ System.out.println( "亚音速飞行!" ); }
};
class SuperSonicFly implements FlyBehavior{
    public void fly( ){ System.out.println( "超音速飞行!" ); }
};
interface TakeOffBehavior {
    public void takeOff( );
};
class VerticalTakeOff implements TakeOffBehavior {
    public void takeOff ( ){ System.out.println( "垂直起飞!" ); }
};
class LongDistanceTakeOff implements TakeOffBehavior {
    public void takeOff( ){ System.out.println( "长距离起飞!" ); }
};
abstract class AirCraft {
    protected （1）;
    protected （2）;
 public void fly( ){ （3）; }
    public void takeOff( ) { （4） ; };
};
class Helicopter （5） AirCraft{
    public Helicopter ( ){
```

```
        flyBehavior = new （6）;
        takeOffBehavior = new （7）;
    }
};
//其它代码省略
```

版权方授权希赛网发布，侵权必究

试题2

现欲构造一文件 / 目录树，采用组合（Composite）设计模式来设计，得到的类图如6-35所示：

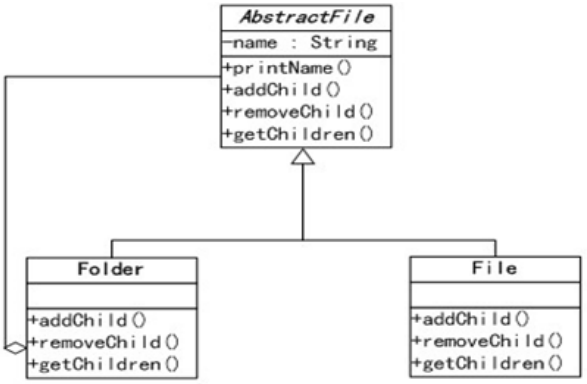

图6-35 类图

【C++代码】

```
#include <list>
#include <iostream>
#include <string>
using namespace std;
class AbstractFile {
protected :
    string name;   //文件或目录名称
public :
    void printName( ){cout<<name;}   //打印文件或目录名称
    virtual void addChild( AbstractFile *file )= 0;  //给一个目录增加子目录或文件 }
    virtual void removeChild( AbstractFile *file )= 0;  //删除一个目录的子目录或文件
    virtual list<AbstractFile*> *getChildren( )= 0;  //获得一个目录的子目录或文件
 } ;
class File: public AbstractFile {
public:
```

```
    File( string name ) { （1） = name; }
    void addChild( AbstractFile *file ) { return; }
    void removeChild( AbstractFile *file ) { return; }
    （2） getChildren( ) { return （3） ; }
};
class Folder:public AbstractFile {
private:
    list <AbstractFile*> childList:  //存储子目录或文件
public:
    Folder( string name ){ （4） name; }
    void addChild( AbstractFile*file ) { childList.push_back( file ); }
    void removeChild( AbstractFile*file ) { childList.remove( file ); }
    list<AbstractFile*>*getChildren( ) { return （5） ; }
};
void main( ) {
    //构造一个树形的文件 / 目录结构
    AbstractFile *rootFolder = new Folder( "c:\\ " );
    AbstractFile*compositeFolder = new Folder( "composite" );
    AbstractFile *windowsFolder = new Folder( "windows" );
    AbstractFile*file = new File( "TestCompositejava" );
    rootFolder->addChild( compositeFolder );
    rootFolder->addChild( windowsFolder );
    compositeFolder->addChild( file );
}
```

版权方授权希赛网发布，侵权必究

　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

试题3

现欲实现一个图像浏览系统，要求该系统能够显示BMP、JPEG 和GIF三种格式的文件，并且能够在Windows和Linux两种操作系统上运行。系统首先将BMP、JPEG 和GIF三种格式的文件解析为像素矩阵，然后将像素矩阵显示在屏幕上。系统需具有较好的扩展性以支持新的文件格式和操作系统。为满足上述需求并减少所需生成的子类数目，采用桥接（Bridge）设计模式进行设计所得类图如图6-36所示。

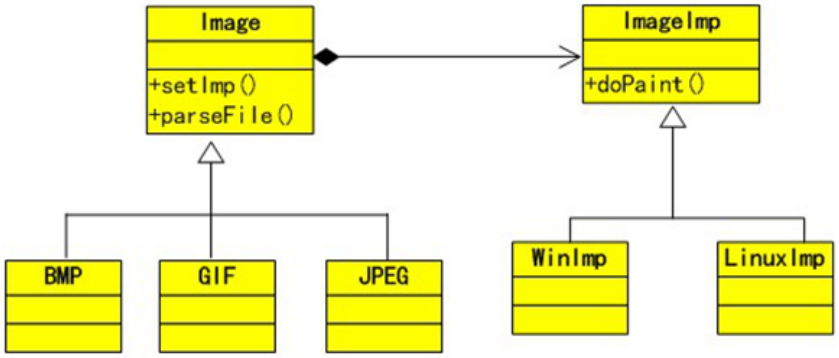

图6-36 类图

采用该设计模式的原因在于：系统解析BMP、GIF与JPEG文件的代码仅与文件格式相关，而在屏幕上显示像素矩阵的代码则仅与操作系统相关。

【Java 代码】

```
class Matrix{   //各种格式的文件最终都被转化为像素矩阵
    //此处代码省略
 };
abstract class ImageImp{
   public  abstract void doPaint( Matrix m );    //显示像素矩阵m
 };
class WinImp extends ImageImp{
   public void doPaint( Matrix m ){ /*调用windows系统的绘制函数绘制像素矩阵*/ }
 };
class LinuxImp extends ImageImp{
   public void doPaint( Matrix m ){/*调用Linux系统的绘制函数绘制像素矩阵*/}
};
abstract class Image {
   public void setImp( ImageImp imp ){
   （1）    = imp; }
   public abstract void parseFile( String fileName );
   protected   （2）  imp;
};
class BMP extends Image{
   public  void parseFile( String fileName ){
    //此处解析BMP文件并获得一个像素矩阵对象m
      （3）  ;// 显示像素矩阵m
   }
};
class GIF extends Image{
    //此处代码省略
 };
class JPEG extends Image{
```

```
    //此处代码省略
};
public class javaMain{
public static void main( String[] args ){
    //在windows操作系统上查看demo.bmp图像文件
    Image image1 = （4） ;
    ImageImp imageImp1 = （5） ;
    （6） ;
    image1.parseFile( "demo.bmp" );
  }
}
```

现假设该系统需要支持 10 种格式的图像文件和 5 种操作系统，不考虑类 Matrix 和类 javaMain，若采用桥接设计模式则至少需要设计 （7） 个类。

版权方授权希赛网发布，侵权必究

第 6 章：程序设计与设计模式 作者：希赛教育软考学院 来源：希赛网 2014年05月06日

试题4

已知某类库开发商提供了一套类库，类库中定义了Application类和Document类，它们之间的关系如图6-37所示，其中，Application类表示应用程序自身，而Document类则表示应用程序打开的文档。Application类负责打开一个已有的以外部形式存储的文档，如一个文件，一旦从该文件中读出信息后，它就由一个Document对象表示。

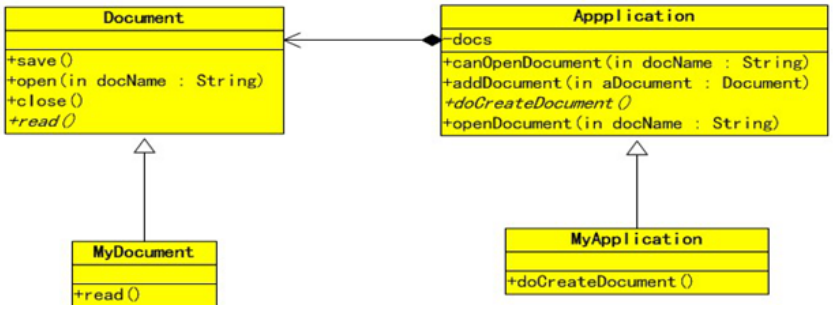

图 6-37 Application与Document关系图

当开发一个具体的应用程序时，开发者需要分别创建自己的 Application 和 Document子类，例如图6-37中的类 MyApplication 和类 MyDocument，并分别实现 Application 和Document类中的某些方法。

已知Application类中的openDocument方法采用了模板方法（Template Method）设计模式，该方法定义了打开文档的每一个主要步骤，如下所示：

（1）首先检查文档是否能够被打开，若不能打开，则给出出错信息并返回；

（2）创建文档对象；

（3）通过文档对象打开文档；

（4）通过文档对象读取文档信息；

（5）将文档对象加入到Application的文档对象集合中。

【C++代码】

```
#include <iostream>
#include <vector>
using namespace std;
class Document{
  public:
    void save( ){ /*存储文档数据，此处代码省略*/ }
    void open( string docName ){    /* 打开文档，此处代码省略  */ }
    void close( ){  /* 关闭文档，此处代码省略*/    }
    virtual void read( string docName ) = 0;
};
class Appplication{
  private:
    vector < （1） >    docs;     /*文档对象集合*/
  public:
    bool canOpenDocument( string docName ){
     /*判断是否可以打开指定文档，返回真值时表示可以打开，返回假值表示不可打开，此处代码省略*/
     }
    void addDocument( Document * aDocument ){
     /*将文档对象添加到文档对象集合中*/
       docs.push_back( （2） ) ;
     }
    virtual Document * doCreateDocument( ) = 0; /*创建一个文档对象*/
    void openDocument( string docName ){ /*打开文档*/
       if ( （3） ) {
         cout << "文档无法打开 ！" << endl;
         return;
        }
       （4） adoc = （5） ;
       （6） ;
       （7） ;
       （8） ;
    }
};
```

版权方授权希赛网发布，侵权必究

　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

试题5

已知某企业欲开发一家用电器遥控系统，即用户使用一个遥控器即可控制某些家用电器的开与关。遥控器如图 6-38 所示。该遥控器共有 4 个按钮，编号分别是 0 至 3，按钮 0和 2 能够遥控打开电器 1 和电器 2，按钮 1 和 3 则能遥控关闭电器 1 和电器 2。由于遥控系统需要支持形式多样的电器，因此，该系统的设计要求具有较高的扩展性。 现假设需要控制客厅电视和卧室电灯，对该遥控系统进行设计所得类图如6-39所示。

图 6-38遥控器

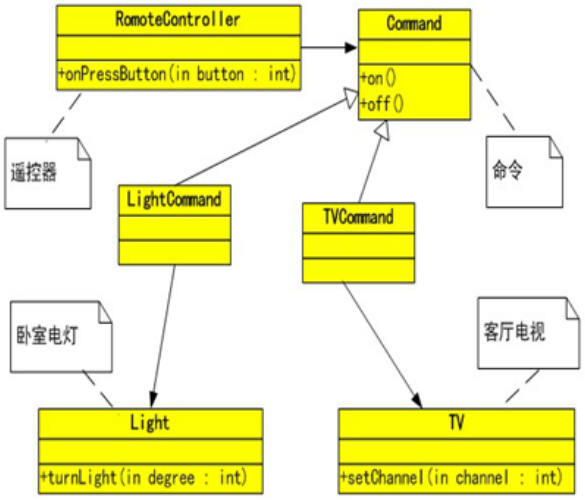

图6-39 设计类图

图6-39中， 类RomoteController的方法onPressButton(int button)表示当遥控器按键按下时调用的方法，参数为按键的编号；Command 接口中 on 和 off 方法分别用于控制电器的开与关；Light中turnLight(int degree)方法用于调整电灯灯光的强弱，参数degree值为0时表示关灯，值为100时表示开灯并且将灯光亮度调整到最大；TV 中setChannel(int channel)方法表示设置电视播放的频道，参数 channel 值为 0 时表示关闭电视，为 1 时表示开机并将频道切换为第1频道。

【Java代码】

```
class Light{    //电灯类
public void turnLight( int degree ){ //调整灯光亮度，0表示关灯，100表示亮度最大}
};
class TV{ //电视机类
      public void setChannel( int channel ){// 0表示关机，1表示开机并切换到1频道  }
};
```

```
interface Command{ //抽象命令类
    void on( );
     void off( );
};
class RemoteController{    //遥控器类
      protected Command []commands = new Command[4];
      //遥控器有4个按钮，按照编号分别对应4个Command对象
      public void onPressButton( int button ){
        //按钮被按下时执行命令对象中的命令
        if( button % 2 = = 0 )commands[button].on( );
        else commands[button].off( );
      }
      public void setCommand( int button, Command command ){
        （1） = command; //设置每个按钮对应的命令对象
      }
};
class LightCommand implements Command{    //电灯命令类
      protected Light light;    //指向要控制的电灯对象
      public void on( ){light.turnLight( 100 );};
      public void off( ){light. （2） ;};
      public LightCommand( Light light ){this.light = light;};
};
class TVCommand implements Command{ //电视机命令类
      protected TV tv;      //指向要控制的电视机对象
      public void on( ){tv.（3） ;};
      public void off( ){tv.setChannel( 0 );};
      public TVCommand( TV tv ){this.tv = tv;};
};
public class rs{
      public static void main( String []args ){
        Light light = new Light( );  TV tv = new TV( );//创建电灯和电视对象
        LightCommand lightCommand = new LightCommand( light );
        TVCommand tvCommand = new TVCommand( tv );
        RemoteController remoteController = new RemoteController( );
        //设置按钮和命令对象
        remoteController.setCommand( 0,（4） );
        …//此处省略设置按钮1、按钮2和按钮3的命令对象代码
      }
}
```

本题中，应用命令模式能够有效让类（5）和类（6）、类（7）之间的耦合性降至最小。

版权方授权希赛网发布，侵权必究

第 6 章：程序设计与设计模式　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

试题6

某游戏公司现欲开发一款面向儿童的模拟游戏，该游戏主要模拟现实世界中各种鸭子的发声特征、飞行特征和外观特征。游戏需要模拟的鸭子种类及其特征如表 6-7 所示：

表6-7 模拟的鸭子种类及其特征

鸭子种类	发声特征	飞行特征	外观特征
灰鸭（MallardDuck）	发出"嘎嘎"声（Quack）	用翅膀飞行（FlyWithWings）	灰色羽毛
红头鸭（RedHeadDuck）	发出"嘎嘎"声（Quack）	用翅膀飞行（FlyWithWings）	灰色羽毛,头部红色
棉花鸭（CottonDuck）	不发声（QuackNoWay）	不能飞行（FlyNoWay）	白色
橡皮鸭（RubberDuck）	发出橡皮与空气摩擦的声（Squeak）	不能飞行（FlyNoWay）	黑白橡皮颜色

为支持将来能够模拟更多种类鸭子的特征，采用策略设计模式（Strategy）设计的类图如图 6-40 所示：

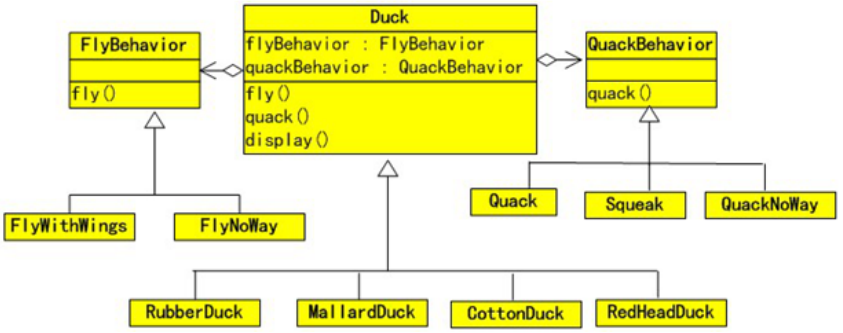

图 6-1 设计的类图

在类图中，Duck 为抽象类，描述了抽象的鸭子，而类 RubberDuck、MallardDuck、CottonDuck和 RedHeadDuck 分别描述具体的鸭子种类，方法 fly( )、quack( )和 display( )分别表示不同种类的鸭子都具有的飞行特征、发声特征和外观特征；类 FlyBehavior 与 QuackBehavior 为抽象类，分别用于表示抽象的飞行行为与发声行为；类 FlyNoWay 与 FlyWithWings 分别描述不能飞行的行为和用翅膀飞行的行为；类 Quack、Squeak 与 QuackNoWay 分别描述发出“嘎嘎”声的行为、发出橡皮与空气摩擦声的行为与不发声的行为。请填补以下代码中的空缺。

【C++代码】

```
#include< iostream>
using namespace （1）;
class FlyBehavior {
    public: （2） fly( )= 0;
};
```

```
class QuackBehavior {
    public:（3）quack( ) = 0;
};
class FlyWithWings:publicFlyBehavior{
    public:void fly( ){ cout〈〈“使用翅膀飞行！”〉〉endl; }
};
class FlyNoWay:publicFlyBehavior{
    public: void fly( ){ cout “不能飞行！” endl; }
};
class Quack:publicQuackBehavior{
    public:void quack( ){ “cout发出\'嘎嘎\'声！” endl; }
};
class Squeak:publicQuackBehavior{
    public:voidquack( ){ cout 〈〈“发出空气与橡皮摩擦声！”〉〉endl;}
};
class QuackNoWay:publicQuackBehavior{
    public:voidquack( ){ cout 〈〈“不能发声!”〉〉endl; }
};
class Duck{
protected:
     FlyBehavior *（4）;
     QuackBehavior *（5）;
public:
     void fly( ){（6）; }
     void quack( ) {（7）; };
     virtual void display( )=0;
};
class RubberDuck：public Duck{
public:
    RubberDuck( ){
        flyBehavior = new（8）;
        quackBehavior = new（9）;
    }
    RubberDuck( ){
        if( !flyBehavior ) delete flyBehavior;
        if( !quackBehavior ) deletequackBehavior;
    }
    void display( ){ /*此处省略显示橡皮鸭的代码 */}
};
```

```
//其它代码省略
```

版权方授权希赛网发布，侵权必究

　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

练习题解析

试题1分析

本题考查基本面向对象设计模式的运用能力。

策略模式提供了替代派生的子类，并定义类的每个行为，把容易发生变化的算法独立出来，易于扩展，系统变得更加灵活。结合文字描述和UML结构图所给的语义关系，AirCraft为抽象类，类FlyBehavior与TakeOffBehavior与AirCraft之间有组合关系，AirCraft有两个特性（成员变量）flyBehavior和takeOffBehavior，两个操作（成员方法）fly( )和takeOff( )表达部分与整体的关系。同时注意关联关系、聚合关系和组合关系都是通过类的成员变量来实现的。在抽象类AirCraft中的成员变量应反应出了这一点，所以空（1）应填写flyBehavior = new FlyBehavior( )；空（2）应填takeOffBehavior = new TakeOffBehavior( )；空（3）应填flyBehavior.fly( )；空（4）应填takeOffBehavior.takeOff( )。由题知类Helicopte是AirCraft的子类，所以空（5）应填extends；空（6）应填AirCraft( )；空（7）应填AirCraft( )。

试题1参考答案

（1）flyBehavior= new FlyBehavior( )

（2）takeOffBehavior = new TakeOffBehavior( )

（3）flyBehavior.fly( );

（4）takeOffBehavior.takeOff( )

（5）extends

（6）AirCraft( )

（7）AirCraft( )

试题2分析

本题考查基本面向对象设计模式的运用能力。组合设计模式主要是表达整体和部分的关系，并且对整体和部分对象的使用无差别。由UML结构图知AbstractFile是File类和Folder类的父类，它抽象了两个类的共有属性和行为。在使用中，不论是File对象还是Folder对象，都可被当作AbstractFile对象来使用。另外由UML结构图知，类Folder和AbstractFile之间存在共享关系，Folder对象可以共享其它的Folder对象和File对象。在类File和类Folder的构造函数中都需要记录文件或目录的名称，因此空（1）和空（4）处主要是对象的名称，应填this->name。由于File对象没有孩子节点，所以空（3）应填NULL。getChildren( )方法继承自AbstractFile类，由于其返回类型应保持一致性，故空（2）应填list<AbstractFile*>。对于空（5），返回Folder对象的孩子节点，因此返回其成员childList的地址，应填& childList。

试题2参考答案

（1）this->name

（2）list<AbstractFile*>

（3）NULL

（4）his->name

（5）& childList

试题3分析

由文字描述和UML结构图可知BMP、GIF与JPEG是Image的子类，分别负责读取不同格式的文件。ImageImp的主要任务是将像素矩阵显示在屏幕上，它的两个子类WinImp 、 LinuxImp分别实现windows系统和Linux系统上的图像显示代码。空（1）处主要设置在哪个平台上进行实现，由于该类的成员变量也是imp，与参数相同，因此应填this.imp。同理，该成员变量的类型和参数的类型也应保持相同，故空（2）处应填ImageImp。空（3）处需要根据imp成员变量存储的实现对象来显示图像，应填imp.doPaint(m)；在空（4）处需要生成一个BMP对象，故应填new BMP()，在空（5）处需要生成一个WinImp对象，故应填new WinImp()，空（6）处应填image1.setImp(ImageImp1)，采用Bridge（桥接模式）能够将文件分析代码和图像显示代码分解在不同的类层结构中，如果不考虑Matrix等类，那么最后需要设计的类包括2个父类，分别为文件格式子类和操作系统平台类，故系统需要支持10种格式的图像文件和5种操作系统至少需要17个类。

试题3参考答案

（1）this.imp

（2）ImageImp

（3）imp.doPaint(m)

（4）new BMP()

（5）new WinImp()

（6）image1.setImp(ImageImp1)

（7）17

试题4分析

本题考查了C++语言的应用能力和模板方法设计模式。空（1）考查了C++标准库中Vector模板类的使用，由于Vector模板类可以存储任意类型，在定义时需要指定其存储类型，根据后面的代码，能够加入到该文档集合对象中的元素是各个文档的指针，因此空（1）处的类型应该为文档指针类型。空（2）处将文档指针加入文档集合对象中。从空（3）开始的代码属于图中Application类的OpenDocument方法，该方法是模板方法，因此，需根据题目给出的步骤一一对应填空。空（3）处判断能否打开文档，需要调用父类自己的方法canOpenDocument。其次需要创建文档对象，调用doCreateDocument方法，接着通过文档对象打开和读取文档，最后通过addDocument方法将该文档对象加入到文档对象集合中。所有这些方法都是在父类或文档对象中进行定义，不涉及到具体的子类，子类负责要实现这些模板方法中需要调用的方法以便运行时被调用。

试题4参考答案

（1）Document*

（2）aDocument

（3）!canOpenDocument(docName)

（4）Document *

（5）doCreateDocument()

（6）adoc->open(docName)

（7）adoc->read(docName)

（8）addDocument(adoc)

试题5分析

本题考查的是设计模式中的命令模式。在设计时，为了保证遥控器和家用电器之间的独立性，定义了Command类，当用户按下遥控器上的按钮时，触发Command上的On或者Off方法，因此，一对按钮分别对应一个Command对象。题目中的LightCommand以及与TVCommand分别为Command的子类，该子类用于控制实际的Light以及TV对象，将On与Off方法委托给Light以及TV实现。空（1）表示要设置遥控器上按钮控制的对象，其参数传递的是某一个命令对象，因此只需将该命令对象存储下来即可；空（2）表示关闭电灯，根据说明，关闭电灯的方法为turnLight（0）；空（3）表示打开电视机，因此需要调用打开电视的方法。空（4）表示将按钮0和相应的Command对象相关联，根据题目描述，按钮0用于控制灯或者电视，因此，应该设置灯或者电视的命令对象。本题中应用命令模式的目的是为了使为了让遥控器和类Light与TV之间的耦合性降至最低。

试题5参考答案

（1）commands[button]

（2）trunLight(0)

（3）setChannel(1)

（4）lightCommand

（5）RemoteController

（6）Light

（7）TV

试题6分析

C++标准的输出输入的命名空间为std,在本题的代码中使用了cout，为此必须使用标准的命名空间，空（1）处应该填std。类 FlyNoWay 与 FlyWithWings继承了FlyBehavior，根据它们的成员函数fly的定义可知，fly函数的返回值为void，加上FlyBehavior中的函数为纯虚函数，故空（2）处应填virtual void.。空（3）处的原理与空（2）处相同，应填virtual void。Duck是各种鸭子的基类，类FlyBehavior 与 QuackBehavior分别实现鸭子的飞行特征、发声特征。因此空（4）和空（5）处应该为这两个类的对象或指针（这两个类为纯虚函数，因此只能采用指针形式），应分别填flyBehavior和quackBehavior。空（6）和空（7）应是调用相应的方法实现其飞行特征、发声特征，故应填flyBehavior->fly( )和quackBehavior->quack( )。同时在每一种具体的鸭类的构造函数中需要指定其具有的飞行特征、发声特征，要实现该特征，可通过构造 FlyNoWay和Squeak的对象来完成，故空（8），空（9）应填FlyNoWay( )和Squeak( )。

试题6参考答案

（1）std

（2）virtual void

（3）virtual void

（4）flyBehavior

（5）quackBehavior

（6）flyBehavior->fly( )

（7）quackBehavior->quack()

（8）FlyNoWay()

（9）Squeak()

版权方授权希赛网发布，侵权必究

第 6 章：程序设计与设计模式　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

试题解答方法

软件设计师下午的程序设计与设计模式考点为二选一试题，两个试题考查的知识点是一样的，只是实现的语言不同而已，主要就是考的设计模式+程序语言基本语法，以程序填空的答题形式出现。这种类型的题需要对所考查的设计模式有一定的了解。这类题目以程序填空形式出现，这不仅要求理解问题本质，同时也要弄清解题思路。人们常讲，答案就在题目中，这是对的。在分析问题的过程中，找到所求答案。不过前提条件是考生要熟悉这种语言，又要明白解题思路，这样才能正确作答。

为此，考生要认真阅读文字，结合所给的UML结构图，理解类与类之间的关系。UML类图中的关系分为四种：UML类图依赖关系、泛化关系、关联关系、实现关系。关联关系又可以细化为聚合和组合。同时注意关联关系、聚合关系和组合关系都是通过类的成员变量来实现的。在此基础上进一步明确哪些是接口，哪些是抽象类。然后阅读所给的C++或JAVA程序，先从宏观上把握，再从微观入手，即遵从类（接口）--------类（接口）间---------类（接口）内这样一个阅读程序思路，把握所考设计模式的思想和解决所描述问题的意图，结合文字描述和UML结构图所给的语义关系，准确写出所给的填空项，完成答题。读者可依据试题解答方法思路回头对考前必练进行自我分析，多体会，加深理解和消化，提高答题的正确率。

版权方授权希赛网发布，侵权必究