C++设计模式及应用

摘要：C++的设计模式是一个被很多人讨论过的话题，其实我们不用把模式看的很神秘，所谓的模式就是我们思考问题的一种方法，一些达人为了后人的方便，而将一些相类似问题的解决方法抽象出来了，从而形成一种模式。其实，每个程序员在编程过程中都会自觉或不自觉的用到某些模式，所以大家不要刻意的去仿照某种模式，在你对编程有了一定的领悟后，模式就会不请自来的出现在你的脑海里。

C++的常用模式有：简单工厂模式（Simple Factory Pattern）、工厂方法模式（Factory Method Pattern）、抽象工厂模式（Abstract Factory）、单件模式（Single Pattern）等。

关键词：程序设计、C++、设计模式、UML。

C + + design patterns and application

Abstract: C + + is a design model by many people have discussed topics, actually we don't see the pattern of the mysterious, so-called pattern is a way of thinking for posterity, some da, and some of the convenience of similar problem solving method was abstracted, thus forming a kind of mode. Actually, each programmer in the programming process will consciously or not to use certain mode, so you don't painstakingly took a pattern of programming, you have understanding, pattern will not invited appeared in your mind.

C++ is the common mode： Simple Factory Pattern、Factory Method Pattern、Abstract Factory）、Single Pattern and so on.

Keywords：Programming、C++、common mode、UML、realizing.

目 录

第一章 前 言

在 1966 年，Dahl 和 Nygaard 发表了关于在 Simula 中提出类（Class）和子类（SubClass）的概念，这些概念后来被应用到 Simula 67 中。1967 年 6 月的一个会议上，Dahl 建议对 Type 和 Class 概念进行明确定义，这引发了一次认真严谨的讨论，为语言的标准化奠定了基础；

Simlua 是第一个实现的面向对象语言，但并不是第一个完整实现的。Smalltalk 虽然并没有被广泛使用，但它对面向对象技术的推广起了非常大的作用。

C++诞生于 1979 年 Bells 实验室，是第一个被广泛使用的面向对象语言。C++是一个从 C 语言演化过来的支持面向对象编程的程序语言。随着 C++等面向对象程序语言的发展，开发人员在开发过程中不断总结了各种经验，最终渐渐形成了一套理论性的开发框架，这就是设计模式。由于设计模式是一个经验累积型的概念，因此没有能够在实际教学过程中得到好的应用；相反需要的开发人员长时间的开发经验才能够接触或领会。

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。 设计模式的产生主要是由开发者在开发过程中不断总结、归纳形成的一个可重用的代码框架或者是设计思想。想对而言，这些关于单个设计模式的论述显得比较零散，又因为各种程序设计语言的差异，导致部分设计模式在 C++中的应用显得比较牵强或者不够灵活[1]。

再者，目前大部分学员对 C++或其他面向对象语言的学习还停留在语言特性的基础上，几乎没有从面向对象的角度去思考、没有使用面向对象的方法来处理面向对象程序设计中遇到的问题，而改变这一传统学习方式的方法就是在学习面向对象语言的时候引入面向对象设计模式。

第二章 程序设计语言发展简史

2.1. 结构化程序设计：

结构化程序设计由迪克斯特拉(E.W.dijkstra)在 1969 年提出，是以模块化设

计为中心，将待开发的软件系统划分为若干个相互独立的模块，这样使完成每一个模块的工作变单纯而明确，为设计一些较大的软件打下了良好的基础。

由于模块相互独立，因此在设计其中一个模块时，不会受到其它模块的牵连，因而可将原来较为复杂的问题化简为一系列简单模块的设计。模块的独立性还为扩充已有的系统、建立新系统带来了不少的方便，因为我们可以充分利用现有的模块作积木式的扩展。

按照结构化程序设计的观点，任何算法功能都可以通过由程序模块组成的三种基本程序结构的组合: 顺序结构、选择结构和循环结构来实现。

结构化程序设计的基本思想是采用"自顶向下，逐步求精"的程序设计方法和"单入口单出口"的控制结构。自顶向下、逐步求精的程序设计方法从问题本身开始，经过逐步细化，将解决问题的步骤分解为由基本程序结构模块组成的结构化程序框图；"单入口单出口"的思想认为一个复杂的程序，如果它仅是由顺序、选择和循环三种基本程序结构通过组合、嵌套构成，那么这个新构造的程序一定是一个单入口单出口的程序。据此就很容易编写出结构良好、易于调试的程序来。

2.2. 面向对象程序设计：

面向对象程序设计中的概念主要包括：对象、类、数据抽象、继承、动态绑定、数据封装、多态性、消息传递。通过这些概念面向对象的思想得到了具体的体现。

面向对象出现以前，结构化程序设计是程序设计的主流，结构化程序设计又称为面向过程的程序设计。在面向过程程序设计中，问题被看作一系列需要完成的任务，函数（在此泛指例程、函数、过程）用于完成这些任务，解决问题的焦点集中于函数。其中函数是面向过程的，即它关注如何根据规定的条件完成指定的任务。

一个语言要称为面向对象语言必须支持几个主要面向对象的概念。根据支持程度的不同，通常所说的面向对象语言可以分成两类：基于对象的语言，面向对象的语言。

基于对象的语言仅支持类和对象，而面向对象的语言支持的概念包括：类与对象、继承、多态。举例来说，Ada 就是一个典型的基于对象的语言，因为它不

支持继承、多态，此外其他基于对象的语言还有 Alphard、CLU、Euclid、Modula。面向对象的语言中一部分是新发明的语言，如 Smalltalk、Java，这些语言本身往往吸取了其他语言的精华，而又尽量剔除他们的不足，因此面向对象的特征特别明显，充满了蓬勃的生机；另外一些则是对现有的语言进行改造，增加面向对象的特征演化而来的。如由 Pascal 发展而来的 Object Pascal，由 C 发展而来的 Objective-C，C++，由 Ada 发展而来的 Ada 95 等，这些语言保留着对原有语言的兼容，并不是纯粹的面向对象语言，但由于其前身往往是有一定影响的语言，因此这些语言依然宝刀不老，在程序设计语言中占有十分重要的地位[2]。

面向对象程序设计的优点：面向对象出现以前，结构化程序设计是程序设计的主流，结构化程序设计又称为面向过程的程序设计。在面向过程程序设计中，问题被看作一系列需要完成的任务，函数（在此泛指例程、函数、过程）用于完成这些任务，解决问题的焦点集中于函数。其中函数是面向过程的，即它关注如何根据规定的条件完成指定的任务。

在多函数程序中，许多重要的数据被放置在全局数据区，这样它们可以被所有的函数访问。每个函数都可以具有它们自己的局部数据。下图显示了一个面向过程程序中函数和数据的关系

面向对象的主要特征有：封装性、继承性、多态性

在面对对象方法中，对象和传递消息分别表现事物及事物间相互联系的概念。类和继承是是适应人们一般思维方式的描述范式。方法是允许作用于该类对象上的各种操作。这种对象、类、消息和方法的程序设计范式的基本点在于对象的封装性和类的继承性。通过封装能将对象的定义和对象的实现分开，通过继承能体现类与类之间的关系，以及由此带来的动态联编和实体的多态性，从而构成了面向对象的基本特征。

面向对象设计是一种把面向对象的思想应用于软件开发过程中，指导开发活动的系统方法，是建立在“对象”概念基础上的方法学。对象是由数据和容许的操作组成的封装体，与客观实体有直接对应关系，一个对象类定义了具有相似性质的一组对象。而每继承性是对具有层次关系的类的属性和操作进行共享的一种方式。所谓面向对象就是基于对象概念，以对象为中心，以类和继承为构造机制，来认识、理解、刻画客观世界和设计、构建相应的软件系统。。按照 Bjarne STroustRUP 的说法，面向对象的编程范式：

1 决定你要的类；

2 给每个类提供完整的一组操作；

3 明确地使用继承来表现共同点。

由这个定义，我们可以看出：面向对象设计就是“根据需求决定所需的类、类的操作以及类之间关联的过程”。

面向对象设计方法的特点和面临的问题

面向对象设计方法以对象为基础，利用特定的软件工具直接完成从对象客体的描述到软件结构之间的转换。这是面向对象设计方法最主要的特点和成就。面向对象设计方法的应用解决了传统结构化开发方法中客观世界描述工具与软件结构的不一致性问题，缩短了开发周期，解决了从分析和设计到软件模块结构之间多次转换映射的繁杂过程，是一种很有发展前途的系统开发方法。

但是同原型方法一样, 面向对象设计方法需要一定的软件基础支持才可以应用，另外在大型的 MIS 开发中如果不经自顶向下的整体划分，而是一开始就自底向上的采用面向对象设计方法开发系统，同样也会造成系统结构不合理、各部分关系失调等问题。所以面向对象设计方法和结构化方法目前仍是两种在系统开发领域相互依存的、不可替代的方法。

第三章　C++语言特性

C++作为一门高效率的面向对象程序设计语言，具有面向对象的所有特性：封装、继承、多态，并且完美的实现了这些特性，这对任何开发者来说都是幸运的，这因为这些，我们的开发变得容易而充满乐趣。我们可以将需要解决的问题方便的转化为计算机的软件模型，这一过程当然需要面向对象思想的支持。

3.1.类

类（Class）实际上是对某种类型的对象定义变量和方法的原型。它表示对现实生活中一类具有共同特征的事物的抽象，是面向对象编程的基础。比如我常说的人、小狗、公交车、学校等具有泛指的名词，通常都是指代“类”。类常常是一类对象的统称。

类是对某个对象的定义。它包含有关对象动作方式的信息，包括它的名称、方法、属性和事件。需要注意的是类并不是实际意义的对象，类是一类事物具有相同或相似属性动作的事物的总称。

可以把类看作“理论上”的对象，也就是说，它为对象提供蓝图。从这个蓝图可以创建任何数量的实际对象。从类创建的所有对象都有相同的成员：属性、方法和事件。但是，每个对象都象一个独立的实体一样动作。例如，一个对象的属性可以设置成与同类型的其他对象不同的值，比如同样是人类，各自的姓名不同、身高不同、爱好不同，就是这些相同属性的不同值的对象形成了对象之间的独立性，但是他们都属于同一个类。

C++中，类定义的一般形式如下：

```
class Name {
public:
    类的公有接口 MFC 类图
private:
    私有的成员函数
    私有的数据成员定义
};
```

类的定义由 类头 和 类体 两部分组成。类头由关键字 Class 开头，然后是类名，其命名规则与一般标识符的命名规则一致。类体包括所有的细节，并放在一对花括号中。类的定义也是一个语句，所以要有分号结尾，否则会产生编译错误。

类体定义类的成员，它支持两种类型的成员：

（1）数据成员，它们指定了该类对象的内部表示。

（2）成员函数，他们指定了该类的操作。

类成员有 3 种不同的访问权限：

（1）公有（ public ）成员可以在类外访问。

（2）私有（ private ）成员只能被该类的成员函数访问。

（3）保护（ protected ）成员只能被该类的成员函数或派生类的成员函数访问。

数据成员通常是私有的，成员函数通常有一部分是公有的，一部分是私有的。

公有的函数可以在类外被访问，也称之为类的接口。可以为各个数据成员和成员函数指定合适的访问权限。

下面是一个人类的定义

```
class CPerson //开始人类的定义
{
    public:
    char *  m_name; //姓名属性
    char *  m_telephone;//电话号码
    int m_hight;//身高
    public:
    void Sleep();//睡觉
    void Speak();//说话
    void Eat();//进食
}
```

上述属性和动作的访问控制都是公开的(public)，这是因为诸如人的属性：姓名、电话、身高和动作：吃饭、睡觉、进食都是公开的，但是有的属性和动作应该是私有的比如一个人的内心世界这就应该设置为 protected (保护)或者 private(私有)。

3.2 继承

继承是指一个对象直接使用另一对象的属性和方法。事实上，我们遇到的很多实体都有继承的含义。例如，若把汽车看成一个实体，它可以分成多个子实体，如：卡车、公共汽车等。这些子实体都具有汽车的特性，因此，汽车是它们的"父亲"，而这些子实体则是汽车的"孩子"。

继承常常用于扩展一类的属性和功能，派生出特殊的类。下面是一个学生类，作为学生出了具有人类的基本属性和动作之外，还具有学校单位属性、年级等属性以及上学的行为；所以学生类定义如下：

```
class CStudent : public CPerson //从人类继承而来 扩展人类的属性和动作
{
    public:
    char *  m_SchoolName;//学校名称
    int m_grage;  //年级
```

```
    public:
    void GotoSchool();//上学的行为
}
```

3.3.多态

C++是以C语言为基础，支持数据抽象和面向对象的程序设计语言。C++对C语言的扩充部分汲取了许多著名语言中最优秀的特征，但人们对它许多潜在的性能(封装、继承、多态等)还没有充分地理解和应用，没有充分发挥其优势。多态性是面向对象系统的重要概念之一，它指的是同样的消息能被发送到父类的对象和它的子类的对象。多态对设计模式的实现具有特别重要的意思

多态性的实现形式

从广义上说，多态性是指一段程序能够处理多种类型对象的能力。在C++语言中，这种多态性可以通过强制多态、重载多态、类型参数化多态、包含多态4种形式来实现。类型参数化多态和包含多态统称为一般多态性，用来系统地刻画语义上相关的一组类型。重载多态和强制多态统称为特殊多态性，用来刻画语义上无关联的类型间的关系。

子集构成1个子类型。每一个子类型中的对象可以被用在高一级的类型中，高一级类型中的所有操作可用于下一级的对象。在C++中公有继承关系是一种包含多态，每一个类可以直接公有继承父类或多个父类，如语句class D　public P1，public P2{……}；表示类D分别是类P1和类P2的子类型。C++中采用虚拟函数实现包含多态，虚拟函数为C++提供了更为灵活的多态机制，这种多态性在程序运行时才能确定，因此虚拟函数是多态性的精华，至少含有一个虚拟函数的类称为多态类。包含多态在程序设计中使用十分频繁。派生类继承基类的所有操作，或者说，基类的操作能被用于操作派生类的对象，当基类的操作不能适应派生类时，派生类需重载基类的操作，见下例中的void circle::showarea()。

```
#include <iostream.h>
class point  //屏幕上的点类
{int x, y;
  public;
```

```
        point(int x1, int y1)
       {x=x1; y=y1;}
        void showarea()
       {cout<<〃Area of point is: 〃<<0.0<<endl;}
      };

      class circle: public point//圆类
       {int radius;
        public:
        circle(int x, int y, int r): point(x, y){ radius=r;
}
       void showarea() {cout<<〃Area of circle is: 〃<
<3.14
        *radius*radius<<endl; }
      };

      void disparea(const point*p) //多态程序段
       {p->showarea();}
      void main()
       {circle c1(1, 1, 1); disparea(&c1);
      }
```

程序的运行结果为 0.0(正确结果应为 3.14)，出错的原因是：表达式 p->showarea()中的函数调用在编译时被束定到函数体上，使得这个表达式中的函数调用执行 point 类的 showarea()。为此，当程序员在实现一个派生类而变动了基类中的操作实现时，C++提供的虚函数机制可将这种变动告诉编译器，即将关键字 virtual 放在类 point 中该函数的函数说明之前(virtual void showarea())，程序其它部分保持不变(circle::showarea()自动地成为虚函数)，编译器就不会对函数调用 p->showarea()进行静态束定(在编译/连接时进行的束定)而产生有关的代码，使函数调用与它所应执行的代码的束定工作在程序运行时进行，这样上述程序的运行结果即为 3.14。在程序运行时进行的

束定被称为动态束定。利用虚函数，可在基类和派生类中使用相同的函数名定义函数的不同实现，从而实现“一个接口，多种方式”。当用基类指针或引用对虚函数进行访问时，软件系统将根据运行时指针或引用所指向或引用的实际对象来确定调用对象所在类的虚函数版本。

C++语言还增加了纯的虚函数机制用来更好地设计包含多态性。假如每个类中都有一个函数“void display(void);”，那么，怎样对它们按多态性进行统一处理呢？对这类问题应先设计一个抽象的类，使它成为所有类的祖先类。设置类 A 的目的是由它说明统一使用该层次中的 display()函数的方法(赋值兼容规则从语法上保证了 A 的子孙类可按 A 说明的方式使用 display()函数；多态性则从语义上保证了在执行时，根据实际的对象访问相应对象类中的 display()函数)。为了保证在类 A 中设置的 display()函数是抽象动作，并能说明类 A 是一个抽象的类，在 C++中，可用纯的虚函数语言机制在类 A 中声明 1 个成员函数“virtual void display(void)＝0;”。请注意，在类 A 的子孙类中要么给出 display()的定义，要么重新将该函数声明为纯的。从上面的分析可以看出，类 A 的设计尽管是用继承性语法表达的，但它的主要目的不是为代码共享而设计的，而是为了提高多态性而设计的，它是另一个维度的抽象。

第四章 统一建模语言 UML

4.1 UML 简介

UML 是一种定义良好、易于表达、功能强大且普遍适用的建模语言。它溶入了软件工程领域的新思想、新方法和新技术。它的作用域不限于支持面向对象的分析与设计，还支持从需求分析开始的软件开发的全过程。

面向对象技术和 UML 的发展过程可用图形来表示，标准建模语言的出现是其重要成果。在美国，截止 1996 年 10 月，UML 获得了工业界、科技界和应用界的广泛支持，已有 700 多个公司表示支持采用 UML 作为建模语言。1996 年底，UML 已稳占面向对象技术市场的 85％，成为可视化建模语言事实上的工业标准。1997 年 11 月 17 日，OMG 采纳 UML 1.1 作为基于面向对象技术的标准建模语言。UML 代表了面向对象方法的软件开发技术的发展方向，具有巨大的市场前景，

也具有重大的经济价值和国防价值[3]。

4.2 UML 的内容

首先，UML 融合了 Booch、OMT 和 OOSE 方法中的基本概念，而且这些基本概念与其他面向对象技术中的基本概念大多相同，因而，UML 必然成为这些方法以及其他方法的使用者乐于采用的一种简单一致的建模语言；其次，UML 不仅仅是上述方法的简单汇合，而是在这些方法的基础上广泛征求意见，集众家之长，几经修改而完成的，UML 扩展了现有方法的应用范围；第三，UML 是标准的建模语言，而不是标准的开发过程。尽管 UML 的应用必然以系统的开发过程为背景，但由于不同的组织和不同的应用领域，需要采取不同的开发过程。

作为一种建模语言，UML 的定义包括 UML 语义和 UML 表示法两个部分。

(1) UML 语义 描述基于 UML 的精确元模型定义。元模型为 UML 的所有元素在语法和语义上提供了简单、一致、通用的定义性说明，使开发者能在语义上取得一致，消除了因人而异的最佳表达方法所造成的影响。此外 UML 还支持对元模型的扩展定义

(2) UML 表示法 定义 UML 符号的表示法，为开发者或开发工具使用这些图形符号和文本语法为系统建模提供了标准。这些图形符号和文字所表达的是应用级的模型，在语义上它是 UML 元模型的实例。

标准建模语言 UML 的重要内容可以由下列五类图（共 9 种图形）来定义：

第一类是用例图，从用户角度描述系统功能，并指出各功能的操作者。如下是学生管理系统的用例图（图 4.1）：

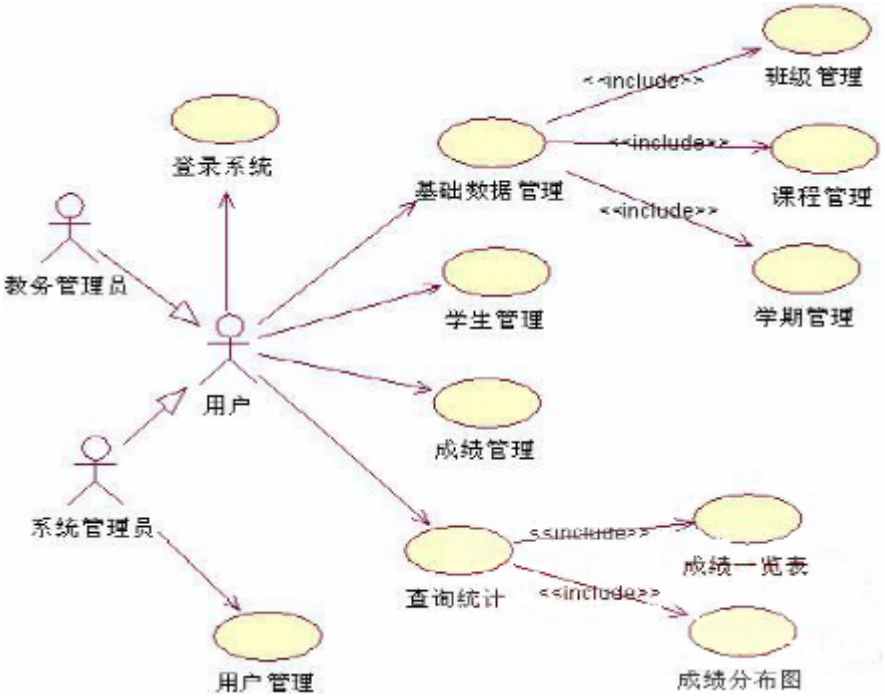


图 4.1 学生管理系统用例图

从这个用例图中，我们可以方便的看到整个系统的功能模块一个各个模块之间的交互关系。

第二类是静态图 (Static diagram)，包括类图、对象图和包图。其中类图描述系统中类的静态结构。不仅定义系统中的类，表示类之间的联系如关联、依赖、聚合等，也包括类的内部结构（类的属性和操作）。类图描述的是一种静态关系，在系统的整个生命周期都是有效的。对象图是类图的实例，几乎使用与类图完全相同的标识。他们的不同点在于对象图显示类的多个对象实例，而不是实际的类。一个对象图是类图的一个实例。由于对象存在生命周期，因此对象图只能在系统某一时间段存在。包由包或类组成，表示包与包之间的关系。包图用于描述系统的分层结构。

这是一个选课系统的简单领域分析类图（图 4.2）。

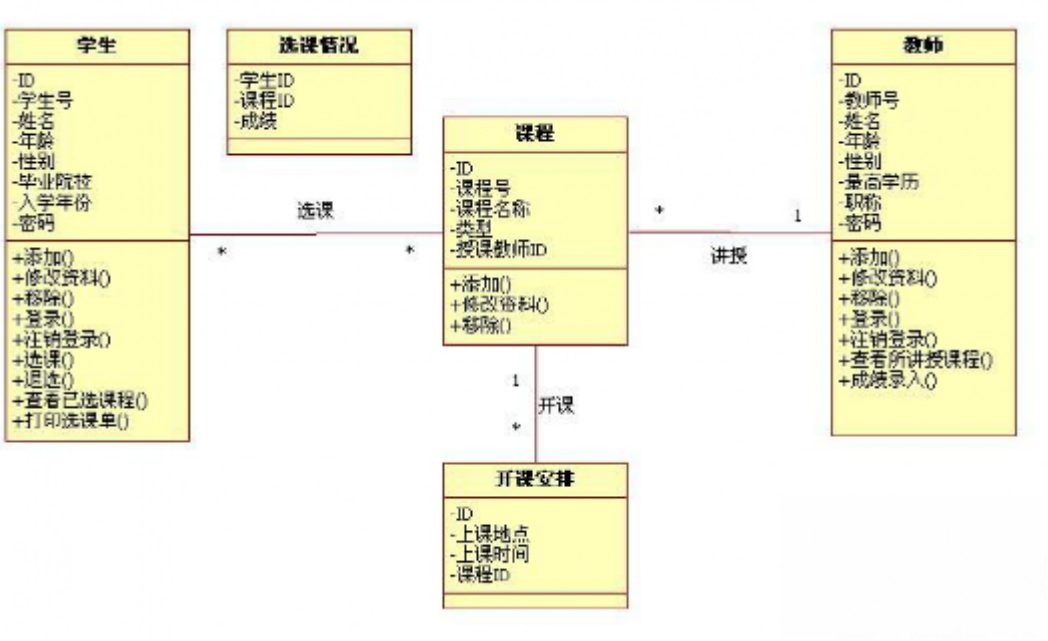


图 4.2 分析类图

可以看到，主要实体有教师、学生、课程和开课安排。每个实体标注了其在业务上具有的属性和方法。而且图中还标明了实体间的关系。

第三类是行为图（Behavior diagram），描述系统的动态模型和组成对象间的交互关系。其中状态图描述类的对象所有可能的状态以及事件发生时状态的转移条件。通常，状态图是对类图的补充。在实用上并不需要为所有的类画状态图，仅为那些有多个状态其行为受外界环境的影响并且发生改变的类画状态图。而活动图描述满足用例要求所要进行的活动以及活动间的约束关系，有利于识别并行活动。

第四类是交互图（Interactive diagram），描述对象间的交互关系。其中顺序图显示对象之间的动态合作关系，它强调对象之间消息发送的顺序，同时显示对象之间的交互；合作图描述对象间的协作关系，合作图跟顺序图相似，显示对象间的动态合作关系。除显示信息交换外，合作图还显示对象以及它们之间的关系。如果强调时间和顺序，则使用顺序图；如果强调上下级关系，则选择合作图。这两种图合称为交互图。

第五类是实现图（Implementation diagram）。其中构件图描述代码部件的物理结构及各部件之间的依赖关系。一个部件可能是一个资源代码部件、一个二进制部件或一个可执行部件。它包含逻辑类或实现类的有关信息。

第五章　C++中类与对象及其关系

在一个系统中，常常会有许多各司其职而有互相影响的类存在，这些类之间的关系通常有：关联、泛化(Generalization)、依赖(Dependency)、聚合(Aggregation)、组合（Composite）

5.1、关联(Association)

两个相对独立的对象，当一个对象的实例与另外一个对象的特定实例存在固定关系时，这两个对象之间就存在关联关系。

5.1.1 单向关联

A1->A2: 表示 A1 认识 A2，A1 知道 A2 的存在，A1 可以调用 A2 中的方法和属性

场景：订单和商品，订单中包括商品，但是商品并不了解订单的存在。

类与类之间的单向关联图

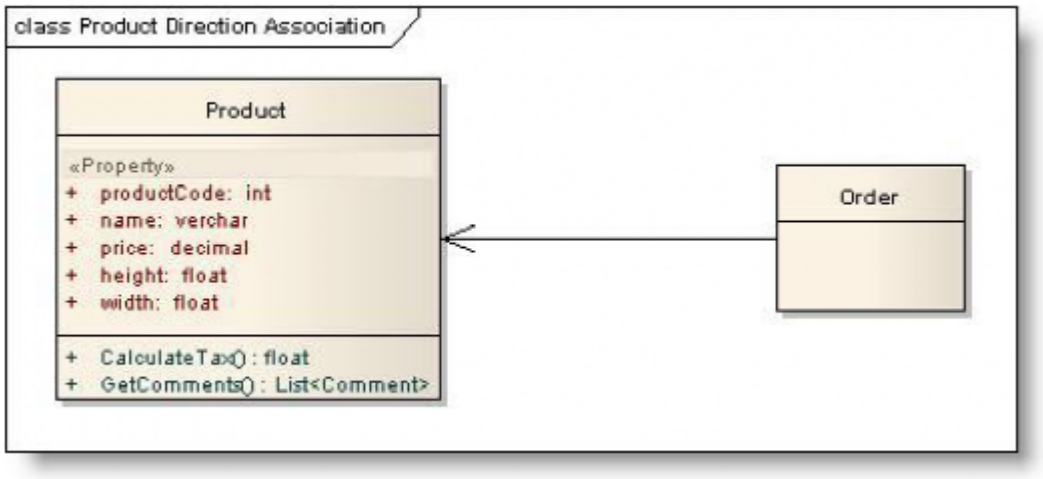


图 5.1 关联图

C++代码:

```
class Order
{
    Public:
     List<Product> order;
void AddOrder(Product product )
```

```
{
            order.Add(product);
}
}

Public Class Product
{
}
```

代码表现为：Order(A1)中有 Product(A2)的变量或者引用

5.1.2 双向关联

B1-B2: 表示 B1 认识 B2，B1 知道 B2 的存在，B1 可以调用 B2 中的方法和属性；同样 B2 也知道 B1 的存在，B2 也可以调用 B1 的方法和属性。

场景：订单和客户，订单属于客户，客户拥有一些特定的订单

类与类之间的双向关联图（图 5.2）

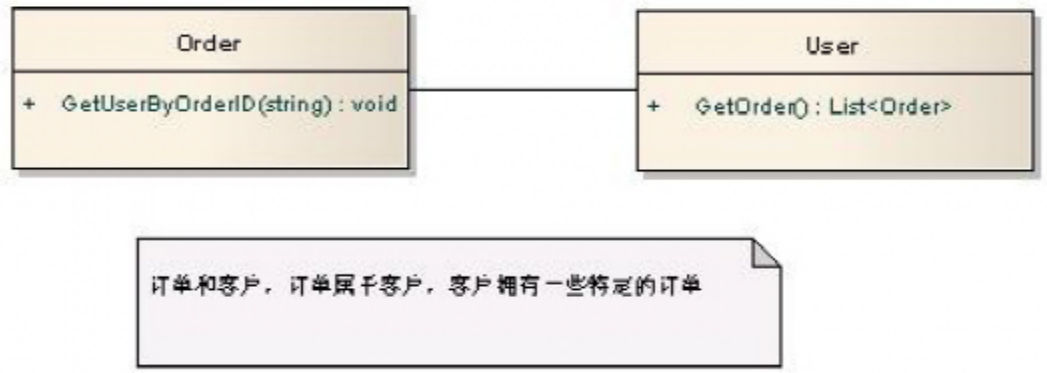


图 5.2 双向关联图

```
class User
{
      Public:
List<Order> GetOrder()
          {
return new List<Order>();
       }

}
Class Order
```

```
{
      Public User GetUserByOrderID(string OrderId )
      {
            Return new User();
}
}
```

5.1.3 自身关联

同一个类对象之间的关联

类与类之间自身关联图（图 5.3）

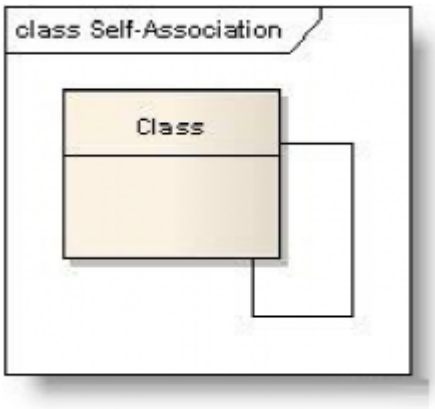


图 5.3 自身关联图

5.1.4、多维关联(N-ary Association)

多个对象之间存在关联

场景：公司雇用员工，同时公司需要支付工资给员工

类与类之间的多维关联图（图 5.4）：

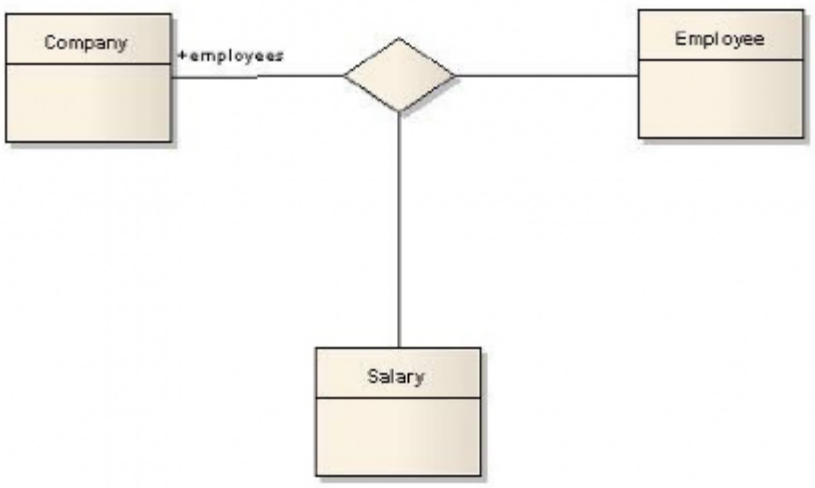


图 5.4 多维关联图

5.2 泛化(Generalization)

类与类的继承关系，类与接口的实现关系。

场景：父与子、动物与人、植物与树、系统使用者与 B2C 会员和 B2E 会员的关系

类与类之间的泛化图（图 5.5）：

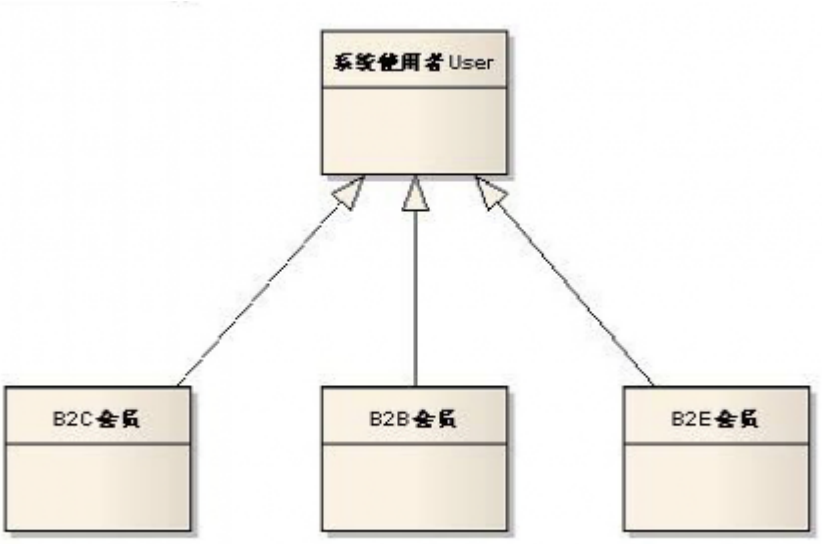


图 5.5 泛化图

系统的使用者包括：B2C 会员、B2B 会员和 B2E 会员。

接口的实现，动物都有吃的行为，而人是动物的一个具体实例,实现具体 Eat 的动作

5.3 依赖(Dependency)

类 A 要完成某个功能必须引用类 B，则 A 与 B 存在依赖关系，依赖关系是弱的关联关系。通常情况下不建议双相依赖，也就是相互引用。例如，本来人与电脑没有关系的，但由于偶然的机会，人需要用电脑写程序，这时候人就依赖于电脑。

类与类的依赖关系图（图 5.6）

图 5.6 依赖关系图

5.4 聚合(Aggregation)

当对象 A 被加入到对象 B 中，成为对象 B 的组成部分时，对象 B 和对象 A 之间为聚合关系。聚合是关联关系的一种，是较强的关联关系，强调的是整体与部分之间的关系。

例如商品和他的规格、样式就是聚合关系。类与类的聚合关系图（图 5.7）

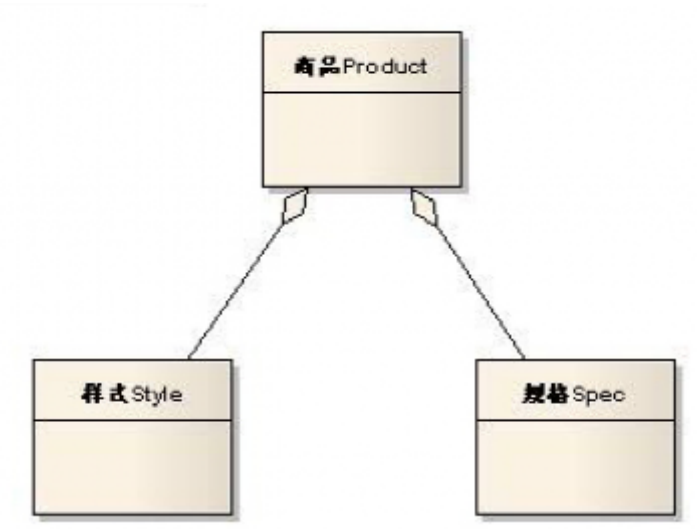


图 5.7 聚合关系图

5.5 组合（Composite）

对象 A 包含对象 B，对象 B 离开对象 A 没有实际意义。是一种更强的关联关系。人包含手，手离开人的躯体就失去了它应有的作用。例如，Window 窗体由滑动条 slider、头部 Header 和工作区 Panel 组合而成。

第六章　C++设计常用设计模式

6.1　Factory 模式

6.1.1 功能:

定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method 使一个类的实例化延迟到其子类。

6.1.2 UML 结构图:

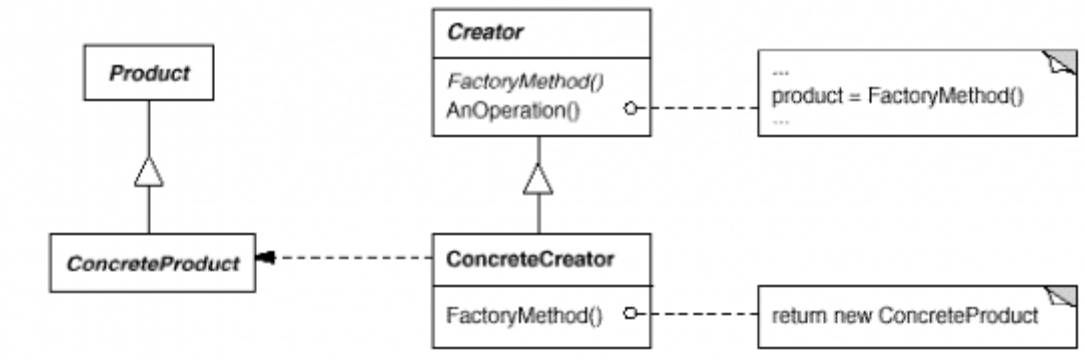


图 6.1 Factory 模式的 UML 结构图

6.1.3 抽象基类

1)Product:创建出来的对象的抽象基类.

2)Factory 创建对象的工厂方法的抽象基类.

接口函数:

1)Creator::FactoryMethod:纯虚函数,由派生类实现,创建出对应的 Product.

6.1.4 解析

在这个模式中,有两个抽象基类,一个是 Product 为创建出来的对象的抽象基类,一个是 Factory 是工厂的抽象基类,在互相协作的时候都是由相应的 Factory 派生类来生成 Product 的派生类,也就是说如果要新增一种 Product 那么也要对应的新增一个 Factory,创建的过程委托给了这 个 Factory.也就是说一个 Factory 和一个 Product 是一一对应的关系。

缺点：核心的主要部分在于一个具体类上，Factory 接收到请求自动判断提供什么产品，但是对于一个 factory 而言，倘若添加一个新的产品，需要做太多的改动，换句话来说，添加一个新的产品必须改动整个 factory 工厂[4]。

6.1.5 示例代码：

```
//view plaincopy to clipboardprint
#include <iostream>
using namespace std;
class product
{
public:
 virtual void produce(){};      //具体的实现交给子类的函数;
 virtual void sell(){};
 product(){};
 virtual ~product(){};
};
class conproduct1:public product
{
public:
 void produce()
 {cout<<"Pro conproduct1.."<<endl;}
 void sell()
 {cout<<"Sel conproduct1.."<<endl;}
};
class conproduct2:public product
```

```
{
public:
 void produce()
 {cout<<"Pro conproduct2.."<<endl;}
 void sell()
 {cout<<"Sel conproduct2.."<<endl;}
};
class Factory              //具体的操作由 factory 控制选择
{
public:
 Factory(){}
 static product *Creatproduct(int a)
 {
  if(a==1) {return new conproduct1();}  //生成相对应的 pro 对象;
  if(a==2) {return new conproduct2();}
 }
};
void main()
{
 Factory *fac=new Factory();
 (fac->Creatproduct(2))->produce();
}
#include <iostream>
using namespace std;
class product
{
public:
 virtual void produce(){};      //具体的实现交给子类的函数;
 virtual void sell(){};
 product(){};
 virtual ~product(){};
};
class conproduct1:public product
{
public:
 void produce()
 {cout<<"Pro conproduct1.."<<endl;}
 void sell()
 {cout<<"Sel conproduct1.."<<endl;}
};
```

```
class conproduct2:public product
{
public:
 void produce()
 {cout<<"Pro conproduct2.."<<endl;}
 void sell()
 {cout<<"Sel conproduct2.."<<endl;}
};
class Factory              //具体的操作由 factory 控制选择
{
public:
 Factory(){}
 static product *Creatproduct(int a)
 {
  if(a==1) {return new conproduct1();}  //生成相对应的 pro 对象;
  if(a==2) {return new conproduct2();}
 }
};
void main()
{
 Factory *fac=new Factory();
 (fac->Creatproduct(2))->produce();
}
```

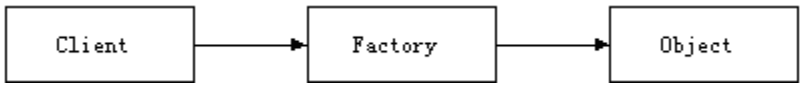
6.2 Abstract Factory 模式

6.2.1 功能

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。

6.2.2UML 结构图

类厂最基本的结构示意图如下：



在实际应用中，类厂模式可以扩充到很复杂的情况，如下图所示（图 6.2）：

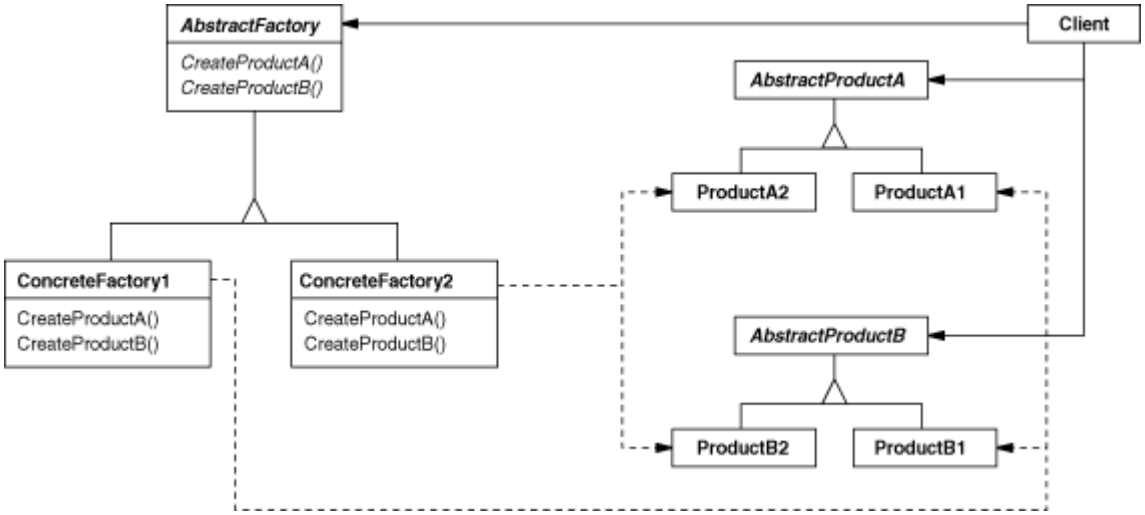


图 6.2 Abstract Factory 模式 UML 结构图

6.2.3 优缺点

优点：(1)封装创建过程。客户不用知道类厂是如何创建类实例的，类厂封闭了所有创建的细节。这样可选择不同的创建方法，增加了灵活性。 (2)将客户与具体类隔离，提高了各自的可重用性。

缺点：Factory 类层次与具体类层次通常是平行的(即一一对应的)。增加一个具体类，一般也要相应地增加一个 factory 类，增加了系统复杂度。

6.2.4 实现

(1)Abstract Factory 类中通常是一组 Factory Method 的集合。个人认为与 Factory Method 模式没有本质区别。

(2)通常可以把工厂作为单件。

6.2.5 示例代码

```
   namespace DesignPattern_AbstractFactory

{

   class AbstractProductA {}; // Product A
   class ProductA1 : public AbstractProductA {};
   class ProductA2 : public AbstractProductA {};
   class AbstractProductB {}; // Product B
      class ProductB1 : public AbstractProductB {};
```

```
   class ProductB2 : public AbstractProductB {};
   class AbstractFactory
   {
   public:
      virtual AbstractProductA* CreateProductA() = 0 ;// 创建 ProductA
      virtual AbstractProductB* CreateProductB() = 0 ;// 创建 ProductB
} ;
   class ConcreteFactory1 : public AbstractFactory
   {
   public:
         virtual AbstractProductA* CreateProductA() { return new
ProductA1() ; }
         virtual AbstractProductB* CreateProductB() { return new
ProductB1() ; }
       static ConcreteFactory1* Instance() { static ConcreteFactory1
instance ; return &instance ; }
protected:
      ConcreteFactory1() {}
   private:
      ConcreteFactory1(const ConcreteFactory1&) ;
      ConcreteFactory1& operator=(const ConcreteFactory1&) ;
   } ;
   class ConcreteFactory2 : public AbstractFactory
   {
   public:
         virtual AbstractProductA* CreateProductA() { return new
ProductA2() ; }
         virtual AbstractProductB* CreateProductB() { return new
ProductB2() ; }
       static ConcreteFactory2* Instance() { static ConcreteFactory2
instance ; return &instance ; }
   protected:
      ConcreteFactory2() {}
   private:
      ConcreteFactory2(const ConcreteFactory2&) ;
      ConcreteFactory2& operator=(const ConcreteFactory2&) ;
   } ;
}
```

客户端代码：

```
{
    using namespace DesignPattern_AbstractFactory ;
    // 第一种创建方法

    AbstractFactory *pFactory = ConcreteFactory1::Instance() ;
    AbstractProductA *pProductA = pFactory->CreateProductA() ;
    AbstractProductB *pProductB = pFactory->CreateProductB() ;

    // 第二种创建方法
    pFactory = ConcreteFactory2::Instance() ;
    pProductA = pFactory->CreateProductA() ;
    pProductB = pFactory->CreateProductB() ;
}
```

6.2.6 实例

最早知道类厂的概念是在 COM 中，但当时也没想到这是如此重要的一种模式，在许多其他模式中都可以用到类厂模式。 COM 中不能直接创建组件，这也是由 COM 的一个特性决定的：即客户不知道要创建的组件的类名。

6.3 Composite 模式

6.3.1 功能

表示“部分-整体”关系，并使用户以一致的方式使用单个对象和组合对象。

6.3.2 UML 结构图

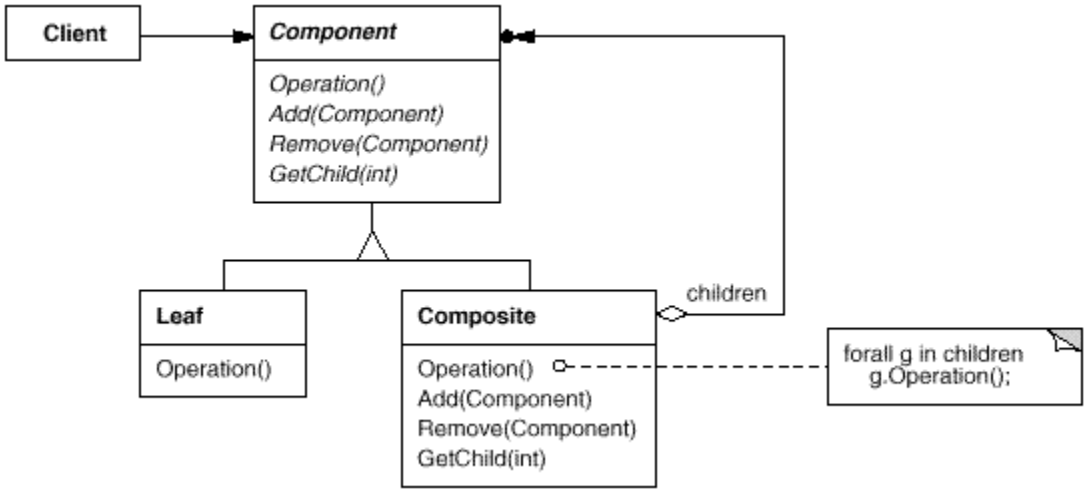


图 6.3 Composite 模式 UML 结构图

上图 6.3 中，也可以做些扩展，根据需要可以将 Leaf 和 Composite 做为抽象基类，从中派生出子类来。

6.3.3 优缺点

优点：对于 Composite 模式，也许人们一开始的注意力会集中在它是如何实现组合对象的。但 Composite 最重要之处在于用户并不关心是组合对象还是单个对象，用户将以统一的方式进行处理，所以基类应是从单个对象和组合对象中提出的公共接口。

缺点：Composite 最大的问题在于不容易限制组合中的组件。

6.3.4 实现

有时需要限制组合中的组件，即希望一个 Composite 只能有某些特定的 Leaf。这个问题我是用多继承和动态类型转换来解决的。假如组合对象 Composite1 只能包含单个对象 ConcreteLeaf1，Composite2 可以包含单个对象 ConcreteLeaf1 和 ConcreteLeaf2。如下图 6.4 所示：

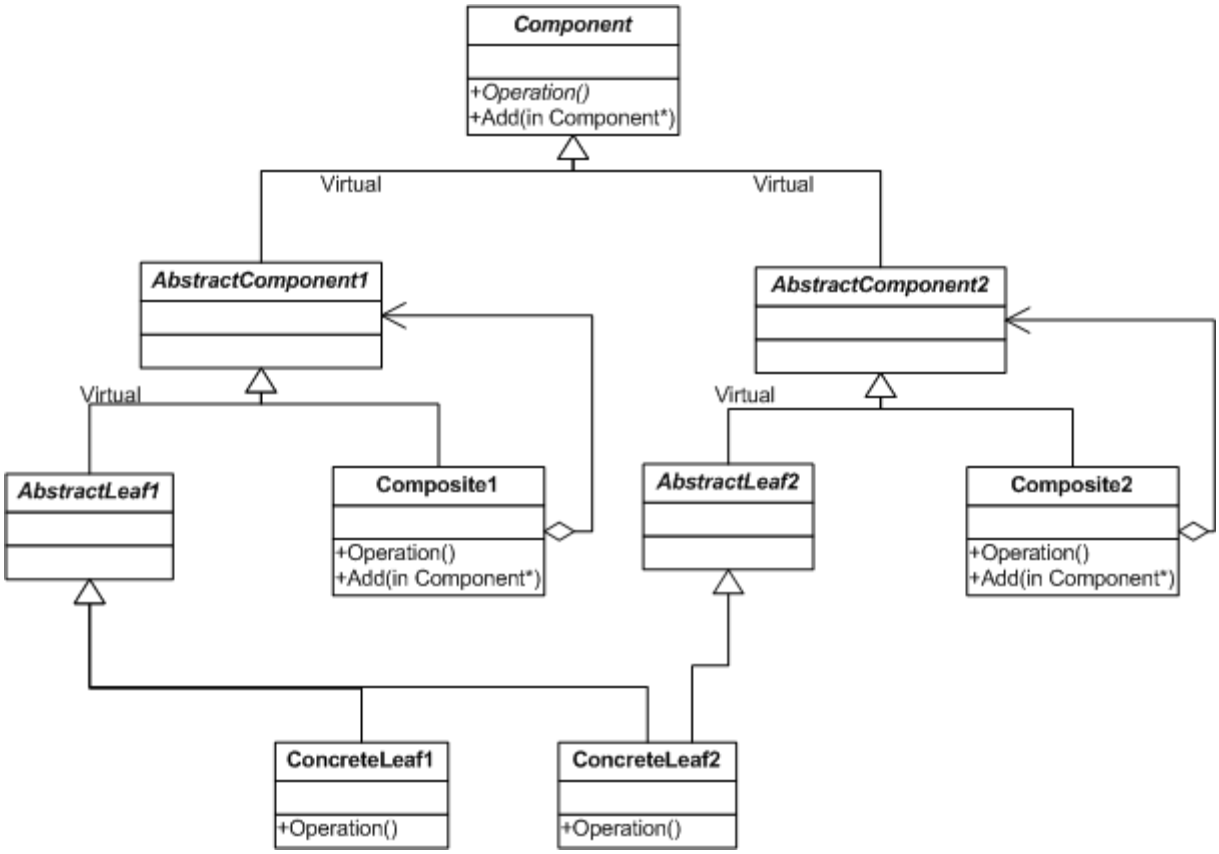


图 6.4

上图中的类层次比较多，使用了 AbstractLeaf1 和 AbstractLeaf2，但没使用 AbstractComposite1 和 AbstractComposite2，这个并不重要，也可以把 AbstractLeaf1 和 AbstractLeaf2 去掉，这个并不重要，可以根据具体情况决定要不要。

简单的代码实现如下：

```
    namespace DesignPattern_Composite
{
class Component
{
public:
virtual void operation() = 0 ;
virtual void Add(Component*) {}
} ;
    class AbstractComponent1 : virtual public Component {} ;
    class AbstractLeaf1 : virtual public AbstractComponent1 {} ;
    class Composite1 : public AbstractComponent1
{
public:
virtual void operation() { /* do operation */ }
virtual void Add(Component*) ;
```

```
} ;
void Composite1::Add(Component *p)
{
AbstractComponent1 *pc1 = dynamic_cast＜ABSTRACTCOMPONENT1*＞(p) ;
if (pc1 == NULL) return ;
// do add operation
}
    class AbstractComponent2 : virtual public Component {} ;
    class AbstractLeaf2 : virtual public AbstractComponent2 {} ;
    class Composite2 : public AbstractComponent2
{
public:
virtual void operation() { /* do operation */ }
virtual void Add(Component*) ;
} ;
void Composite2::Add(Component *p)
{
AbstractComponent2 *pc2 = dynamic_cast＜ABSTRACTCOMPONENT2*>(p) ;
if (pc2 == NULL) return ;
// do add operation
}
    class ConcreteLeaf1 : public AbstractLeaf1
{
public:
virtual void operation() { /* do operation */ }
} ;
    class ConcreteLeaf2 : public AbstractLeaf1, public AbstractLeaf2
{
public:
virtual void operation() { /* do operation */ }
} ;
}
```

客户端代码：

```
 {
using namespace DesignPattern_Composite ;
    Component *pc1 = new ConcreteLeaf1() ;
Component *pc2 = new ConcreteLeaf2() ;
Component *pc3 = new Composite1() ;
Component *pc4 = new Composite2() ;
pc3->Add(pc1) ; // ok
```

```
pc3->Add(pc2) ; // ok
pc3->Add(pc3) ; // ok
pc3->Add(pc4) ; // fail
pc4->Add(pc1) ; // fail
pc4->Add(pc2) ; // ok
pc4->Add(pc3) ; // fail
pc4->Add(pc4) ; // ok
}
```

有两点需要注意，一是因为用了多继承，所以需要使用 virtual inheritance。二是要用 dynamic_cast 来判断是否允许组合该组件。

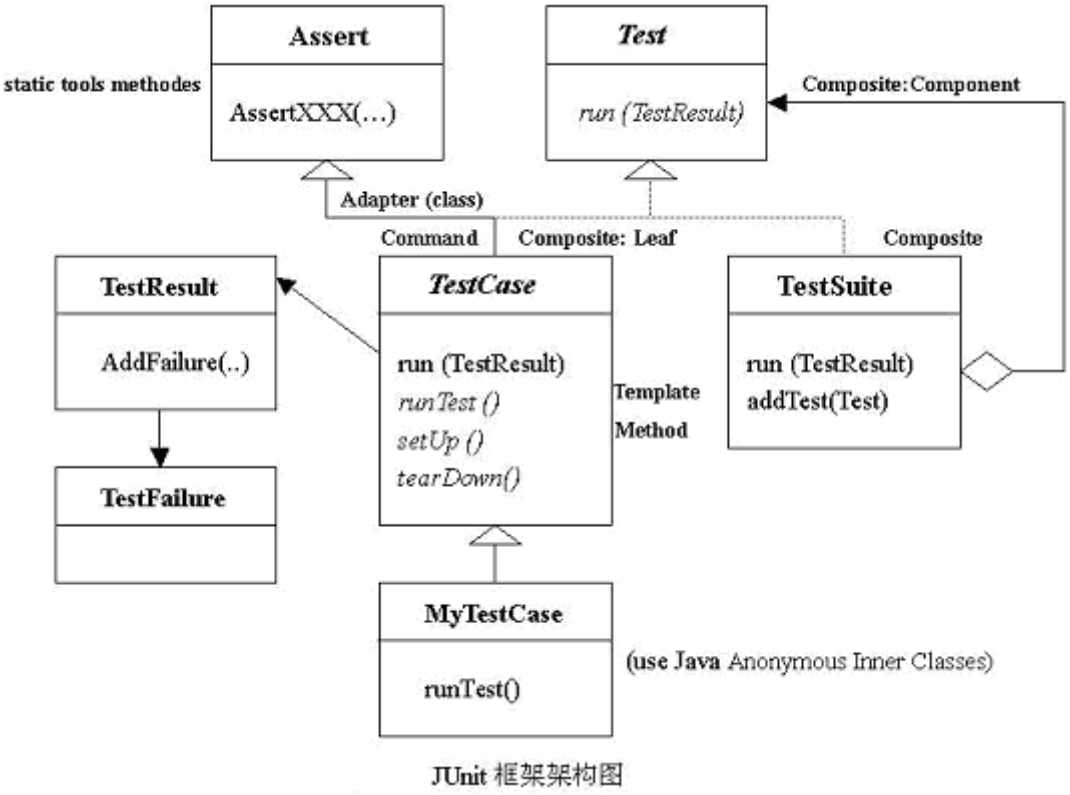
6.3.5 示例代码

```
   namespace DesignPattern_Composite
{
// class Component
class Component
{
public:
virtual void Operation() = 0 ;
virtual void Add(Component*) {}
} ;
   // class Leaf
class Leaf : public Component
{
public:
virtual void Operation() {}
} ;
   // class Composite
class Composite : public Component
{
public:
virtual void Add(Component *p) { _list.push_back(p) ; }
virtual void Operation()
{
vector< Component* >::const_iterator it ;
for (it = _list.begin(); it != _list.end(); it++)
(*it)->Operation() ;
}
private:
```

```
vector< Component* > _list ;
} ;
}
```

6.3.6 实例

(1)JUnit 中就用的是 Composite 模式。



JUnit 框架架构图

6.4 Proxy 模式

6.4.1 功能

为其他对象提供一种代理以控制对这个对象的访问。

6.4.2 UML 结构

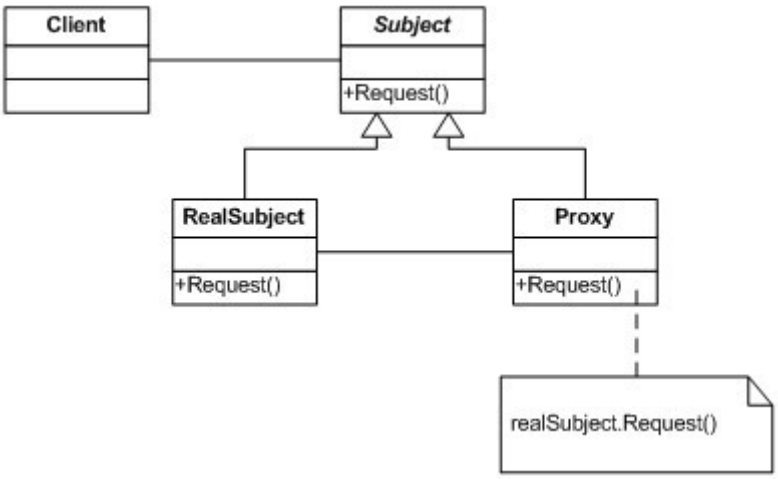


图 6.5 Proxy 模式 UML 结构图

1. Subject 定义了 RealSubject 和 Proxy 的公用接口，使得任何调用 RealSubject 的地方都可以调用 Proxy。

2. RealSubject 是代理的实体类。

3. Proxy 保存一直指向代理实体（RealSubject）的指针。Proxy 与 Subject 的接口相同，代理就可以替代实体；Proxy 负责 RealSubject 的创建和管理。

6.4.4 要点

根据代理模式效果的不同，有如下几种代理应用：

远程代理（Remote Proxy）

为一个位于不同的地址空间的对象提供隐藏实现。这个不同的地址空间可以是在本机器中，也可是在另一台机器中。远程代理又叫做大使（Ambassador）。

远程调用和 webservice 服务就是典型的远程代理模式，客户并不知道网络访问细节（访问本地还是远程网络），代理对象对外屏蔽了网络通讯细则。

虚拟代理（Virtual Proxy）

根据需要创建一个资源消耗较大的对象，使得此对象只在需要时才会被真正创建。使用虚拟代理模式的好处就是代理对象可以在必要的时候才将被代理的对象加载；代理可以对加载的过程加以必要的优化[5]。

文档中有非常大的图片，为了保证文档打开速度，在打开文档的时候并不真正创建图片对象，只有当用到图片的时候，才会去创建图片对象。

写时拷贝代理（Copy-on-write Proxy）

写时拷贝是虚拟代理的一种特殊应用,可以大幅降低拷贝大对象的开销。写时拷贝可以延迟一个对象的拷贝操作到客户调用里。一般来说，对象的深度拷贝是一个高开销的动作，该代理可以让这个动作延迟，只有对象被修改的时候才被拷贝一份。

Linux 创建进程就用到了 Copy-on-write。fork 派生子进程的时候，若子进程拷贝父进程的整个地址空间，会使得创建进程的速度非常慢而且效率低下。而实事上，子进程几乎不必读和修改父进程所拥有的资源， Copy-on-write 允许父子进程拥有相同的物理页，只有两者中有一个试图修改一个物理页时，内核就把要修改的物理页内容拷贝到一个新的物理页，并把这个新的物理页地址分配给正在写的进程[6]。

保护代理（Protection Proxy）

控制对 RealSubject 对象的访问。可以实现给不同的用户提供不同级别的使用权限。保护代理可以在运行时间对用户的有关权限进行检查，然后在核实后决定将调用传递给被代理的对象。

智能引用代理（Smart Reference Proxy）

取代了简单的指针(->, *)，在对象访问时提供一些附加操作：对指向实际对象的引用计数，引用计数为 0 时，可以自动释放资源（智能指针）；当第一次引用一个持久的对象，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。

缓存代理（Cache Proxy）

为创建时高开销的对象提供缓存，以供多客户端共享。如数据库代理，客户端进行 Sql 查询时，并不访问数据库，而是通过与数据库代理之间进行交互，如果数据库代理中没有缓存数据，才真正去数据库查询。

防火墙代理（Firewall Proxy）

保护目标对象不受某些不良客户端的侵害。

同步代理（Synchronization Proxy）

使几个用户能够同时使用一个对象而没有冲突，提供并发控制。

C++语言可通过重载->和*运算符实现类的代理。

Image* ImagePtr::operator-> () {

```
        return LoadImage(); //装载对象并返回对象指针
    }
```

这样对象被引用时，就装载对象，起到了代理的作用。但这种方式的问题是，只要引用就加载，而不能指定在调用某个方法（如 Draw）的时候才加载。

6.4.5 实现

每种代理模式的应用场景都非常之多。共同点是代理起到了客户端与真实对象之间中间层的作用。通过对中间层进行改造，到达各种指责目的。

源码通过缓存代理应用来举例说明。

示例代码

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
//代理抽象基类
class CProxy
{
public:
  CProxy() {};
  virtual ~CProxy() {};

  virtual string Get(const string &strKey) = 0;
};
//代理的实际目标
class CDataServer : public CProxy
{
public:
  CDataServer() {};
  virtual ~CDataServer() {};

  virtual string Get(const string &strKey)
  {
    //TODO: query db here…
    cout << "real server query db…" << endl;
    return strKey + strKey;
  }
};
```

```
//代理类
class CCacheProxy : public CProxy
{
public:
  CCacheProxy() : m_dataServer(new CDataServer)
  {};
  virtual ~CCacheProxy()
  {
    delete m_dataServer;
  };
  virtual string Get(const string &strKey)
  {
    //target cache
    map<string, string>::iterator itr = m_mapValue.find(strKey);
    if (itr != m_mapValue.end())
    {
      cout << "target cache…" << endl;
      return itr->second;
    }
    //request from real server
    string strVal = m_dataServer->Get(strKey);
    m_mapValue[strKey] = strVal;
    return strVal;
  }

protected:
  CDataServer*  m_dataServer;
  map<string, string> m_mapValue; //cache
};
int main()
{
  //client 并不知道 CDataServer
  CProxy* p = new CCacheProxy;
  string strKey = "ab";
  string strVal = p->Get(strKey);
  cout << "key: " << strKey << ", value: " << strVal << endl;
  strKey = "abcd";
  strVal = p->Get(strKey);
  cout << "key: " << strKey << ", value: " << strVal << endl;
  strKey = "ab";
  strVal = p->Get(strKey);
```

```
    cout << "key: " << strKey << ", value: " << strVal << endl;

    return 0;
}
```

6.5 Singleton 模式

6.5.1 功能

保证一个类仅有一个实例。

6.5.2 UML 结构图

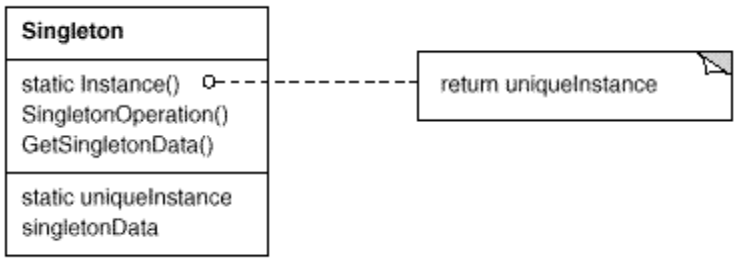


图 6.7 Singleton UML 结构图

6.5.3 优缺点

Singleton 模式是做为"全局变量"的替代品出现的。所以它具有全局变量的特点：全局可见、贯穿应用程序的整个生命期，它也具有全局变量不具备的性质：同类型的对象实例只可能有一个。

6.5.4 实现

Singleton 定义如下：

```
   class Singleton
{
public:
static Singleton* Instance() ;
protected:
Singleton() {}
private:
```

```
static Singleton *_instance ;
Singleton(const Singleton&) ;
Singleton& operator=(const Singleton&) ;
} ;
    Singleton* Singleton::_instance = NULL ;
    Singleton* Singleton::Instance()
{
(_instance == NULL) ? _instance = new Singleton() : 0 ; //lazy
initialization
return _instance ;
}
```

(1)因为返回的是指针，为防止用户调用 delete 函数，可把 static Singleton *_instance；改为在 Instance()中定义 static Singleton _instance。这样显然更安全，同时也具有 lazy initialization 的特性(即第一次访问时才创建)。

(2)假设需要从 Singleton 派生子类，而子类也需要有同样的性质，既只能创建一个实例。我觉得，这很难办。根本原因在于 Instance()函数不是虚函数，不具有多态的性质。一种常用方法是把 Instance()函数移到子类中，这时就只能用 static Singleton *_instance，而不能用 static Singleton _instance 了，除非把_instance 也要移到子类，无论怎么做都不优雅。另一种方法是用模板。具体用什么方法，只能根据实际情况权衡。

6.5.5 示例代码

(1)没子类的情况

```
    namespace DesignPattern_Singleton
{
    class Singleton
{
public:
static Singleton* Instance() { static Singleton _instance ; return
&_instance ; }
protected:
Singleton() {}
private:
Singleton(const Singleton&) ;
Singleton& operator=(const Singleton&) ;
```

```
} ;
}
```

(2)有子类的情况

```
    方法一：
namespace DesignPattern_Singleton
{
// class Singleton
class Singleton
{
protected:
Singleton() {}
static Singleton *_instance ;
private:
Singleton(const Singleton&) ;
Singleton& operator=(const Singleton&) ;
} ;
Singleton* Singleton::_instance = NULL ;
    // class ConcreteSingleton
class ConcreteSingleton : public Singleton
{
public:
static Singleton* Instance() ;
protected:
ConcreteSingleton() {}
} ;
    Singleton* ConcreteSingleton::Instance()
{
(_instance == NULL) ? _instance = new ConcreteSingleton() : 0 ;
return _instance ;
}
}
    客户端代码：
{
using namespace DesignPattern_Singleton ;
Singleton *p = ConcreteSingleton::Instance() ;
}
```

方法二：

```
namespace DesignPattern_Singleton
{
// class Singleton
```

```
class Singleton
{
protected:
Singleton() {}
private:
Singleton(const Singleton&) ;
Singleton& operator=(const Singleton&) ;
} ;
    // class ConcreteSingleton
class ConcreteSingleton : public Singleton
{
public:
static Singleton* Instance() { static ConcreteSingleton _instance ;
return &_instance ; }
protected:
ConcreteSingleton() {}
} ;
}
```

客户端代码：

```
{
using namespace DesignPattern_Singleton ;
Singleton *p = ConcreteSingleton::Instance() ;
}
```

方法三：

```
namespace DesignPattern_Singleton
{
template < class T >
class Singleton
{
public:
static T* Instance() { static T _instance ; return &_instance ; }
protected:
Singleton() {}
private:
Singleton(const Singleton &) ;
Singleton& operator=(const Singleton&) ;
} ;
    class ConcreteSingleton : public Singleton< ConcreteSingleton >
{} ;
}
```

```
    客户端代码
{
using namespace DesignPattern_Singleton ;
    ConcreteSingleton *p = ConcreteSingleton::Instance() ;
}
```

第七章　设计模式在软件开发过程中的使用——分区 LED 显示屏取模软件

7.1 软件概述

室外 LED 显示屏是广泛使用在广告、公共信息发布等场合，具有可视范围大，清晰度高，成本较低的等优点。LED 显示屏主要分为两个部分，一是靠电子电路支持的硬件设备，其二是为了保证数据更新方便而专门设计开发的上位机图文处理软件，通过电脑编辑图文信息并发布到显示驱动卡，由显示驱动卡完成信息的显示。该软件就是一个针对分区 LED 显示屏的上位机图文处理软件，通过该软件处理图文并按照一定的通讯协议与驱动卡交换信息，实现控制 LED 显示屏的显示内容和显示效果。

7.2 开发目标

1. 系统应具有实用性、可靠性和适用性。
2. 具有分区的管理功能。
3. 能够对各个分区进行单独编辑。
4. 将各分区的图片或文字信息以及显示效果按照通讯协议转换成对应指令并发送到下位机显示。
5. 通过串口与下位机交换信息并控制下位机状态。
6. 预留一定的接口，方便软件的二次开发与功能扩展。

7. 尽量保证软件在常用操作系统的可移植性。

7.3 软件详细功能

1. 分区添加、删除
2. 分区属性设置，包括分区类别，分区大小、背景、分区显示的动态效果
3. 分区的图片编辑
4. 分区的文字编辑
5. 控制卡扫描设置
6. 显示卡的控制，包括开关机、亮度设置、时间设置等
7. 通过串口发送显示信息到下位机

7.4 设计模式在该软件中的使用

7.4.1 单例模式

为了保证系统的可操作性，只允许一个用户启动一次该系统，若系统启动后，当用户再次试图启动则应该对用户的操作提示。这里涉及两个基本的问题，一是在程序的第二个实例启动时，如何发现该程序已有一个实例在运行，而是如何将第一个实例激活，而第二个实 例退出。对于第一个问题，可以通过给应用程序设置信号量，实例启动时首先检测该信号量,如已存在，则说明程序已运行一个实例。第二个问题的难点是获取第一个实例的主窗对象指针或句柄，然后便可用 SetForegroundWindow 来激活。虽然 FindWindow 函数能寻找正运行着的窗口,但该函数要求指明所寻找窗口的标题或窗口类名,不是实现通用方法的途径。我们可以用 Win32 SDK 函数 SetProp 来给应用程序主窗设置一个特有的标记。用 GetDesktopWindow 可以获取 Windows 系统主控窗口对象指针或句柄，所有应用程序主窗都可看成该窗口的子窗口，即可用 GetWindow 函数来获得它们的对象指针或句柄。用 Win32 SDK 函数 GetProp 查找每一应用程序主窗是否包含有我们设置的特定标记便可确定它是否我们要寻找的第一个实例主窗。使第二个实例退出很简单，只要让其应用程序对象的 InitInstance 函数返回 FALSE 即可。此外，当主窗口退出时，应用 RemoveProp 函数删除我们为其设置的标记。下面的

InitInstance、OnCreate 和 OnDestroy 函数代码将实现上述的操作：

```
BOOL CLedWndApp::InitInstance()
{
// 用应用程序名创建信号量
HANDLE hSem = CreateSemaphore(NULL, 1, 1, m_pszExeName);

    // 信号量已存在?

    // 信号量存在，则程序已有一个实例运行
    if (GetLastError() == ERROR_ALREADY_EXISTS)
    {
        // 关闭信号量句柄
        CloseHandle(hSem);
        // 寻找先前实例的主窗口
        HWND              hWndPrevious              =              ::
GetWindow(::GetDesktopWindow(),GW_CHILD);
        while (::IsWindow(hWndPrevious))
        {
            // 检查窗口是否有预设的标记?
            // 有，则是我们寻找的主窗
            if (::GetProp(hWndPrevious, m_pszExeName))
            {

                // 主窗口已最小化，则恢复其大小
                if (::IsIconic(hWndPrevious))
                    ::ShowWindow(hWndPrevious,SW_RESTORE);
                // 将主窗激活
                ::SetForegroundWindow(hWndPrevious);
                // 将主窗的对话框激活
                ::SetForegroundWindow(
                ::GetLastActivePopup(hWndPrevious));
                // 退出本实例
                return FALSE;
            }
            // 继续寻找下一个窗口
            hWndPrevious = ::GetWindow(hWndPrevious,GW_HWNDNEXT);
        }
```

```
        // 前一实例已存在，但找不到其主窗
        // 可能出错了
        // 退出本实例
        return FALSE;
}
```

7.4.2 工厂模式

软件中的分区是独立编辑的，为了使用和开发方便，将分区分为三种分区：图片类、文字类和图形类（图形类未开发完成），三种分区类的接口功能相同的，只是显示方式和处理方式不一样。然而对于上层读取分区数据和控制信息而言，各类分区之间的方式是不一样的，因此需要定义一个总的分区类，在派生出三种不同的分区类别，将详细的操作留到具体的分区类去实现，分区基类定义如下：

```
class CLayer
{
public:
 CLayer(unsigned int index,const CRect & re);
 virtual ~CLayer();
public:
 CRect   m_Rect;//图层矩形区域 自动初始化
 unsigned int m_index;//叠放次序
 BOOL m_ShowLayer;//是否显示图层
 CString m_name;//图层名称
 int m_LayerType;//图层属性 文字 图像 图形
protected:
 CEffect * m_pEFCT;//效果属性的类别 互相关联
public:
 const CRect * GetLayerRect();//获取图层矩形区域
 virtual unsigned long GetLayerData(COLORREF * & pd,CDC * dc)=0;//获取一个区域的 RGB 数据 返回数据长度
 virtual void ShowLayer(CDC * dc)=0;//显示图层数据 等待之类实现
 virtual void ShowMsg(CComProDlg * dlg)=0;//更新公共对话框的属性显示
 virtual void UpDateThing(CComProDlg * dlg)=0;//将对话框显示的数据存入对象
 virtual void UpDateEFT(CComProDlg * dlg)=0;//将对话框动态属性存入对象
 virtual void UpDateFunkThing(CComProDlg * dlg)=0;//将对话左边乱七八糟的数据存
```

```
入对象
    static int m_LayerCount;
   };
```

可以看见，分区类中基类的操作函数是纯虚函数，这就要求之类必须实现对应的函数

那么文字分区类和图片分区类的定义如下：

class CTextLayer:public CLayer//继承自分区类

{……}

class CImgLayer:public CLayer//继承自分区类

{……}

那么利用工厂模式，在创建分区时，根据制定的分区类别创建对应的分区对象

```
   switch(layertype)
       {
           case LAYER_TYPE_TEXT:pLayer=new CTextLayer(index,re);break;
           case LAYER_TYPE_IMG:pLayer=new CImgLayer(index,re);break;
           case                              LAYER_TYPE_SHAPE:pLayer=new
CShapeLayer(index,re);break;
           default:return false;
       }
       this->m_LayerList.InsertAt(index,pLayer);
```

m_LayerLis 是一个由所有分区对象指针构成的指针链表，当需要读取一个分区的数据时，只需要获得对应的指针，利用多态即可完成对应的操作，如下是获取一个分区显示数据的函数代码：

//获取一个区域的 RGB 数据 返回数据长度

```
   unsigned long CLED1Doc::GetLayerData(unsigned int index,COLORREF
* pd,CDC * dc)
   {
```

```
        if(index<0)
            return 0;
        CLayer * pl=(CLayer *)this->m_LayerList.GetAt(index);//获取分区
对象指针
        if(pl==NULL)
            return 0;
        return pl->GetLayerData(pd,dc);
    }
```

可以看到，在代码中并不需要指定调用某个分区类的函数，获取的是一个基类的指针，利用多态，即可调用相应子类的操作函数。

7.4.3 文档视图模型

在程序中，需要管理好各个分区的数据信息，并将这些信息通过视图界面合理的展现给用户，这里使用了 MFC 中的文档视图框架：在文档中保存分区和配置信息，而视图只负责这里信息的显示以及界面交互，当通过试图修改数据时，有试图通知文档修改对应数据。如下是文档类的定义：

//以下函数为自定义函数

```
    class CLED1Doc : public CDocument
    {
    public:
        BOOL AddLayer(unsigned int index,int layertype,const CRect &
re=CRect(0,0,128,64));//添加图层 由视图类调用
        BOOL DelLayer(unsigned int index);//删除图层由视图类调用
        void ExchangeLayer(unsigned int la,unsigned int lb);//交换图层

        void SetLayerDisplay(unsigned int index,BOOL code);//设置图层的
显示与隐藏
        unsigned long GetLayerData(unsigned int index,COLORREF *
pd,CDC * dc);//获取一个区域的 RGB 数据 返回数据长度

    public:
        //开始定义显示屏数据
        unsigned int m_width;//LED 的矩形大小
        unsigned int m_height;
```

```
        COLORREF m_bgcolor;//背景颜色
        CPtrArray m_LayerList; //分区对象指针
        CConfig m_config;//配置信息
    private:
        BOOL m_ifshownewDlg; //基本上没有的                只是为了第一次打开
的时候不显示出新建图层
    };
```

7.5、软件设计

7.5.1 开发环境

Microsoft 的集成开发环境 Visual C++ 6.0，操作系统为 Windows XP SP3

7.5.2 界面设计

程序界面使用 Visuall C++6.0 的单文档框架，并使用 VC6 自带的对话框编辑器设置出各个对话框，然后给每个对话框关联相应的类并添加成员变量.。

主界面如下

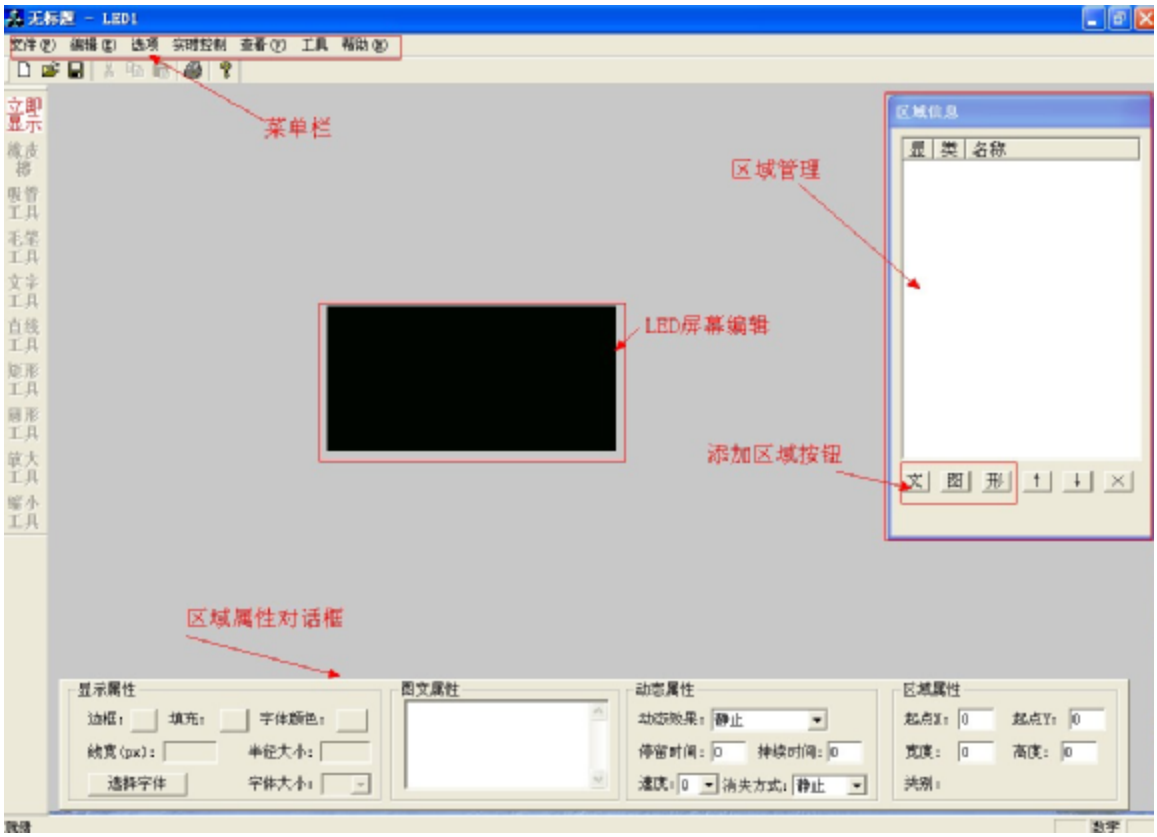


图 7.1 软件运行主界面

显示屏属性设置对话框

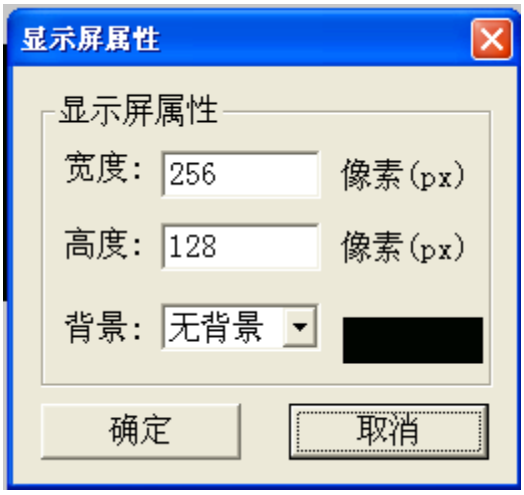


图 7.2 显示屏属性设置对话框

LED 控制卡属性对话框

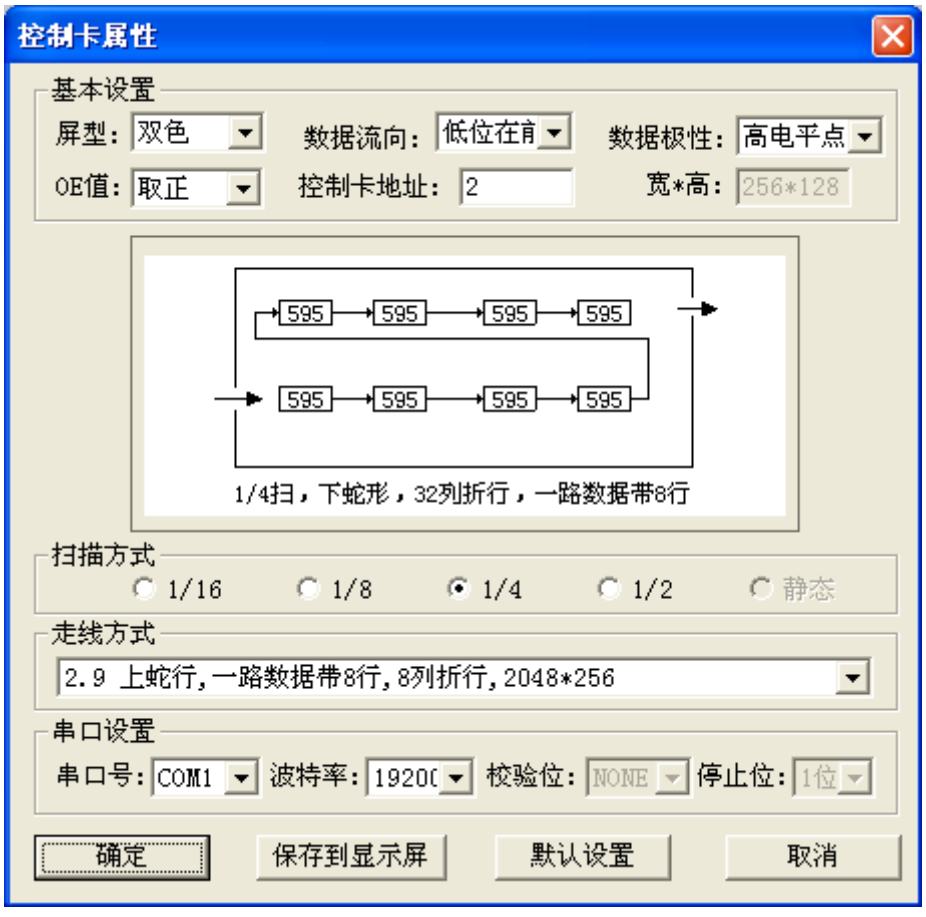


图 7.2 控制卡属性设置对话框

第八章 总结

2009 年 12 月，我开始了我的毕业论文工作，时至今日，论文基本完成。从最初的茫然，到慢慢的进入状态，再到对思路逐渐的清晰，整个写作过程难以用语言来表达。历经了几个月的奋战，紧张而又充实的毕业设计终于落下了帷幕。回想这段日子的经历和感受，我感慨万千，在这次毕业设计的过程中，我拥有了无数难忘的回忆和收获。

3 月初，在与导师的交流讨论中我的题目定了下来，是：C++设计模式及其应用。当选题报告，开题报告定下来的时候，我当时便立刻着手资料的收集工作中，当时面对浩瀚的书海真是有些茫然，不知如何下手。我将这一困难告诉了导师，在导师细心的指导下，终于使我对自己现在的工作方向和方法有了掌握。

在搜集资料的过程中，我在学校图书馆搜集资料，还在网上查找各类相关资料，将这些宝贵的资料全部记在笔记本上，尽量使我的资料完整、精确、数量多，这有利于论文的撰写。然后我将收集到的资料仔细整理分类，及时拿给导师进行沟通。

4 月初，资料已经查找完毕了，我开始着手论文的写作。在写作过程中遇到困难我就及时和导师联系，并和同学互相交流，请教专业课老师。在大家的帮助下，困难一个一个解决掉，论文也慢慢成型。

4 月底，论文的大致思路已经完成。在设计初期，由于没有设计经验，觉得无从下手，空有很多设计思想，却不知道应该如何写，经过导师的指导，我的设计渐渐有了头绪，通过查阅资料，逐渐确立系统方案。

当我终于完成了所有打字、绘图、排版、校对的任务后整个人都很累，但同时看着电脑荧屏上的毕业设计稿件我的心里是甜的，我觉得这一切都值了。这次毕业论文的制作过程是我的一次再学习，再提高的过程。在论文中我充分地运用了大学期间所学到的知识。

我不会忘记这难忘的几个月的时间。毕业论文的制作给了我难忘的回忆。在我徜徉书海查找资料的日子里，面对无数书本的罗列，最难忘的是每次找到资料时的激动和兴奋；记忆最深的是每一步小小思路实现时那幸福的心情；为了论文我曾赶稿到深夜，但看着亲手打出的一字一句，心里满满的只有喜悦毫无疲惫。这段旅程看似荆棘密布，实则蕴藏着无尽的宝藏。我从资料的收集中，掌握了很

多关于电子商务的知识，让我对我所学过的知识有所巩固和提高，并且让我对当今电子商务的最新发展技术有所了解。在整个过程中，我学到了新知识，增长了见识。在今后的日子里，我仍然要不断地充实自己，争取在所学领域有所作为。

在这次毕业设计中也使我们的同学关系更进一步了，同学之间互相帮助，有什么不懂的大家在一起商量，听听不同的看法对我们更好的理解知识，所以在这里非常感谢帮助我的同学。

在此更要感谢我的导师和专业老师，是你们的细心指导和关怀，使我能够顺利的完成毕业论文。在我的学业和论文的研究工作中无不倾注着老师们辛勤的汗水和心血。老师的严谨治学态度、渊博的知识、无私的奉献精神使我深受启迪。从尊敬的导师身上，我不仅学到了扎实、宽广的专业知识，也学到了做人的道理。在此我要向我的导师致以最衷心的感谢和深深的敬意。

参考文献

[1]沙洛维，特罗特 <<设计模式解析>>[M](第 2 版)》 人民邮电出版社 2010 年

[2]伽玛 等著《设计模式：可复用面向对象软件的基础》[M] 机械工业出版社 2000 年

[3]弗里曼《Head First 设计模式》[M] 中国电力出版社 （美）2009 年

[4] 程杰 《大话设计模式》[M] 清华大学出版社 2007 年

[5]王少锋 《面向对象技术 UML 教程》[M]北京清华大学出版社 2004 年

[6]巴拉赫，兰宝 《UML 面向对象建模与设计（第 2 版）》[M] 上海人民邮电出版社 2010 年

[7] 拉曼《UML 和模式应用（原书第 3 版）》[M] 机械工业出版 2006 年

[8]普拉塔《C++Primer Plus（第五版）中文版》[M]人民邮电出版社 2006 年

[9]钱能 《C++程序设计教程（第二版）》 [M]清华大学出版社 2005 年

[10] 侯俊杰 《深入浅出 MFC（第 2 版）》[M]华中科技大学出版社 2001 年

[11] 明日科技《MFC 程序开发参考大全》[J] 人民邮电出版社 2007 年

[12]明日科技 《Visual C++开发技术大全》[J]人民邮电出版社 2009 年

[13]（美）弗里曼（Freeman,E.） 等著.《深入浅出设计模式》[M] 南京： 东南大学出版社 2009 年 01 月

致谢

在本次毕业设计中，我由衷地感谢蔡立老师给我指导。他无论在理论上还是在实践中，都给与我很大的帮助，使我得到不少的提高这对于我以后的工作和学习都有一种巨大的帮助，感谢他耐心的辅导。

另外，在论文的撰写和系统的设计过程中学院的其他老师也给于我很大的帮助，帮助解决了不少的难点，使得论文及设计及时开发完成，还有班上的同学、我的朋友们给与我不少帮助和支持，感谢所有计算机与信息工程学院的各位老师和领导的关怀，在学校的四年中，我不仅学到了相关的专业知识，也学会了把知识应用到实践当中的能力。在此向所有给予我帮助的领导、老师、同学以及朋友表示真挚的感谢。

附：学位论文原创性声明和关于学位论文使用授权的声明

原 创 性 声 明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的科研成果。对本文的研究曾做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律责任由本人承担。

论文作者签名：_____ 日 期：__年 月__

关于学位论文使用授权的声明

本人完全了解贵州民族学院有关保留、使用学位论文的规定，同意学校保留或向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅；本人授权贵州民族学院可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或其他复制手段保存论文和汇编本学位论文。

（保密论文在解密后应遵守此规定）

论文作者签名：____导师签名：_____日 期：__年 月__