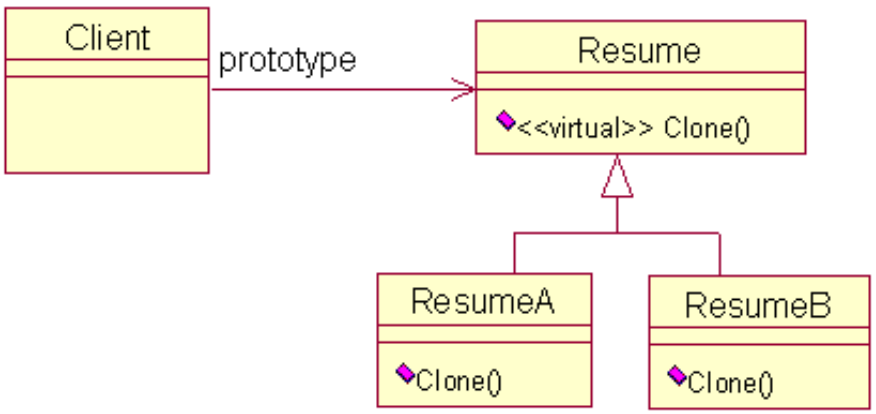
設計模式C++實現（５）——原型模式、模板方法模式

星期六, 2013 12月 14, 1:03 上午

軟件領域中的設計模式為開發人員提供了一種使用專家設計經驗的有效途徑。設計模式中運用了面向對象編程語言的重要特性：封裝、繼承、多態，真正領悟設計模式的精髓是可能一個漫長的過程，需要大量實踐經驗的積累。最近看設計模式的書，對於每個模式，用C++寫了個小例子，加深一下理解。主要參考《大話設計模式》和《設計模式:可復用面向對象軟件的基礎》（DP）兩本書。本文介紹原型模式和模板方法模式的實現。首先介紹原型模式，然後引出模板方法模式。

DP書上的定義為：用原型實例指定創建對象的種類，並且通過拷貝這些原型創建新的對象。其中有一個詞很重要，那就是拷貝。可以說，拷貝是原型模式的精髓所在。舉個現實中的例子來介紹原型模式。找工作的時候，我們需要準備簡歷。假設沒有打印設備，因此需手寫簡歷，這些簡歷的內容都是一樣的。這樣有個缺陷，如果要修改簡歷中的某項，那麼所有已寫好的簡歷都要修改，工作量很大。隨著科技的進步，出現了打印設備。我們只需手寫一份，然後利用打印設備複印多份即可。如果要修改簡歷中的某項，那麼修改原始的版本就可以了，然後再複印。原始的那份手寫稿相當於是一個原型，有了它，就可以通過複印（拷貝）創造出更多的新簡歷。這就是原型模式的基本思想。下面給出原型模式的UML圖，以剛才那個例子為實例。



原型模式實現的關鍵就是實現Clone函數，對於C++來說，其實就是拷貝構造函數，需實現深拷貝，下面給出一種實現。

```
1.  //父類
2.  class  Resume
3.  {
4.  protected :
5.      char  *name;
6.  public :
7.      Resume() {}
8.      virtual  ~Resume() {}
9.      virtual  Resume* Clone() {  return  NULL; }
10.     virtual  void  Set( char  *n) {}
11.     virtual  void  Show() {}
12. };
```

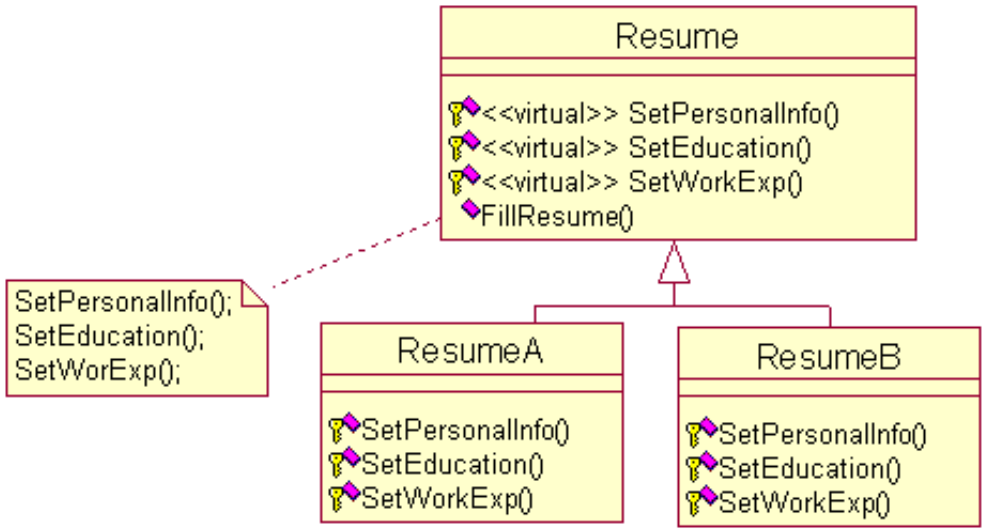
```
class ResumeA : public Resume
{
public :
    ResumeA( const char *str);  //構造函數
    ResumeA( const ResumeA &r); //拷貝構造函數
    ~ResumeA();                 //析構函數
    ResumeA* Clone();           //克隆，關鍵所在
    void Show();                //顯示內容
};
ResumeA::ResumeA( const char *str)
{
    if (str == NULL) {
        name = new char [1];
        name[0] = '\0';
    }
    else {
        name = new char [strlen(str)+1];
        strcpy(name, str);
    }
}
ResumeA::~ResumeA() { delete [] name;}
ResumeA::ResumeA( const ResumeA &r) {
    name = new char [strlen(r.name)+1];
    strcpy(name, r.name);
}
ResumeA* ResumeA::Clone() {
    return new ResumeA(* this );
}
void ResumeA::Show() {
    cout<< "ResumeA name : " <<name<<endl;
}
```

這裡只給出了ResumeA的實現，ResumeB的實現類似。使用的方式如下：

```
int main()
{
    Resume *r1 = new ResumeA( "A" );
    Resume *r2 = new ResumeB( "B" );
    Resume *r3 = r1->Clone();
    Resume *r4 = r2->Clone();
    r1->Show(); r2->Show();
    //刪除r1,r2
    delete r1; delete r2;
    r1 = r2 = NULL;
    //深拷貝所以對r3,r4無影響
    r3->Show(); r4->Show();
    delete r3; delete r4;
    r3 = r4 = NULL;
}
```

最近有個招聘會，可以帶上簡歷去應聘了。但是，其中有一家公司不接受簡歷，而是給應聘者發了一張簡歷表，上面有基本信息、教育背景、工作經歷等欄，讓應聘者按照要求填寫完整。每個人拿到這份表格後，就開始填寫。如果用程序實現這個過程，該如何做呢？一種方案就是用模板方法模式：定義一個操作中的算法的骨架，而將一些步驟延遲到子類中。模板方

法使得子類可以不改變一個算法的結構即可重定義該算法的某些特定步驟。我們的例子中，操作就是填寫簡歷這一過程，我們可以在父類中定義操作的算法骨架，而具體的實現由子類完成。下面給出它的UML圖。



其中FillResume() 定義了操作的骨架，依次調用子類實現的函數。相當於每個人填寫簡歷的實際過程。接著給出相應的C++代碼。

```
1. //簡歷
2. class  Resume
3. {
4. protected :  //保護成員
5.     virtual  void  SetPersonalInfo() {}
6.     virtual  void  SetEducation() {}
7.     virtual  void  SetWorkExp() {}
8. public :
9.     void  FillResume()
10.    {
11.        SetPersonalInfo();
12.        SetEducation();
13.        SetWorkExp();
14.    }
15. };
16. class  ResumeA:  public  Resume
17. {
18. protected :
19.    void  SetPersonalInfo() { cout<< "A's PersonalInfo" <<endl; }
20.    void  SetEducation() { cout<< "A's Education" <<endl; }
21.    void  SetWorkExp() { cout<< "A's Work Experience" <<endl; }
22. };
23. class  ResumeB:  public  Resume
24. {
25. protected :
26.    void  SetPersonalInfo() { cout<< "B's PersonalInfo" <<endl; }
27.    void  SetEducation() { cout<< "B's Education" <<endl; }
28.    void  SetWorkExp() { cout<< "B's Work Experience" <<endl; }
29. };
```

使用方式如下：

```
1.  int  main()
2.  {
3.     Resume *r1;
4.     r1 =  new  ResumeA();
5.     r1->FillResume();
6.     delete  r1;
7.     r1 =  new  ResumeB();
8.     r1->FillResume();
9.     delete  r1;
10.    r1 = NULL;
11.    return  0;
12. }
```