第一部分

附件8是“设计显示一个UI界面中包含的几何图形元素”的C++例子，其中提供了组合和继承两种设计，主调测试程序为TestUI.cpp。请分别分析这两种设计，如果需求改变，如给界面中增加圆形元素，请分别按组合和抽象设计方法增加这个功能。对比所需修改的部分，分析哪种设计修改量少、通用性高、更适用于软件复用？ 组合与继承哪种方法更能体现面向对象的核心思想：抽象？

> 客户需求变化：
> 如果客户要求给界面中增加圆形元素，请分别按组合和继承设计方法增加这个功能。对比所需修改的部分，分析哪种设计修改量少、通用性高、更适用于软件复用？如果未来再增加其他图形呢？

对于组合：

修改如下：

添加Circle.hpp 定义圆形元素

```
#ifndef __Circle__
#define __Circle__
#include "Point.h"
#include <iostream>
class Circle{
private :
    Point center;
    double radius;
public :
    Circle(const Point & A, float r) {
        center = A;
        radius = r;
    }
    void show() {
        std::cout << "Circle : (" << center.getX() << "," << center.getY()
<< " ) r = " << radius << std::endl;
    }
    Point getCenter() {
        return center;
    }
    double getRadius() {
        return radius;
    }
};
#endif
```

在testUI.cpp中添加以下代码，用来测试圆类型

```
    Circle circle(p1, 5);
    ui.circleVector.push_back(circle);
```

在UI.h中做以下修改，用来在UI中增加圆类型

```
// ------省略------
#include "Circle.hpp"
//类的组合
class UI{   //用户界面，组合类
    public:
    // ------省略------
    vector<Circle> circleVector;
    void show(){
        // ------省略------
        for (auto & i : circleVector) {
            i.show();
        }
    }
};
```

测试结果如下

对于继承

添加Circle.hpp，定义圆类型

```
#ifndef __Circle__
#define __Circle__
#include "Shape.h"
#include "Point.h"
#include <iostream>
class Circle : public Shape{
private :
    Point center;
    double radius;
public :
    Circle(const Point & A, float r) {
        center = A;
        radius = r;
    }
    virtual void show() {
        std::cout << "Circle : (" << center.getX() << "," << center.getY()
<< " ) r = " << radius << std::endl;
    }
    Point getCenter() {
        return center;
    }
    double getRadius() {
        return radius;
    }
};
#endif
```

在testUI.cpp做以下修改

```
//测试设计主调程序
// ------省略------
#include "Circle.hpp"
#include "UI.h"

main(){
    UI ui;
    // ------省略------
    ui.shapeVector.push_back(new Circle(p1, 5));
    ui.show();
}
```

测试结果：

详细代码可以在当前目录下inherit文件夹和composite文件夹下找到

问题解答

分析哪种设计修改量少、通用性高、更适用于软件复用？ 如果未来再增加其他图形呢？组合与继承哪种方法更能体现面向对象的核心思想：抽象？

第二种，继承的方式修改量少，通用性高. 组合的方式更适用于软件复用 . 如果未来在增加其他图形，继承也只需要添加一个文件继承Shape类，不需要修改UI. 相比之下组合要修改很多文件.

继承更能体现出抽象的思想，对所有的图形抽象出来Shape类

第二部分

实现图6-13中的 Engineer与Software Engineer的两种设计实例，并进行讨论。如果要利用组合来实现可替换性，需要怎样设计？

类继承：

SoftwareEngineer继承Engineer

```
class Engineer{
private :
    bool someThing;
public :

};

class SoftwareEngineer : public Engineer{
private :

public :

};
```

组合：

SoftwareEngineer有一个engCapabilities子类，该类指向Engineer;

```
class Engineer{
private :

public :

};

class engCapabilities {
private :
    Engineer *test;
public :
    engCapabilities(Engineer *x) {
        test = x;
    }
};

class SoftwareEngineer{
private :
    engCapabilities *ability;
public :
    SoftwareEngineer(Engineer *ability) {
        this->ability = new engCapabilities(ability);
    }
};
```

讨论：

设计中一个关键的决策就是如何最好地组织和关联复杂的对象。在面向对象的系统中，构造大型对象的技术主要有两种:继承和组合。也就是说，可以通过扩展和重载现有类的行为来创建新的类，或者通过组合简单的类来形成一个新类。如图6-13所示，左边的software Engineer类定义为Engineer类的子类，它继承了父类的工程能力，图的右边将software Engineer类定义为因含有构件Engineer对象而具有工程能力的组合类。我们注意到两种方法都可以复用和扩展设计。也就是说，在两种方法中，可复用的代码都可以作为独立的类(即父类或构件对象）来进行维护，新类(即子类或组合对象）通过引进新属性和新方法扩展了类的行为，且没有改变可复用的代码。除此之外，由于可复用的代码封装在独立的类中，我们可以安全地改变它的实现，从而间接更新了新类的行为。因此，在示例中不管使用的方法是继承还是组合，对Engineer类所做的改变都会自动地在software Engineer类中实现。

每一种构造范型都优劣并存。在保持被复用代码的封装性方面，组合的方法优于继承，因为组合的对象仅能通过它声明的接口来访问构件。在我们的示例中，Software Engineer对象只能使用它的构件方法来访问和更新自己的.工程能力。相比较之下，根据设计，子类可以直接访问它所继承的属性。组合的最大优点在于它允许动态替换对象构件。对象构件是组合对象的一个属性变量，和其他变量一-样，在程序执行的任何时候它的值都可以随时改变。此外，如果构件是根据一个接口来定义的，那么它可以用一个不同但类型兼容的对象替换。在组合而成的software Engineer类中,我们通过把engCapabilities重新指定到另一个对象，就可以改变它的.工程能力，包括方法实现。这种可变性本身也会导致问题，因为使用组合方法设计的系统在运行时可以被重新配置，所以我们很难仅通过研究代码，就能够想清楚或推理出程序的运行时结构。也并不总是能够搞清楚一个对象到底引用了哪些其他对象。组合的另外一个缺点就是，对象组合引入了一层间接性，一-个构件的方法的每一次访问都必须先访问这个构件对象，这种间接性可能会影响到程序运行时的性能。

相比较而言，如果使用继承的方法，子类的实现在设计的时候就已经确定了并且是静态的。与从组合类进行对象实例化相比，这种对象具有更小的灵活性，因为它们从父类继承的方法不可能在运行时发生改变。再者，由于从父类继承的特性对于子类而言往往是可见的，如果不是可直接访问理解和预测通过继

承方式构造的类将会相对容易些。当然，继承最大的好处就是，通过选择性地罚载被继承的定义，可以改变和特化继承方法的行为。这个特性可以帮助我们快速创建具有新行为的新的类型的对象。

利用组合来实现可替换性：

抽象一个Ability类，Engineer类继承Ability类 . engCapabilities指向Ability类型，传参只需要传Ability类型的指针即可，如果要替换则继承Ability类即可.

```
class Ability {

};
class Engineer : public Ability{

};

class engCapabilities {
private :
    Ability *test;
public :
    engCapabilities(Ability *x) {
        test = x;
    }
};

class SoftwareEngineer{
private :
    engCapabilities *ability;
public :
    SoftwareEngineer(Ability *ability) {
        this->ability = new engCapabilities(ability);
    }
};

int main() {
    Engineer x;
    SoftwareEngineer y(&x);
}
```