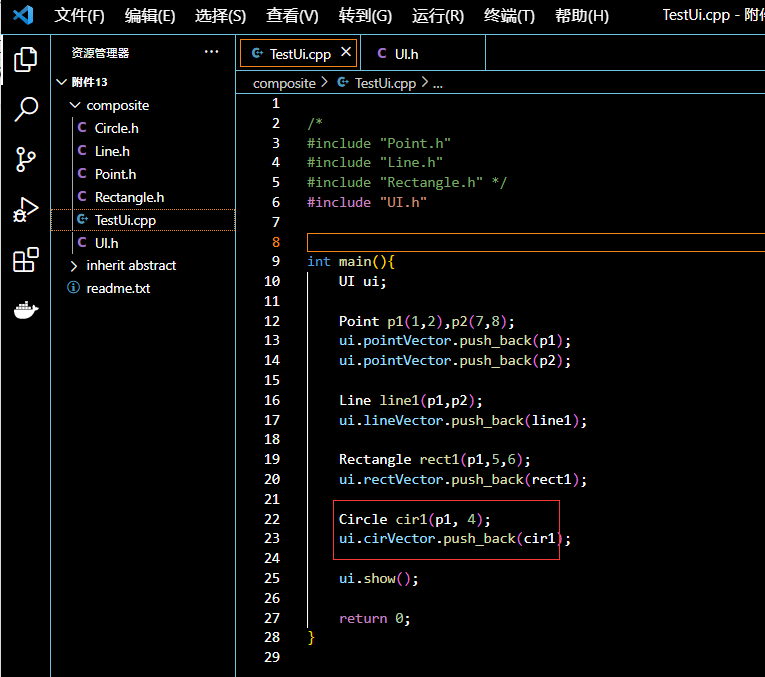
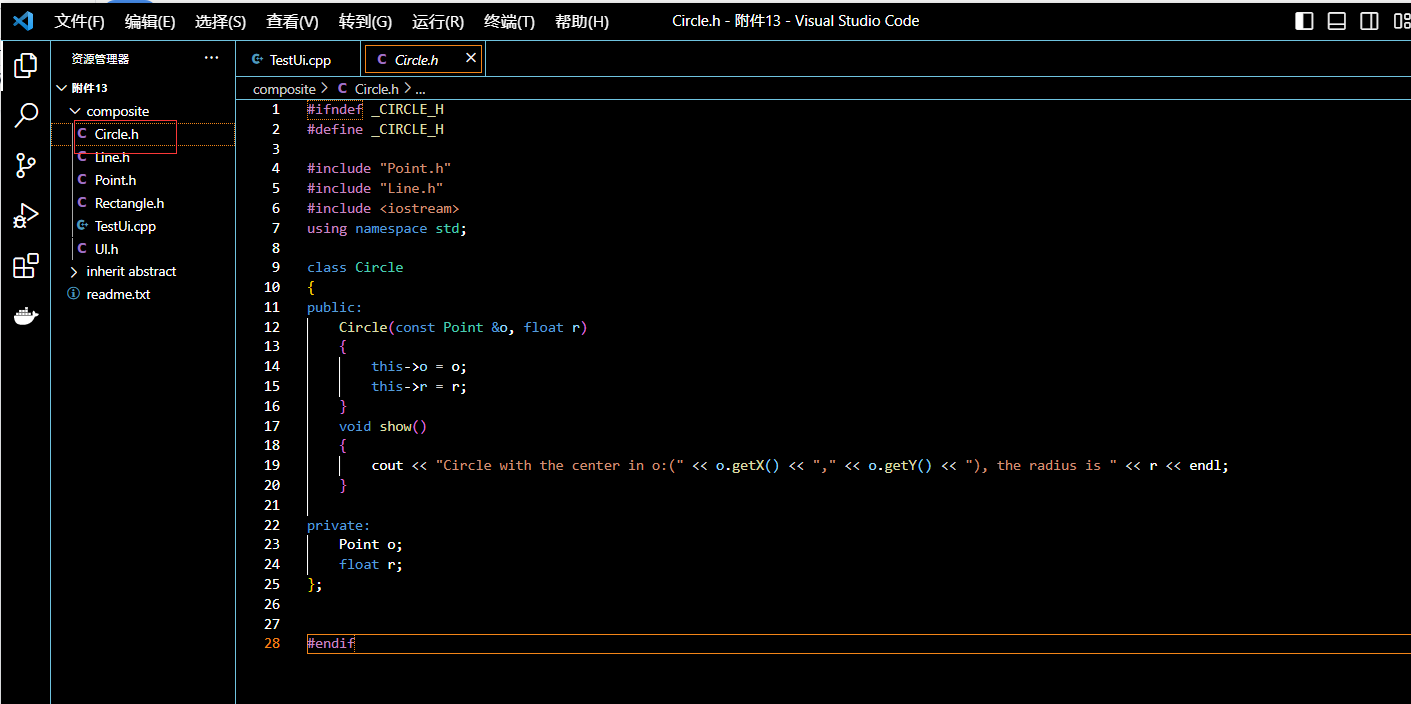
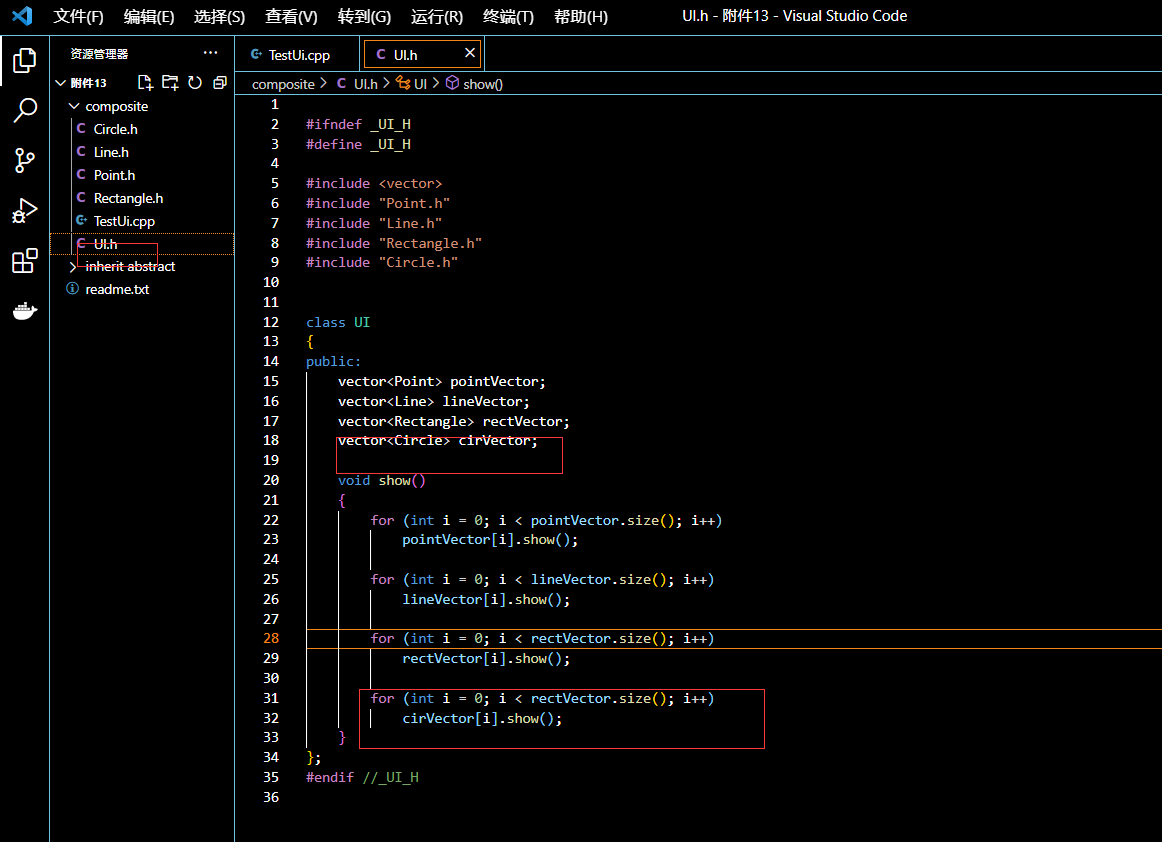
1. **附件13.zip是“设计显示一个UI界面中包含的几何图形元素”的C++例子，其中提供了两种设计，主调测试程序为TestUI.cpp。请分别分析评价这两种设计。如果需求改变，如给界面中增加圆形元素，请分别按原设计方法增加这个功能。对比所需修改的部分，分析哪种设计修改量少、通用性高、更适用于软件复用？**

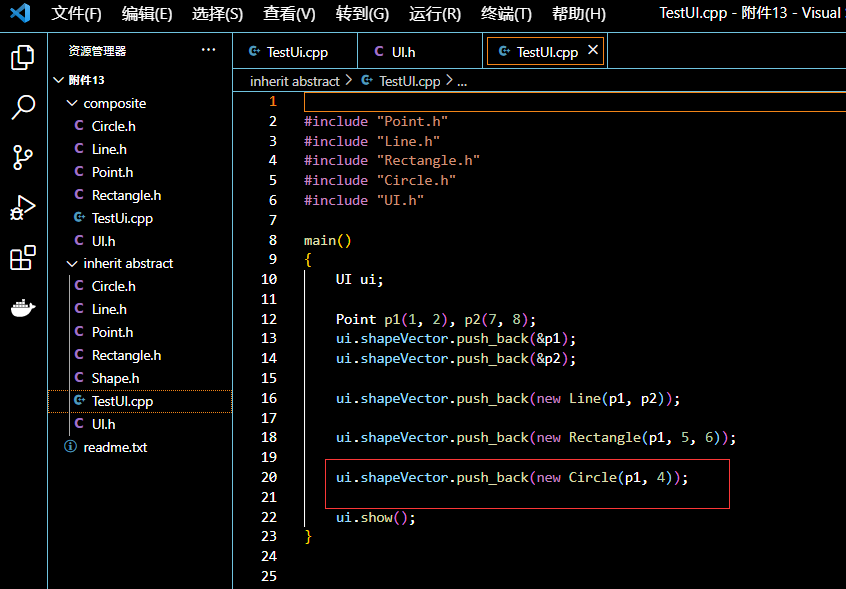
**Composite**

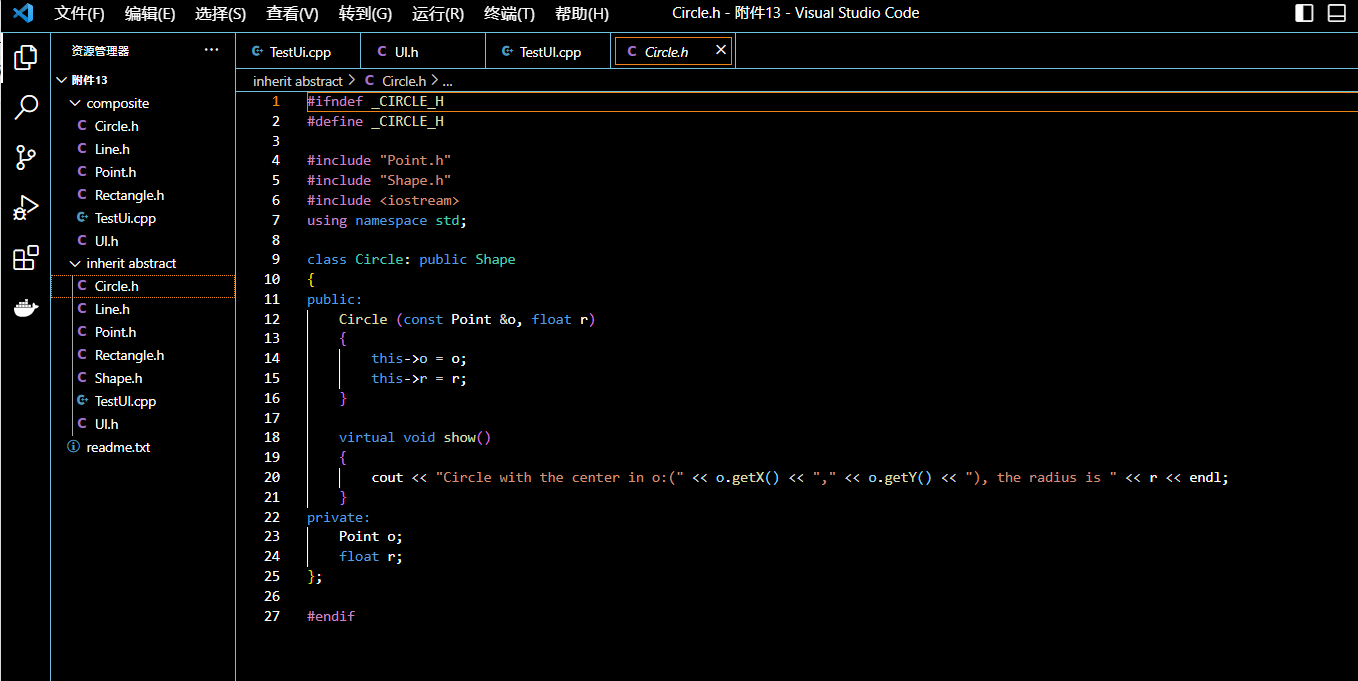






inherit abstract





可以明显看到composite中每个形状line，Point等的形状的show函数没有用虚函数，这样在ui.cpp中还得分别写出每个形状的show函数，而inherit abstract中每个形状的show函数使用了虚函数，这样在ui.cpp中就可以用一个show函数统一处理所有的形状。

虚拟类继承方式的代码修改量少，通用性高，更适用于软件复用。相对而言，普通方式的代码修改量大，通用性低，在大规模软件复用中效果不佳。经过分析和进行修改代码的实践inherit abstract修改量少、通用性高、更适用于软件复用。

**（2）实现教材图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？**

假设有三个特性（模块）， ModuleA, ModuleB, ModuleC。有两个类 ClassA 和 ClassB 需要使用它们。ClassA 会用到 ModuleA 和 ModuleB，ClassB 会用到全部三个特性。而且我们还假设现在全部A，B和C模块会只被ClassA和ClassB用到。

继承：

class ClassA {

functions of ModuleA;

functions of ModuleB;

};

class ClassB extends ClassA {

functions of ModuleC;

};

组合：

class ModuleA {};

class ModuleB {};

class ModuleC {};

class ClassA {

ModuleA ma;

ModuleB mb;

};

class ClassB {

ModuleA ma;

ModuleB mb;

ModuleC mc;

};

//或者：

class ClassB {

ClassA ca;

ModuleC mc;

};

两种方法都重用了模块。

对于组合，我们一般把模块叫做组件。在游戏开发中，这样的组合又叫做基于组件的实体系统。

组合比继承有哪些优势？

组合允许将问题分割成互不依赖的子元件。每个元件可以由不同的开发者开发。我们可以让三个程序员来同时开发 ModuleA， ModuleB 以及 ModuleC。

元件可以在运行时被替换，删除，或者添加（动态绑定）。与之相反，继承不能或者非常难（取决于编程语言）在运行时替换，删除，或者添加特性（静态绑定）。

组合比继承有更少的耦合。继承强制了 ClassB 和 ClassA 之间的耦合，但组合不会。

组件可以被用在其它的组合当中用以构成不同的行为。在继承中则没有办法只重用一个单个特性。继承强制了父类的所有特性被带入到了子类。

更少的代码冗余。继承比较容易导致冗余。B 继承自 A，通常 B 只需要 A 的部分特性，不需要的部分就成了冗余。如果 B 是由 A 的部分元件组合来的，那么 B 可以舍弃不需要的元件来避免冗余。

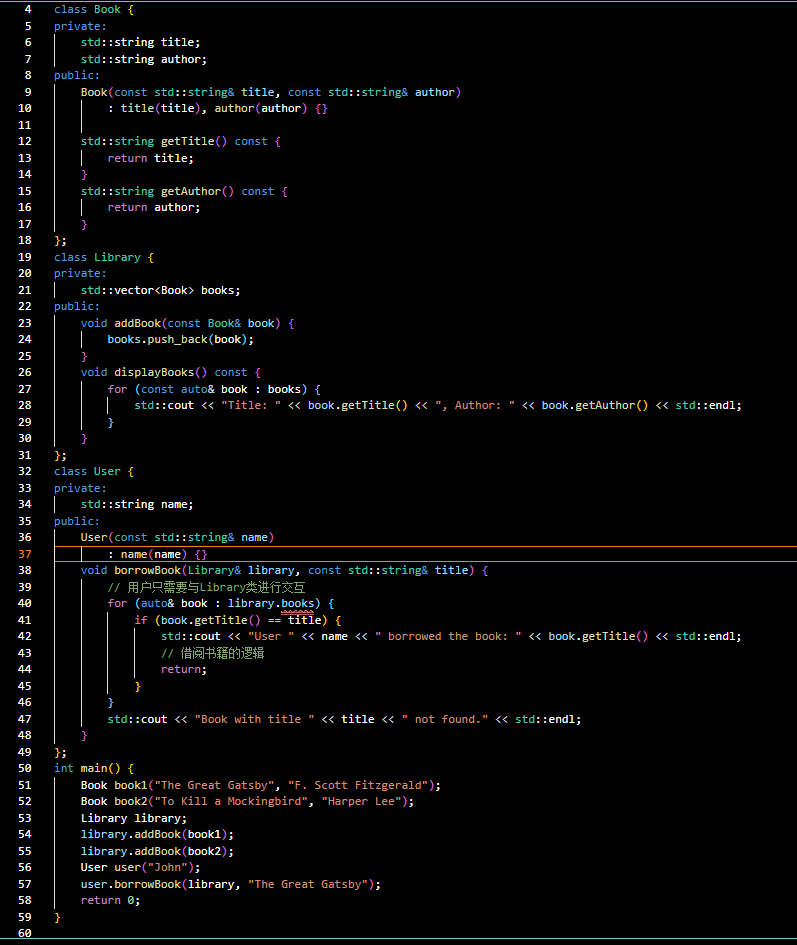
更好的封装。组合是基于公开接口的。每个元件不知道其它元件的内部细节。这正是封装所指的。而继承则把父类的内部细节（保护的接口）透露给了子类。更糟的是，子类通常会对父类做某些假设，而且父类也会假设子类会尊重它要求的接口。

容易更改。在组合中，任何元件都可以被更改而保证对其它元件影响很小，只要保持公开的接口。继承则强制了很紧密耦合的层次链。对于层次链任何节点的改变都将影响整个链。

小模块和原子模块。组合中，大的模块是由小的模块组成的，因此模块的层次可以得到控制。有可能我们只有两个层次，原子模块，和真实的功能模块，因 而层次非常平坦。在继承中，大的模块来自从小的模块继承。想要一个模块？继承吧，得到一个新的层级。这就可能造成有很多的层级，导致系统的模块层次非常复 杂且难以理解。

1. **参照课本P224图6-14，设计体现德米特法则的实例。分析该种设计的特点?**

在一个简单的图书管理系统中应用德米特法则C++如下:



Library 类负责管理图书，User 类表示用户，而 Book 类表示书籍。User 类只与 Library 类进行交互，通过 Library 类提供的接口来借阅书籍。这样，User 类不需要了解 Book 类的内部实现细节，也不需要直接与 Book 类进行交互。

这个设计示例的特点与前面所述的特点相似，包括降低了类之间的直接依赖关系，提高了类的独立性和灵活性。