**软件工程实验小组——写bug组**

**团队作业**

**一．小组成员介绍**

成员：阎发祥 202222130195

黄茁 202200130264

迟骋 202200130146

徐林 202200130168

刘浩南 202200130194

**本次实验报告由团队成员5人一起完成，第三部分Enginee设计实例由徐林完成，第四部分Software Engineer设计实例由黄茁完成，要利用组合来实现可替换性设计由迟骋完成，第五部分设计实现依赖倒置原则进行模块解耦的实例由刘浩南、阎发祥完成，个人作业由每个人自行完成。**

**二．本周工作内容简介**

1. 练习应用设计原则的进行设计

**三．实现课本P221图6-13中的 Enginee设计实例【徐林完成】**

# include <stdio.h>

# include <string.h>

// 定义Engineer结构体

typedef struct {

    char name[100];

} Engineer;

// Engineer的工作函数

void Engineer\_work(Engineer \*e) {

    printf("%s is working as an engineer.\n", e->name);

}

// 定义SoftwareEngineer结构体

typedef struct {

    Engineer base;

    char capabilities[100];

} SoftwareEngineer;

// SoftwareEngineer的编码函数

void SoftwareEngineer\_code(SoftwareEngineer \*se) {

    printf("%s is coding with %s capabilities.\n", se->base.name, se->capabilities);

}

int main() {

    // 创建一个Engineer实例

    Engineer e;

    strcpy(e.name, "John Doe");

    Engineer\_work(&e);

    // 创建一个SoftwareEngineer实例

    SoftwareEngineer se;

    strcpy(se.base.name, "Jane Smith");

    strcpy(se.capabilities, "C, C++, Python");

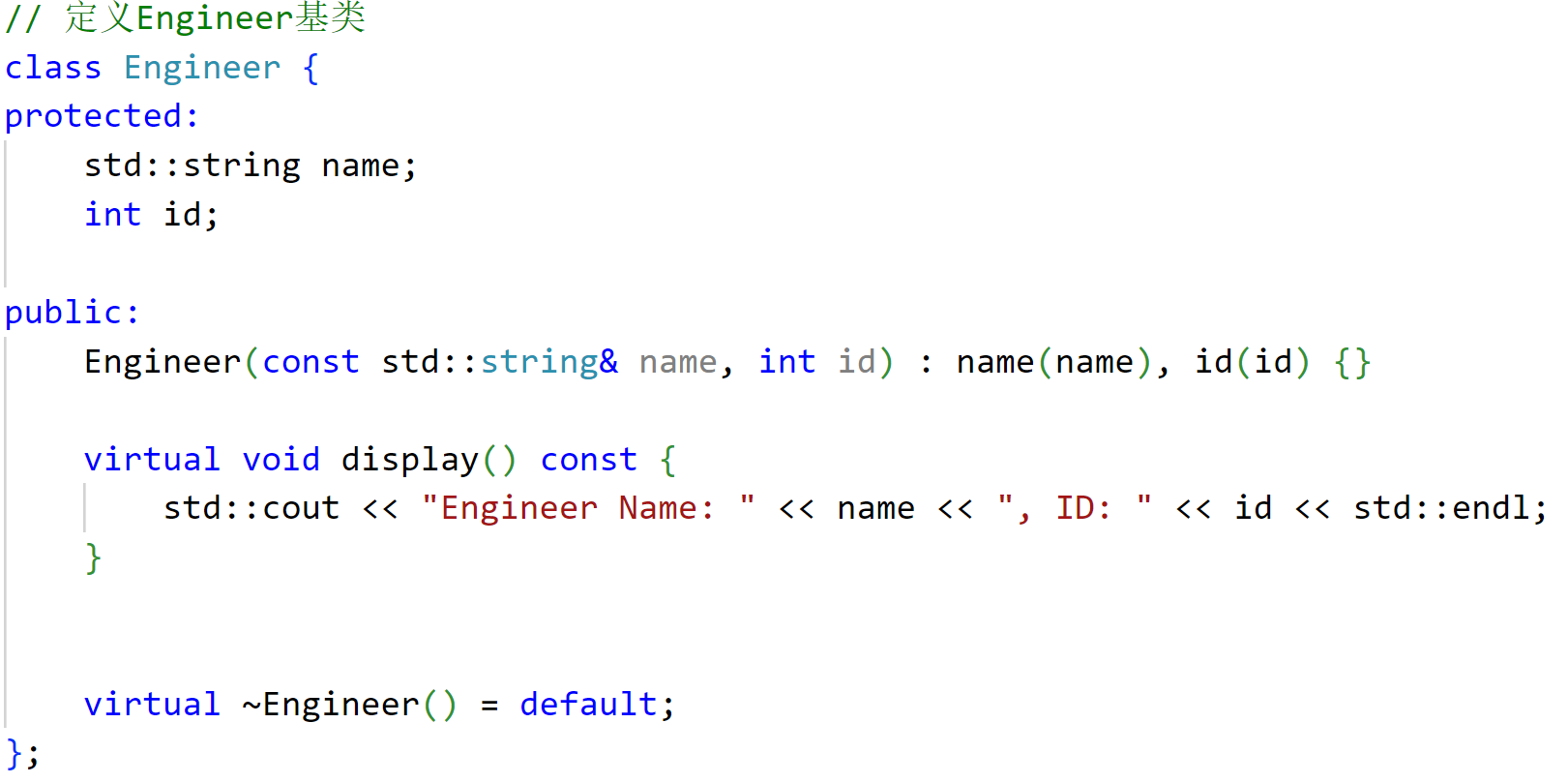
    Engineer\_work((Engineer\*)&se);

    SoftwareEngineer\_code(&se);

    return 0;

}

**四． 实现课本P221图6-13中的 Software Engineer设计实例【黄茁完成】**

****

****

// 定义Engineer基类

class Engineer {

protected:

    std::string name;

    int id;

public:

    Engineer(const std::string& name, int id) : name(name), id(id) {}

    virtual void display() const {

        std::cout << "Engineer Name: " << name << ", ID: " << id << std::endl;

    }

    virtual ~Engineer() = default;

};

// SoftwareEngineer派生类

class SoftwareEngineer : public Engineer {

private:

    std::string engCapabilities;

public:

    SoftwareEngineer(const std::string& name, int id, const std::string& capabilities)

        : Engineer(name, id), engCapabilities(capabilities) {}

    void display() const override {

        Engineer::display(); // 调用基类的display方法

        std::cout << "Engineering Capabilities: " << engCapabilities << std::endl;

    }

    Engineer\* engineerIdentity;

    void setEngineerIdentity(Engineer\* identity) {

        engineerIdentity = identity;

    }

};

**Q：如果要利用组合来实现可替换性，需要怎样设计？【迟骋完成】**

定义BaseEngineer类，包含所有工程师都有的方法接口：

class BaseEngineer {

public:

virtual void display() const = 0;

virtual ~BaseEngineer() = default;

};

定义Engineer类，继承自BaseEngineer类并实现接口：

class Engineer : public BaseEngineer {

protected:

std::string name;

int id;

public:

Engineer(const std::string& name, int id)

: name(name), id(id) {}

void display() const override {

std::cout << "Engineer Name: " << name << ", ID: " << id << std::endl;

}

};

定义SoftwareEngineer类，不继承自Engineer，而是通过组合一个BaseEngineer 对象来实现其功能，实现相同的接口：

class SoftwareEngineer : public BaseEngineer {

private:

std::unique\_ptr<BaseEngineer> baseEngineer;

std::string capabilities;

public:

SoftwareEngineer(std::unique\_ptr<BaseEngineer> engineer, const std::string& capabilities)

: baseEngineer(std::move(engineer)), capabilities(capabilities) {}

void display() const override {

baseEngineer->display();

std::cout << "Capabilities: " << capabilities << std::endl;

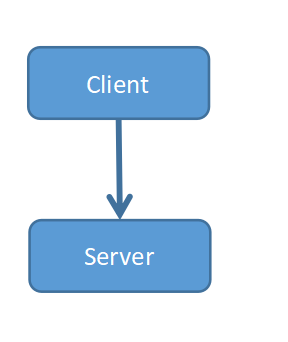
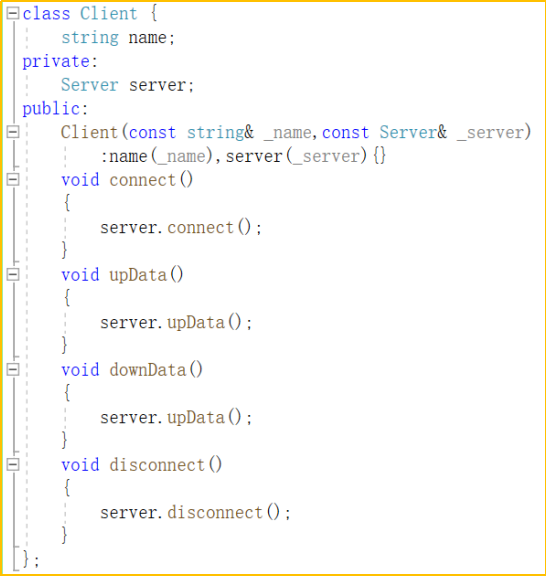
}

};

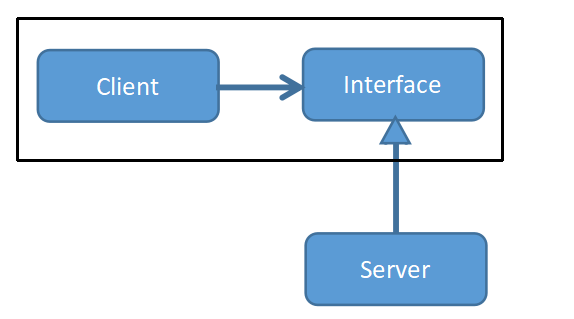
Engineer和SoftwareEngineer都实现了BaseEngineer接口，因此可以将它们统一作为BaseEngineer指针使用，利用组合实现了可替换性。

**五．参照课本P225图6-15，设计实现依赖倒置原则进行模块解耦的实例。【刘浩南、阎发祥完成】**

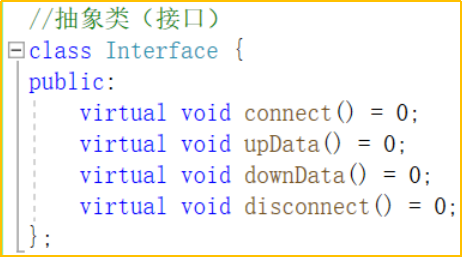
Client类依赖于Server类

****

依赖倒置



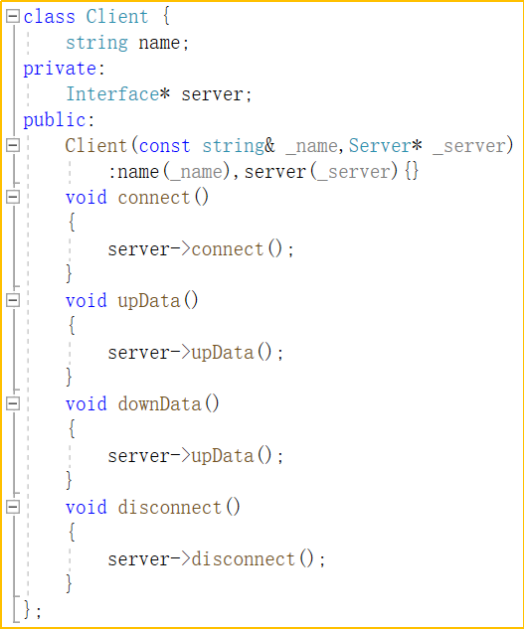
定义server抽象类（接口），在抽象类中定义用到的所有方法



Server类继承自抽象类



Client类依赖于抽象类



**个人作业**

**题目：附件8.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。**

**（a）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？**

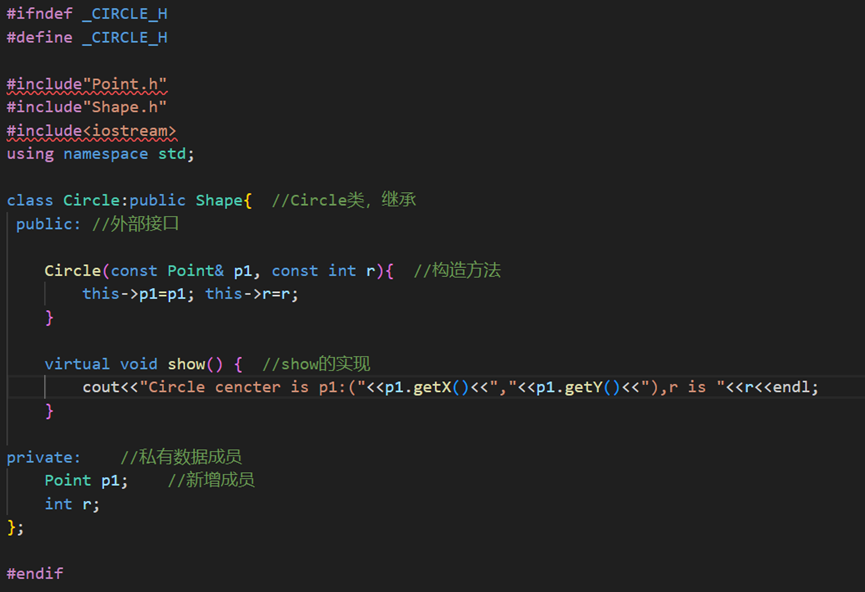
**（b) 讨论例子中所符合的模块设计的六项设计原则**

1. **阎发祥 202222130195**

第一种设计模式是基于抽象类的设计。所有的几何元素都对 shape 抽象类进行了实现。每一个具体的几何元素，除了拥有自身特有的方法和属性之外，还必须实现抽象类中的纯虚函数 show。在 UI 界面的设计中，只需要一个用于存储 shape \* 类型的数组，就能够存放各种不同的几何元素。当需要显示 UI 界面时，只需对这个数组进行一次遍历操作，就可以将所有的几何元素展示出来。如果后续要增加新的几何元素种类，只需对 shape 抽象类进行新的实现即可，无需额外创建新的数组来存储。

第二种设计模式则不依赖于 shape 抽象类，所有的几何元素都独立进行设计。由于缺乏统一的抽象类基础，在 UI 界面中，就需要使用不同的数组来分别存储不同类型的几何元素，无法像第一种方式那样将所有几何元素存储在同一个数组中。因此，在显示这些几何元素时，就需要分别对不同的数组进行遍历操作。而且，每当要添加新的几何元素种类时，就必须创建一个新的数组来专门存储该几何元素。

（a）对于抽象类实现下，增加圆形，只需要新增一个Circle类实现shape即可，无需修改UI。这样不存在大量的代码修改，可见这种实现具有较高的通用性和代码复用性。



而对于非抽象类的实现，要想增加圆形，除了新增一个Circle类，还要在UI下新增一个数组用于存储Circle对象，需要进行大量的代码修改，通用性和复用性较差。

****

（b）这段程序是一个主调测试程序，用于测试 UI 界面中几何图形元素的显示功能。它包含了对 Point.h、Line.h、Rectangle.h 和 UI.h 头文件的引用。在 main 函数里，创建了 UI 对象，接着分别创建了 Point、Line 和 Rectangle 类型的对象，并将它们添加到 UI 对象的相应向量中，最后调用 UI 对象的 show 方法显示这些图形元素。

从模块设计原则来看，该程序在一定程度上遵循了单一职责原则，各图形类职责明确；也符合开闭原则，便于添加新图形；同时满足迪米特法则，UI 类与各图形类交互简单。不过在里氏替换原则、依赖倒置原则和接口隔离原则方面，由于代码中未体现明显的继承、抽象和接口等关系，需要进一步设计扩展。

1. **黄茁**

a)

抽象类的实现代码更少，只需新增一个类，无需修改现有代码。通用性高，通过多态机制，可以统一处理各种形状。更适用于软件复用，新的形状类可以独立开发和测试，不影响其他部分

而普通类则需要创建一个新的Circle.h文件，定义Circle类。修改UI类，使其能够处理Circle对象。需要在UI类中添加特定的方法或条件判断来处理圆形。修改量大，需要修改UI类以适应新的形状。通用性低，每次增加新形状都需要修改UI类，耦合度高。不利于软件复用，新的形状类不能独立于UI类存在，增加了维护难度。

b).

例子中的抽象类设计方法符合以下六项设计原则：

**单一职责原则（SRP）：**

Shape抽象类及其子类只负责图形的显示逻辑，UI类只负责管理图形对象和调用显示方法，职责明确。

**开放封闭原则（OCP）：**

系统对扩展开放，对修改封闭。增加新的形状类（如Circle）时，不需要修改现有的UI类和其他形状类。

**里氏替换原则（LSP）：**

子类（如Rectangle, Line, Circle）可以替换其基类（Shape）出现的任何地方，并且不会影响程序的正确性。

**依赖倒置原则（DIP）：**

高层模块（UI类）不依赖于低层模块（具体形状类），而是依赖于抽象（Shape类）。具体形状类依赖于抽象，而不是相互依赖。

**接口隔离原则（ISP）：**

Shape抽象类提供了一个最小且完整的接口，每个子类只需要实现这个接口即可，避免了不必要的方法。

**迪米特法则（LoD）：**

UI类只与直接的朋友（Shape类及其子类）交互，不与朋友的朋友交互，降低了系统的耦合度

1. **徐林202200130168**

**“normal”设计： 使用了多个向量来分别存储不同类型的图形对象。这种设计在增加新的图形类型时需要修改 UI 类，以添加新的向量和管理新的图形类型。**

**“abstract”设计： 使用了单一向量来存储所有图形对象的指针。这种设计利用了多态性，在增加新的图形类型时不需要修改 UI 类，只需确保新的图形类继承自一个共同的基类并实现相应的虚函数。**

**（a）增加圆形元素：**

**“normal”设计： 需要为圆形添加一个新的向量，并在 UI 类中添加处理圆形的逻辑。**

**“abstract”设计： 只需添加一个圆形类，继承自共同的基类，并实现相应的虚函数，无需修改 UI 类。**

**对比分析：**

**修改量： “abstract”设计的修改量更少，因为不需要更改现有的 UI 类。**

**通用性： “abstract”设计具有更高的通用性，因为它可以很容易地扩展以包含新的图形类型。**

**软件复用： “abstract”设计更适用于软件复用，因为它的扩展不需要修改现有类结构，符合开闭原则。**

**（b）模块设计的六项设计原则：**

**单一职责原则：每个类都有明确的职责。**

**开放封闭原则：类对扩展开放，对修改封闭。**

**里氏替换原则：子类可以替换父类且不影响程序的正确性。**

**接口隔离原则：客户端不应该依赖它不需要的接口。**

**依赖倒置原则：高层模块不应该依赖低层模块，两者都应该依赖抽象。**

**迪米特法则：一个对象应该对其他对象有最少的了解。**

1. **刘浩南 202200130194**

普通类则需要创建一个新的Circle.h文件，定义Circle类。修改UI类，使其能够处理Circle对象。需要在UI类中添加特定的方法或条件判断来处理圆形。修改量大，需要修改UI类以适应新的形状。通用性低，每次增加新形状都需要修改UI类，耦合度高。不利于软件复用，新的形状类不能独立于UI类存在，增加了维护难度。

而抽象类的实现代码更少，只需新增一个类，无需修改现有代码。通用性高，通过多态机制，可以统一处理各种形状。更适用于软件复用，新的形状类可以独立开发和测试，不影响其他部分。

（a）对于抽象类实现下，增加圆形，只需要新增一个Circle类实现shape即可，无需修改UI。这样不存在大量的代码修改，可见这种实现具有较高的通用性和代码复用性。而对于非抽象类的实现，要想增加圆形，除了新增一个Circle类，还要在UI下新增一个数组用于存储Circle对象，需要进行大量的代码修改，通用性和复用性较差。

（b）这段程序是一个主调测试程序，用于测试 UI 界面中几何图形元素的显示功能。它包含了对 Point.h、Line.h、Rectangle.h 和 UI.h 头文件的引用。在 main 函数里，创建了 UI 对象，接着分别创建了 Point、Line 和 Rectangle 类型的对象，并将它们添加到 UI 对象的相应向量中，最后调用 UI 对象的 show 方法显示这些图形元素。

从模块设计原则来看，该程序在一定程度上遵循了单一职责原则，各图形类职责明确；也符合开闭原则，便于添加新图形；同时满足迪米特法则，UI 类与各图形类交互简单。不过在里氏替换原则、依赖倒置原则和接口隔离原则方面，由于代码中未体现明显的继承、抽象和接口等关系，需要进一步设计扩展。

**5.迟骋 202200130146**

面向对象继承架构方案采用基类封装机制构建图形系统，所有具体图形类（如圆形、矩形、多边形等）均继承自统一的Shape基类。Shape基类定义了display()图形渲染等逻辑的核心接口规范。在界面渲染模块中，通过建立Shape指针容器实现多态存储，可同时容纳各类图形对象实例。渲染流程中仅需对该容器进行单次迭代遍历，即可完成全部图形对象的绘制操作。系统扩展时，新增图形类型只需继承基类并实现接口方法，无需修改现有存储结构即可自动兼容。

独立组件架构方案采用扁平化设计思路，各图形类作为独立组件开发，缺乏统一的接口规范，界面层需要为每种图形类型独立维护，渲染时需针对不同容器分别处理，导致显示逻辑分散；增加新图形类型时必须创建专用存储容器并重构渲染流程，代码复杂度过高，代码可读性不佳，维护难度较高，并且导致代码膨胀，不适合复杂的工程，在后续维护中增量更新困难。

（a）新增圆形元素的抽象类设计通过继承Shape基类并重写draw()虚函数完成扩展，无需修改现有代码，仅需新增Circle类，继承基类并利用多态机制在TestUI.cpp中统一调度

普通类设计方案需要创建Circle类并修改UI容器类的接口与实现，添加类型判断逻辑或专用处理方法，修改涉及到的模块更多，且会涉及到其他模块。

从扩展效率看，抽象类方案代码修改范围更低，可扩展性强，有利于后续维护和增量更新；普通类方案缺乏扩展性，对一个模块的修改容易引发一系列大量的修改，影响到多个模块，维护成本更高，代码可读性较差，对于复杂项目出错的概率也更高。

（b）

1. 单一职责原则（SRP）

一个类或模块只承担一个职责，避免将多个不相关的功能耦合在一起，可以提高代码可读性、可维护性和复用性，减少因需求变更带来的影响范围，提升系统的稳定性。

2. 里氏替换原则（LSP）

子类应该能够替换父类而不破坏程序的行为逻辑，是对继承使用方式的规范，确保在使用继承扩展功能时，不会因为子类对父类方法的重写而导致不可预期的错误，保障程序的正确性和一致性。

3. 依赖倒置原则（DIP）

高层模块不依赖低层模块，而是依赖于抽象；抽象不依赖具体实现，具体实现依赖抽象，通过面向接口编程，降低模块之间的耦合度，提高系统灵活性和可扩展性。

4. 接口隔离原则（ISP）

使用多个权责分明、内部相关性强的小接口，不使用大接口，使客户端只依赖于必要的方法，减少不必要的耦合，增强系统的可维护性和灵活性。

5. 开闭原则（OCP）

类、模块、函数等对扩展开放，对修改关闭，在增加功能时尽量通过新增代码而非修改已有代码来实现，保证原有功能的稳定，避免在原有模块中引入错误。

6. 迪米特法则（LoD）

减少对象与其他对象的交互，只相互开放必要的信息，可以降低类与类之间的耦合度，提高模块的独立性，使系统更易于维护和扩展，提高系统的安全性。