程序设计方法
课后作业 #2

作业提交时间:2019-04-08 上课前

注意事项: 本作业的解题方法有很多种，本答案只是给出一个参考，但不是唯一的解题方法。

[1] (25 pts) 阅读下列程序，回答以下几个问题;

```
class Contained {
    public void disp() {
        System.out.println("disp() of Contained Object");
    }
}
public class Container
{
    private Contained c;
    //以下为结构 A
    Container(){
         c = new Contained();
      }
     //以下为结构 B
     public Contained getC(){
        return c;
    }
    public void setC(Contained c) {
        this.c = c;
    }
    public static void main(String[] args) {
        Container container = new Container();
```

```
        Contained contained = new Contained();
        container.setC(contained);
    }
}
```

(1).请解释程序段的设计使用了程序设计的那些设计原则

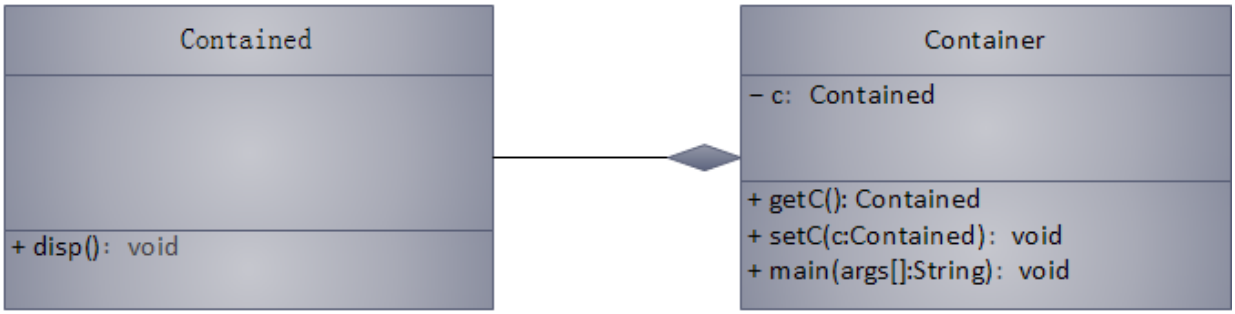
1.开放封闭原则：

Container 类通过合成 Contained 类，对 Contained 类进行修改。

2.合成-聚合复用原则：

Container 类拥有 Contained 类。

(2).请画出本段程序的 **UML** 类图



(3).请将本程序修改为使用继承来实现复用

```
class Contained  {
    public void disp() {
        System.out.println("disp() of Contained Object");
    }
}
public class Container extends Contained {
    private Contained c;
```

```
    public Contained getC() {
        return c;
    }
    public void setC(Contained c) {
        this.c = c;
    }

    public static void main(String[] args) {
        Container container = new Container();
        container. disp();
    }
}
```

(4).请解释面向对象程序设计中继承 **(Inheritance)**的优点与缺点

1. 优点：

（1）新的实现较为容易，因为超类的大部分功能可以通过继承关系自动进入子类。

（2）修改或扩展继承而来的实现较为容易。

2. 缺点：

（1）继承破坏包装，因为其将父类的实现细节暴露给子类，父类的内部细节常常是对子类透明的，因此这种复用是透明的复用，又称白盒复用。

（2）如果超类的实现发生改变，那么子类的实现不得不发生改变。

（3）从超类继承而来的实现是静态的，不可能在运行时间内发生改变，缺乏足够的灵活性。

[2] (25 pts) 以下程序是一个 Shell 解释器的伪代码，请指出以下程序违反了什么设计原则，并指出如何改正这个程序:

```
public void Parse()
{
    StringReader reader = new StringReader(scriptTextToProcess);
    StringBuilder scope = new StringBuilder();
    string line = reader.ReadLine();
    scope = new StringBuilder();

    while (line != null)
    {
        switch (line[0])
        {
            case '$':
            // Process the entire "line" as a variable,
             // i.e. add it to a collection of KeyValuePair.
                line.AddToVariables();
                break;
            case '!':
             // Depending of what comes after the '!' character,
             // process the entire "scope" or command in "line".
                if (line == "!execute")
                    scope.ExecuteScope();
                else if (line.StartsWith("!custom_command"))
                    line.RunCustomCommand(scope);
                else if (line == "!single_line_directive")
                    line.ProcessDirective();
                 break;
             default:
                // No processing directive, i.e. add the "line"
                // to the current scope.
```

```
                scope.Append(line);
                break;
            }
            line = reader.ReadLine();
        }
}
```

答：违反了单一职责原则，在 switch 语句中，对不同的情况实施了不同的操作，使得本方法承担了个不同的职责，所以违反了单一职责原则。

改正代码：

```
public void Parse_$ ()  {
    StringReader reader = new StringReader(scriptTextToProcess);
    StringBuilder scope = new StringBuilder();
   string line = reader.ReadLine();
   scope = new StringBuilder();


 while (line != null) {
     if(line[0] ==  '$')
        line.AddToVariables();
      line = reader.ReadLine();
   }
}

public void Parse_excalmatory_mark ()  {
    StringReader reader = new StringReader(scriptTextToProcess);
    StringBuilder scope = new StringBuilder();
        string line = reader.ReadLine();
        scope = new StringBuilder();
```

```
        while (line != null) {
        if(line[0] ==  '!') {
                if (line == "!execute")
                    scope.ExecuteScope();
        else if (line.StartsWith("!custom_command"))
          line.RunCustomCommand(scope);
                else if (line == "!single_line_directive")
            line.ProcessDirective();
          line = reader.ReadLine();
              }
          }

public void Parse_null () {
    StringReader reader = new StringReader(scriptTextToProcess);
    StringBuilder scope = new StringBuilder();
        string line = reader.ReadLine();
        scope = new StringBuilder();

        while (line != null) {
      if((line[0] !=  '$') $$ (line[0] !=  '!')
                scope.ExecuteScope();
       line = reader.ReadLine();
     }
}
```

[3] **Factory** 模式

请用 **Java** 语言实现一个计算器控制台程序，要求输入两个数和以下五种运算符号+ −× ÷ √， 求解并显示得到的结果，注意请使用简单工厂模式来实现这些功能，画出 **UML** 图并附上你开发的代码。**(25pts)**

♉存在多种答案，请将代码运行并执行以验证正确性，不提供标准答案

[4]在 **Java** 中 **collections** 接口的 **iterator** 方法是一个 **Factory** 模式的设计方法，请解释为什么，并举例。**(25pts)**（不少于 200 个汉字，举例请写 **Java** 程序或者 **C++**程序，否则本题得零分）

解答：

工厂方法模式是对简单工厂模式进行了抽象。有一个抽象的 **Factory** 类（可以是抽象类和接口），这个类将不在负责具体的产品生产，而是只制定一些规范，具体的生产工作由其子类去完成。在这个模式中，工厂类和产品类往往可以依次对应。即一个抽象工厂对应一个抽象产品，一个具体工厂对应一个具体产品，这个具体的工厂就负责生产对应的产品。

Iterator 迭代器是一种方法可以访问一个容器对象中各个元素而又不需要暴露该对象内部细节。**Java Collection** 是由客户程序来控制遍历的过程，具体的 **iterator** 角色是定义在容器角色中的内部类，这样便保护了容器的封装，但是同时容器也提供了遍历算法接口。而 **Factory** 模式就是抽象和封装了实例化的动作的工厂，提供尽可能统一的接口给用户，简化对象的实例化过程。在 **Java** 中 **Collections** 的接口的 **iterator** 方法是 **Factory** 模式的设计方法，用户可以忽略对象具体实例化的逻辑过程，通过调用同意的接口来进行实例化，使得使用者仅依赖于抽象。

举例：请自行举例，不提供标准答案