# 实验四：组合模式

## 实验内容

被测试的类：

**package** cn.edu.scau.sec.**\*\*\***.composite;

**public** **class** FourMixedOperations {

**public** **int** add(**int** a, **int** b) {

**return** a+b;

}

}

测试用例(TestCase)有原子测试用例(AtomTestCase)和测试用例套件(TestSuite)两种类型；测试用例套件可以包含零到多个测试用例(零到多个原子测试用例和零到多个测试用例套件)；原子测试用例套件不能包含测试用例。

原子测试用例(AtomTestCase)和测试用例套件(TestSuite)两种类型都可以执行测试；原子测试用例执行测试是用于比较期望值和**实际值**是否相等(public boolean test())，如果期望值和实际值相等，返回true, 否则返回false；组合测试用例执行测试是递归执行组合测试用例中包含的所有原子测试用例。

分别使用安全型组合模式和一致型组合模式完成本实验。

客户端基本测试验证：输入正确的和错误的多种测试用例以及测试用例组合，输出测试结果，例如

1 创建第一个原子测试用例，期望的值是2，两个数分别是1，1；控制台输出的结果是“测试通过”；

1. 创建第二个原子测试用例，期望的值是3，两个数分别是2，1；控制台输出的结果是“测试通过”；
2. 创建第三个原子测试用例，期望的值是4，两个数分别是2，1；控制台输出的结果是“测试没有通过，请检查测试数据和被测试的方法，查找测试没有通过的原因。”；
3. 创建第一个组合测试用例，添加第一个原子测试用例，运行这个测试用例的测试方法，运行完毕后输出“第一个组合测试添加了第一个原子测试用例后测试完成”；
4. 在第一个组合测试用例中添加第二个原子测试用例并运行这个组合测试用例，运行完毕后输出“第一个组合测试添加了第一个和第二个原子测试用例后的组合测试用例测试完成”；
5. 创建第二个组合测试用例，添加第三个原子测试用例，运行这个测试用例的测试方法，运行完毕后输出“第二个组合测试用例测试完成”；
6. 在第一个组合测试用例中添加第二个组合测试用例，并运行第一个组合测试用例，运行完毕后输出“第一个组合测试用例添加了第二个组合测试用例后，第一个组合测试用例运行完毕”；
7. 在第7步的第一个组合测试用例中删除第二个组合测试用例，并运行第一个组合测试用例，运行完毕后输出“第一个组合测试用例删除了第二个组合测试用例后，第一个组合测试用例运行完毕”；

## 安全型组合模式

### UML设计



### 核心代码

TestCase.java

public abstract class TestCase {

public abstract boolean judgeResult();

}

AtomTestCase.java

public class AtomTestCase extends TestCase {

public String name;

public int a;

public int b;

public int expectedValue;

public AtomTestCase(String name, int a, int b, int expectedValue) {

this.name = name;

this.a = a;

this.b = b;

this.expectedValue = expectedValue;

}

@Override

public boolean judgeResult(){

FourMixedOperation op = new FourMixedOperation();

if (op.add(a,b)==expectedValue){

System.out.print(a + " " + b + " " + expectedValue + " ");

System.out.print(this.name);

System.out.println(" 测试通过！");

return true;

}else {

System.out.print(a + " " + b + " " + expectedValue + " ");

System.out.print(this.name);

System.out.println(" 测试没有通过!");

return false;

}

}

}

TestSuite.java

public class TestSuite extends TestCase {

public String name;

private List<TestCase> components = new ArrayList<>();

public TestSuite(String name){

this.name = name;

}

@Override

public boolean judgeResult() {

for (TestCase item : components){

if (item instanceof AtomTestCase){

if (item.judgeResult()){

continue;

}

}else {

item.judgeResult();

}

}

return true;

}

}

FourMixesOperation.java

public class FourMixedOperation {

public int add(int a,int b){

return a + b;

}

}

### 实现效果



## 一致型组合模式

### UML设计



### 核心代码

TestCase.java

public abstract class TestCase {

public abstract void remove(TestCase f) throws Exception;

public abstract void addEle(TestCase f) throws Exception;

public abstract boolean judgeResult();

}

AtomTestCase.java

public class AtomTestCase extends TestCase {

public String name;

public int a;

public int b;

public int expectedValue;

public AtomTestCase(String name, int a, int b, int expectedValue) {

this.name = name;

this.a = a;

this.b = b;

this.expectedValue = expectedValue;

}

@Override

public void remove(TestCase f)throws Exception{

throw new Exception();

}

@Override

public void addEle(TestCase f) throws Exception {

throw new Exception();

}

@Override

public boolean judgeResult(){

FourMixedOperation op = new FourMixedOperation();

if (op.add(a,b)==expectedValue){

System.out.print(a + " " + b + " " + expectedValue + " ");

System.out.print(this.name);

System.out.println(" 测试通过！");

return true;

}else {

System.out.print(a + " " + b + " " + expectedValue + " ");

System.out.print(this.name);

System.out.println(" 测试没有通过!");

return false;

}

}

}

TestSuite.java

public class TestSuite extends TestCase {

public String name;

private List<TestCase> components = new ArrayList<>();

public TestSuite(String name){

this.name = name;

}

@Override

public void addEle(TestCase f){

components.add(f);

}

@Override

public void remove(TestCase f){

components.remove(f);

}

@Override

public boolean judgeResult() {

for (TestCase item : components){

if (item instanceof AtomTestCase){

if (item.judgeResult()){

continue;

}

}else {

item.judgeResult();

}

}

return true;

}

}

FourMixedOperatin.java

public class FourMixedOperation {

public int add(int a,int b){

return a + b;

}

}

### 实现效果



## 实验体会

通过组合模式，简化了代码，可以一致地使用组合结构和单个对象，通常用户不知道是一个叶结点还是一个组合组件，而且更容易增加新类型的组件。本次的实验自己学到了新的一种设计模式，通过本次的实验，对该模式理解得更加透彻了。