

SHANGHAI JIAO TONG UNIVERSITY



课程名称： 高级软件测试

学生姓名/学号: 唐天成515030910287

学生姓名/学号: 李林生515030910272

学生姓名/学号:

学生姓名/学号:

专 业: 软件工程

指导教师: 姚建国

学院(系): 电子信息与电气工程学院

目录

[一、 简介 3](#_Toc534472159)

[1.1、 编写目的 3](#_Toc534472160)

[1.2、测试范围 3](#_Toc534472161)

[二、 测试资源 3](#_Toc534472162)

[2.1、人力资源 3](#_Toc534472163)

[2.2、测试对象 3](#_Toc534472164)

[2.3、测试环境 4](#_Toc534472165)

[2.4、测试工具 4](#_Toc534472166)

[三、 进行回归测试 4](#_Toc534472167)

[3.1、ConvertPressure类 4](#_Toc534472168)

[3.1.1、getResult方法 4](#_Toc534472169)

[3.1.2、getConversion方法 4](#_Toc534472170)

[3.1.3、ConvertPressure类测试结果 5](#_Toc534472171)

[3.2、ConvertTemperature类 5](#_Toc534472172)

[3.2.1、getResult方法 5](#_Toc534472173)

[四、 对回归测试的分析 6](#_Toc534472174)

[4.1、分析回归测试在软件产品迭代更新中的优点 6](#_Toc534472175)

[4.2、分析所选回归测试方法在测试效率的提高程度 6](#_Toc534472176)

[4.2.1、再测试全部用例 6](#_Toc534472177)

[4.2.2、基于风险选择测试用例 6](#_Toc534472178)

# 简介

## 编写目的

本软件测试报告面向高级软件测试的课程作业，本次测试做的是回归测试。我们选取的程序是作业一和作业二都采用的单位转换程序，该程序分为温度和压强的单位转化，该程序一共有三个模块，分别为ConverterService , ConvertPressure, ConcertTemperature。我们在原来代码的基础上进行了修改和新功能的增加。我们增加了一个压强的单位，同时也修改了温度转换的逻辑实现。

之后，我们对该项目进行回归测试。由于代码的逻辑有了部分的修改，所以有的测试用例已经是不正确的了，因而我们首先在原来的代码里找出了那些已经没有用的测试用例，把它们删除掉了。之后我们采用不同的选择方法去筛选回归测试的测试用例集。

在回归测试结束之后我们编写测试报告，从而发现被测试的软件原有的功能是否还是符合设计的要求，发现软件开发中存在的各种问题，便于改正。

## 1.2、测试范围

本文档适用于“单位转化”程序的回归测试工作。

# 测试资源

## 2.1、人力资源

小组成员：

唐天成：主要负责的是修改实验一的源代码，使得每个测试用例产生错误，并且找了个自动化的变异测试工具mujava。

李林生：主要负责的是使用自动化工具mujava进行变异测试，使得每个模块的每个函数（方法），至少生成5个不同错误的变异体。

## 2.2、测试对象

单位转化的小程序，分为温度和压强的的单位转化，一共有三个模块，分别为ConverterService , ConvertPressure, ConcertTemperature，还有一个记录了常量的类Constants。

## 2.3、测试环境

系统：Windows10；java版本：1.8.0\_121

## 2.4、测试工具

测试工具：JUnit4；

# 进行回归测试

注：ConverterMutation项目为修改过后的代码，而ConverterOrigin就是修改以前的代码

## 3.1、ConvertPressure类

## 3.1.1、getResult方法

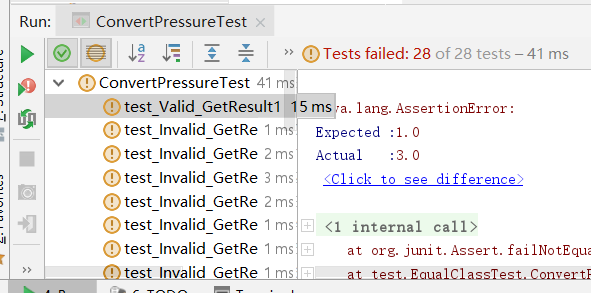
* 修改前：case Constants.PA:
* 修改后：case Constants.TEMPERATURE:
* 修改前：value = value \* 1000;
* 修改后：value = value + 1000;
* 修改前：value = value \* 101325;
* 修改后：value = value \* 200
* 修改前：return value;
* 修改后：return value + 2;

## 3.1.2、getConversion方法

* 修改前：case Constants.PA:
* 修改后：case Constants.KELVIN:
* 修改前：value = value / 1000;
* 修改后：value = 18181.121;
* 修改前：value = value / 101325;
* 修改后：value = value - 101325;
* 修改前：return value;
* 修改后：return -299.0;

## 3.1.3、ConvertPressure类测试结果

修改完源代码之后，运行作业一设计的测试用例，结果如下：



测试用例全部产生了错误

## 3.2、ConvertTemperature类

## 3.2.1、getResult方法

* 修改前：value = ((value - 32) \* 5) / 9;
* 修改后：value = ((value + 32) \* 5) \* 9;
* 修改前：case Constants.KELVIN:
* 修改后：case Constants.PRESSURE:
* 修改前：break;
* 修改后：value += 2; break;
* 修改前：return value;
* 修改后：return -223321.5451;

# 对回归测试的分析

## 4.1、分析回归测试在软件产品迭代更新中的优点

在软件产品的迭代更新中，回归测试是必不可少的一部分。

我们经常会去修改之前的代码，但是修改旧代码很有可能会引入新的错误，可能会导致原来代码或者是其他代码的问题。同样，新加入程序的代码可能也会对旧的代码产生影响。所以说，我们需要进行回归测试，来确保代码的修改没有引入新的错误。否则的话，我们无法保证在软件产品迭代更新中原有的功能是否还保持完整。所以说，回归测试的优点之一就是它验证了修改的正确性及其影响。

此外，回归测试有效避免了缺陷的放大，可以说实在软件产品迭代更新中的第一道，也是最好的防线。随着软件开发的越来越大，开发时产生的问题的可能也会越来越大。如果不做回归测试的话，很有可能就将一个缺陷引入到一个系统中了，这样就把问题放大了，所以说回归测试可以降低系统测试、维护升级时候的成本，可以节约很多时间和金钱，它是测试过程中的重要组成部分，甚至在极端的编程方法中，每天多次进行回归测试也不过分。

另外，回归测试是需要时间的，不过我们可以通过选择不同的方法来缩短回归测试的时间。我们可以不用重新测试全部用例的方法，通过一个回归测试的选择方法来选取用于回归测试的测试用例，这样我们测试的用例就没有那么多，也不会占用特别多的时间。虽然这样可能也有一点风险，但是一般来说，这些测试用例如果覆盖了风险大的代码，或者覆盖了修改部分的代码，那么相对来说就是比较安全了。

## 4.2、分析所选回归测试方法在测试效率的提高程度

我们在这次的回归测试中尝试了不同的回归测试方法。这些不同的选择方法有的更加着重于回归测试的效率，有的更加注重于安全性。接下来我们比较一下这些不同的方法。

## 4.2.1、再测试全部用例

再测试全部用例指的是重新运行全部的测试用例。

这个方法虽然安全，但是时间的开销往往是无法接受的。随着不断的开发，测试用例会越来越多，这样的重复测试会带来相当大的工作量，往往会超出能承受的范围。

所以说这个方法的测试效率很低。

## 4.2.2、基于风险选择测试用例

人工去分析哪些风险最高的代码部分，去优先选择那些关键的测试用例。

这个算是比较安全的方法了，毕竟就算非关键的代码出问题了，缺陷的严重性也不会太高。另外，测试的效率也大幅提升了。

这个回归测试方法不需要测试全部的用例，只需要选择其中一部分测试，时间就比再测试全部用例要短很多了。另外的时间开销也就只是去判断和维护各个测试用例或者模块的风险系数，有了制定好的风险标准，就可以方便地从中选择出测试用例了。