

OO第一单元总结——论表达式求导中的神奇操作

一、概述

本博客全部内容是根据三次作业的设计进行的总结，是博主的真实设计经历：

- 第一次作业：由于只存在幂函数和常数形式，因此直接构建Term类，一个main函数就可以了。
- 第二次作业：采用变量替换、表达式树形式完成设计（重构）
- 第三次作业：继承第二次作业的表达式树，定义新运算；然后暴力破解WF（没有重构）

然后说说每次评测结果，虽然很难堪但是也算得上是对自己的一个提醒吧：

- 第一次强测：问题出现在toString上，意外地漏掉了一些常数
- 第二次强测：问题同样在toString上，在最后表达式树输出的时候出现了一些问题
- 第三次强测：问题主要在没有考虑周全WF的形式，比如指数爆Long，括号不匹配、sin内部为空等


整体而言第一单元作业完成质量不高，不仅在代码设计上没有合理运用面对对象的思想，同时程序编写能力较差，代码风格较为难堪。希望在之后的单元中能够有所改进。

二、作业评析

第一周作业：幂函数和简单表达式求导

1、设计思路

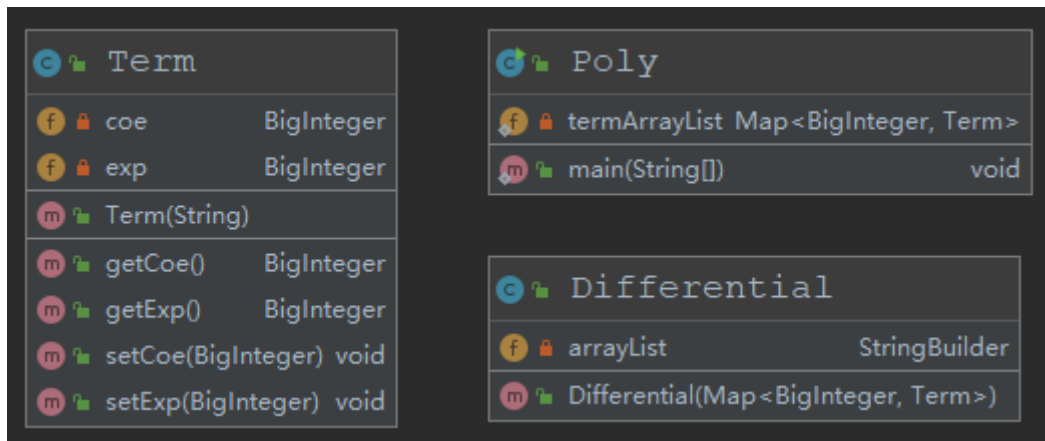
- 设计架构

 OO第一次作业

代码优势：代码简单，整体行数较少；debug过程比较容易

代码缺点：正则表达式是整个设计的关键，容易出错；一边匹配一边合并优化容易出bug；代码语言一般；对于之后的架构不再适用

- UML类图



从UML类图可以看出，模块之间相对独立。事实上只有Term可以作为一个大类，其他两者都是模块化的方法，更类似于一种函数调用。

- 复杂度分析

method	▲	CogC	ev(G)	iv(G)	v(G)
Term.getCoe()		0	1	1	1
Term.getExp()		0	1	1	1
Term.setCoe(BigInteger)		0	1	1	1
Term.setExp(BigInteger)		0	1	1	1
Term.Term(String)		2	1	3	3
Poly.main(String[])		8	1	5	5
Differential.Differential(Map<BigInteger, Term>)		18	1	8	9
Total		28	7	20	21
Average		4.00	1.00	2.86	3.00

- 1 **ev(G)**: 基本复杂度，用于衡量代码非结构化程度的。数值越高证明结构性越差，不利于维护。
- 2 **iv(G)**: 模块设计复杂度，用于衡量不同模块之间的耦合关系，数值越高证明模块间耦合程度越大。
- 3 **v(G)**: 圈复杂度，用来衡量一个模块判定结构的复杂程度，合理的预防错误所需测试的最少路径条数，圈复杂度大说明程序代码可能质量低且难于测试和维护。
- 4 **Ocavg**: 类平均圈复杂度。
- 5 **WMC**: 类总圈复杂度。

从上面复杂度分析的结果来看，在 `Differential` 类中设计较为糟糕。实际上 `Differential` 类中有着太多的 `if-else` 结构，而且嵌套层数较大（用于优化时），这种设计极大影响了程序性能。

2、bug经验

- 不要一边求导一边优化

一边求导一边优化极其容易出bug，本人的教训如下：

在 `Differential` 求导过程中：

```

1 //termMap 存储了所有的因子(定义成了Term (逃))
2 for (BigInteger item : termMap.keySet())
3 {
4     Term term = termMap.get(item);
5     //系数不为0
6     if (!term.getCoe().equals(BigInteger.ZERO))
7     {
8         //指数为1
9         if (term.getExp().equals(BigInteger.ONE)){.....}
10        //指数不为0
11        else if (!term.getExp().equals(BigInteger.ZERO))
12        {
13            //求导
14        }
15    }
16    arrayList.append("+");
17 }
18 String string = arrayList.toString();
19 string = string.substring(0, string.length() - 1);
20 if (string.length == 0) {OUTPUT(0)}
21 else {OUTPUT(string)}

```

这样看来似乎没有什么问题，因为我们知道最后结果肯定会有一个`+`，因此用`substring`方法去除即可。

评测如下：

```

1 STDIN:
2
3 0000*x**1234*x**123+000*x**12*23*34*45+-000*x**-03*x**-04
4
5 STOUT:
6 +
7
8 EXPECTED ANSWER:
9
10 0

```

bug在于我默认系数为0的时候直接忽略，如果全部忽略则输出0，但是第16行会使得所有系数为0的项全部加上`+`，而我只处理了最后一个。从而造成错误。

很简单的错误，很多同学可能觉得这个错误很糟糕，我也承认如此，在今后的测试数据构造和代码分析中还是尽量多多学习🧐。

3、互测

实际上个人没有学习测评机，因此实际上第一互测没有很多经验。

第二周作业：三角函数和嵌套模式

1、设计思路（重构）

- 设计架构

这个地方根据自己写的main函数详细说说。这一次重构直接奠定了第三次作业的工作量。

main函数如下：

```
1 public static void main(String[] args) {
2     Scanner in = new Scanner(System.in);
3     String polys = in.nextLine();
4     //先分割表达式成为标准的中缀的形式
5     polys = segments(polys);
6     //构建后缀表达式
7     ArrayList<String> strings = new
BuildPostFix(polys).getExp();
8     //生成表达式树
9     Tree tree = buildTree(strings);
10    //输出求导后的结果
11    System.out.println(tree.setDerivation().toString());
12 }
```

显然，第二次作业是通过二叉树（表达式树）写的。

代码优点：整个架构具有一定的扩展性；过程思想很明确。

代码缺点：冗杂；风格低下；main方法太长；没有优化，结果触目惊心。

- `segment` 函数

 OO第二次作业

1、符号代换的思想：这个地方实际上是因为自己在学习数据结构时对二叉表达式树理解不太深刻，印象中其只能解决**非负的、整数的**组合运算。因此**为了简化构建逆波兰表达式**的过程，将其**整个复杂表达式转变成只存在符号**的形式。

举例：

```

1 STDIN:
2      x**+2    +4    *   sin(x)    ** -2    *x
3 STDOUT:
4      P1 + N1 * T1 * P2
5 //P1: x**+2
6 //N1: 4
7 //T1: sin(x) ** -2
8 //P2: x

```

2、关于负数和带符号数：事实上逆波兰表达式是可以处理负数的，但是由于本人能力确实不太行，因此对于负数和带符号正数的处理中，依旧面向题目中可能出现的格式来写的（有点面向测评那味儿了）。

- 对于指数中的带符号数，利用正则表达式在匹配的时候就直接换掉对应的形式。
- 对于负数，`-` 变成 `0-` 的形式，正则表达式如下：

```

1 str = str.replaceAll("(?!((\\d)|[ ]))-", "N0-");
2 //在Map存储中N0默认为常数0

```

能够换成 `N0-` 的前提是前面不是 `)` 和 `\\d`（即 `Pn`、`Nn`、`Tn`），否则会出现错误形式。

- 对于带符号正数，同上：

```

1 str = str.replaceAll("(?!((\\d)|[ ]))\\+", "");
2 //直接清除

```

这个地方建议读者仔细思考其合理性，个人当时也是思考了一段时间才发觉正确形式只有这几类。如果有什么问题，也希望大家在评论区指点指点。

- 补充：在匹配常数的过程中，我采用了这种正则表达式：

```

1 string normalNumber = "(\\d+)|((?<=\\*)-\\d+)";

```

仅包括数字本身以及**在连乘中出现的负数**，带正号的整数会在2中消除正号，其余的负数同样在2中进行了处理，也避免了连乘过程中出现的负号出现干扰。

```

1  STDIN:
2      x *-2
3  WRONG STOUT:
4      P1 * N0-N1
5  RIGHT STOUT:
6      P1 * N1
7  //直接当做整个整数处理
8  STDIN:
9      -2 * 3 *x **+3 -(x-2)- 2
10 STOUT:
11     N0-N1*N2*P1-(P2-N3)-N4

```

实际上先不说读者们是否看得明白，我已经尽力将整个过程不加代码的分析完成了。但是读者可能都会觉得这个方法冗杂、多余。确实是这样，在自己代码能力不足的情况下，没有足够的思维能力去解决负数等其他问题，只能出此下策。这种方式会在第三次作业中再次体现。

○ BuildPostFix (string) 函数、Tree函数

这两个函数是构建后缀表达式函数和生成树函数。相对而言没有很多需要说明的，第二次直接按照网上的后缀表达式和生成树的模版套用即可。

下面给出一些例子进行详细说明：

```

1  STDIN:
2      x ** -3 + -2*x +sin(x)  + (sin(x)+(cos(x)**-2 ))
3  segment:
4      P1-N1*P2+T1+(T2+(T3))
5  BuildPostFix:
6      ArrayList = {P1,N1,P2,*,-,T1,+,T2,T3,+,+}
7  Tree:
8      如下图

```

 OO第二次作业树的样例

从而可以得到非常容易处理的式子。

○ setDerivation函数

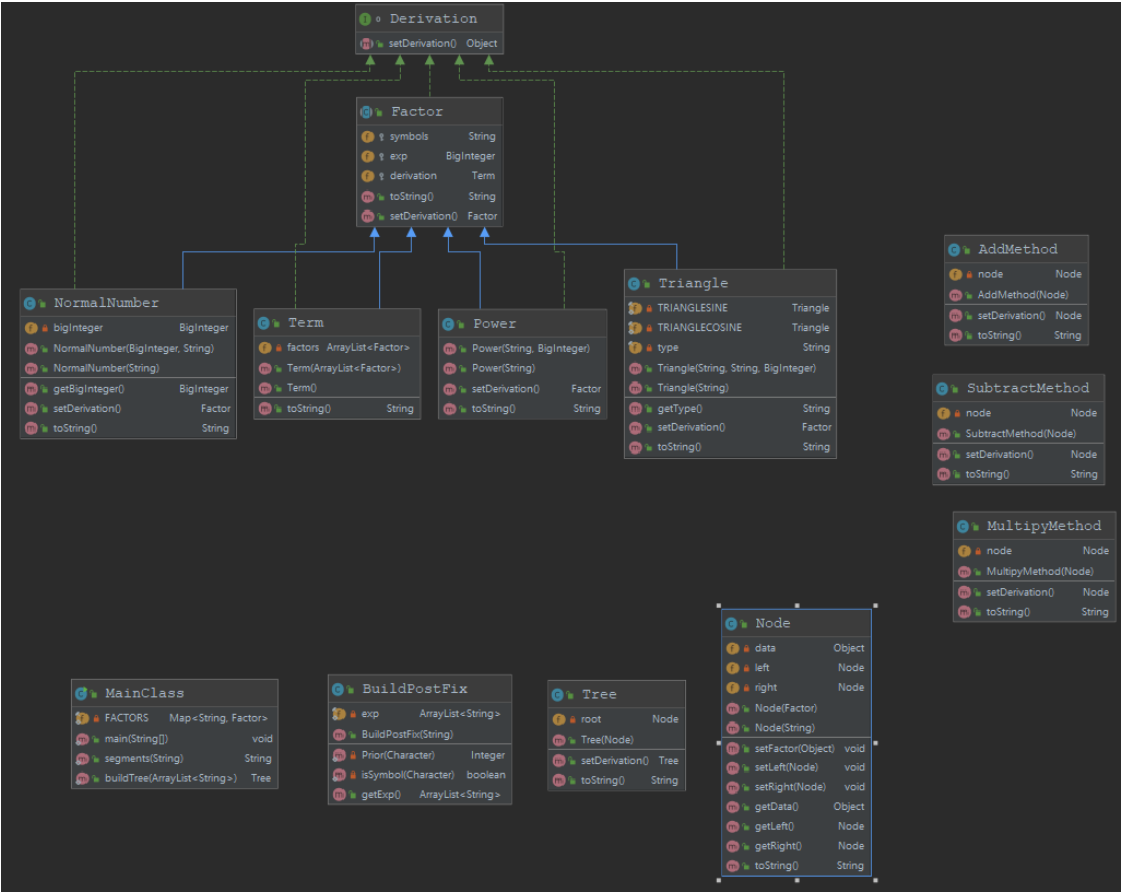
这个函数定义了求导方式，根据表达式树进行求导。

这里就不详细说了，直接放图，之后的操作就比较容易理解了。

 OO第二次作业树的求导

最终结果就不展示了，根据树的结构进行中序遍历即可。

• UML类图



这里分别进行说明：

○ 继承类

	独有属性	构造函数	父类属性	toString方法
NormalNumber	Big integer: 存储大数	正则表达	setDerivation	
Power	无	正则表达	setDerivation	
Triangle	type: 定义cos和sin TriangleSine: sin常量 TrangleCosin: cos常量	正则表达	setDerivation	
Term	factors: 存储前三者求导后的内容	连乘	setDerivation	

○ 加减乘除类

```

1 //包含树的结点
2 private Node node;
3 //构造方式
4 public Method(Node node)
5 {
6     this.node = node;
7 }
8 //求导方式，这里给出左子树进行示例
9 public Node setDerivation()
10 {
11     //核心：依次下降、依次递归
12     if (value1 instanceof Factor)
13     {
14         leftNode = new Node(((Factor)
15         value1).setDerivation().derivation);
16     }

```



```

16     else if (value1 instanceof String)
17     {
18         if (value1.equals("-")) {
19             leftNode = new
SubtractMethod(left).setDerivation();
20         } else if (value1.equals("+")) {
21             leftNode = new
AddMethod(left).setDerivation();
22         } else if (value1.equals("*")) {
23             leftNode = new
MultiplyMethod(left).setDerivation();
24         }
25     }
26     ret.setLeft(leftNode);
27 }
28 //右子树同理
29 public String toString()
30 {
31     return "(" +
32         (node.getLeft() == null ? "" :
node.getLeft().toString()) +
33         node.getData() +
34         (node.getRight() == null ? "" :
node.getRight().toString()) +
35         ")";
36 }

```

这里用语言描述构造的思想有点晦涩，于是这里我就直接给出部分代码了，读者可以从这种代码结构中了解到我的整个求导过程以及toString函数是如何书写的。

- 复杂度分析

复杂度分析很惨。在代码风格上运用多个 *protected* 已经宣布低分了。这里给出类的复杂度，至于方法复杂度分析由于太长了（一整张屏幕都不够）就省略了。

class	OCavg	OCmax	WMC
src.AddMethod	5.00	11	15
src.BuildPostFix	4.25	12	17
src.Factor	1.00	1	2
src.MainClass	4.33	7	13
src.MultiplyMethc	5.00	11	15
src.Node	2.11	11	19
src.NormalNumb	1.60	3	8
src.Power	2.75	4	11
src.SubtractMeth	5.00	11	15
src.Term	1.33	2	4
src.Tree	2.33	5	7
src.Triangle	3.00	6	15
Total			141
Average	3.00	7.00	11.75

整体而言标红的地方很多，实际考虑到我使用了很多函数，而且有些函数直接嵌套在标准类里面。

- main函数：main函数中将segment和buildtree函数直接嵌进去了。
main函数有两百多行🤖。
- 三种加减乘除：写得有点像工厂类，因此整个耦合比较重。

2、bug经验

• 节点的克隆很重要

事实上，clone (x) , new (v) 🤖。

否则非常容易出现以下问题：

```

1  STDIN:
2      x * (sin(x) + (cos(x) + 2))
3  WRONG STOUT:
4      cos(x) - sin(x) + x * (cos(x) - sin(x))
5  RIGHT STOUT:
6      cos(x) - sin(x) + x * (sin(x) + (cos(x) + 2))

```

一个节点在求导后没有保存原值，从而在后面始终是其导数形式。

解决办法：new。求导之前直接new一个保存求导后的值，或者克隆应该也可以。

new真的好用🤖。

• toString一定要仔细

第二次作业强测直接爆了，导致没有进入互测，最终发现问题依旧在toString上面。之前的toString是这样写的：

```

1 public String toString()
2 {
3     return "(" +
4         (node.getLeft() == null ? "0" :
5         node.getLeft().toString()) +
6         node.getData() +
7         (node.getRight() == null ? "0" :
8         node.getRight().toString()) +
9         ")";
10 }

```

这个多出来的0直接使我的整个强测暴毙。

第三周作业：三角嵌套和错误表达式

1、设计思路（非重构）

先考虑三角嵌套：同样我可以新定义一个运算方式，运用表达式树进行处理。

对于WF：暴力破解。根据所有可能出现的WF进行暴力破解，分三个阶段进行。

- 设计架构

和第二次相当类似。

 OO第三次作业

- **定义新运算'/'**

这种定义新运算的考虑来源于怎么对sin嵌套类型进行处理。后来想到可以这么处理：

- 将 $\sin(\dots)$ 分成两部分，一部分是其本身，一部分是其内容。
- 定义新运算'/'，其中左子树为其本身，右子树为其内容。
- 求导时，'/'变成'*'，左子树直接变成cos类似形式，右边为其内容的导数。

另外在构建后缀表达式时，'/'优先级大于'*'，小于括号。

示例：

```

1  STDIN:
2      sin( ( x + sin( cos(x)**2 ) ) )
3  STOUT:
4      T1/(P1+T2/(T3/(P2)))
5  //T1:  sin( ( x + sin( cos(x)**2 ) ) )
6  //P1:  x
7  //T2:  sin( cos(x)**2 )
8  //T3:  cos(x)**2
9  //P2:  x
10 Tree:

```

 OO第三次作业的树

暴力WF

第一阶段WF：输入之后

核心在于找出WF的**空格形式**，便于之后的去空格操作。

```

1  String regex = "[\\f\\n\\r\\v]";
2  regex = "s\\s+in|si\\s+n|c\\s+os|co\\s+s";
3  regex = "\\s*\\s+\\s*";
4  //---x,---sinx,---cosx
5      regex = "[+-]\\s*[+-]\\s*[+-]\\s*[xsc]";
6  //--- 4
7      regex = "[+-]\\s*[+-]\\s*[+-]\\s+[\\d]";
8  //too many symbols
9      regex = "[+-]\\s*[+-]\\s*[+-]\\s*[+-]";
10 //x*- 2,x**+ 3, sin( - 3), cos ( - 3)
11      regex = "(\\s*|(sin\\s*[()](cos\\s*[()])\\s*[+-]\\s+[\\d]";
12 //最后是非法其他字符[a-z]等，略。可以先replaceAll所有合法形式，最后看看是否为空

```

第二阶段WF：

核心在于两种形式：

```

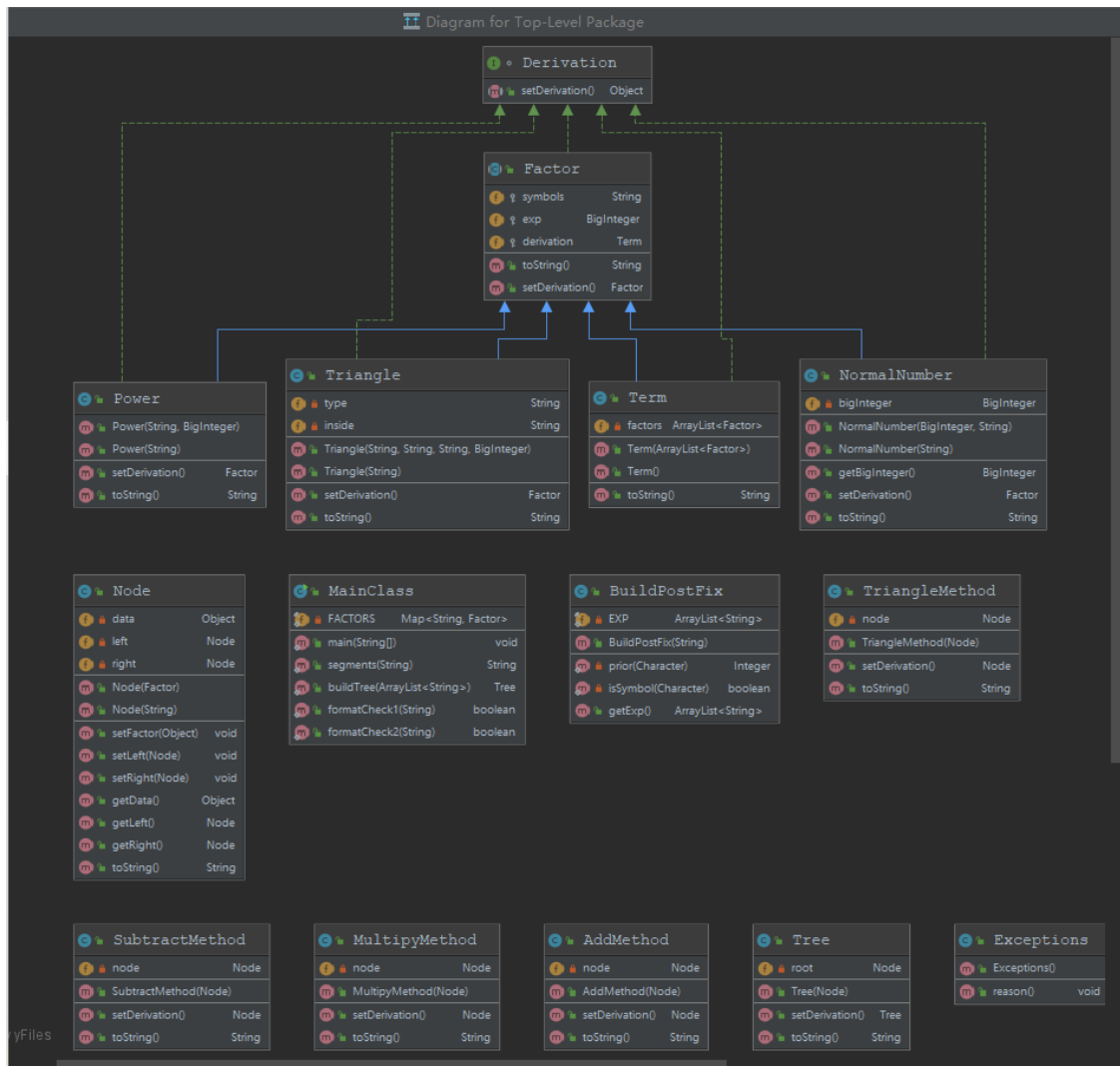
1  //2x,xx,xsin(x),sin(x)sin(x),x(...),2(...),sin(x)
   (...),(..)(..)
2      String regex = "([\\d])[PTN()]";
3  //(x+1)**2 won't be matched
4      s = s.replaceAll(RIGHT_REGEX, "");
5      isEmpty(s);

```

第三阶段WF：

核心在于指数大于50、括号不匹配、内部为空、sin内部不为因子等,这个地方分布较为零散和复杂,就不细说了。

- UML类图



实际上只在第二次作业基础上加了一个 **TriangleMethod** 和 **Exceptions**。前者用于处理 / 运算, 后者处理WF。

这个部分和第二次作业的分析几乎完全相同, 这里就不赘述了, 只说明两点:

1、针对 **TriangleMethod** 求导方式如下:

 OO第三次作业除法

2、在 `toString` 中对于 '/' 只需要输出左子树, 因为右子树是重复的。

- 复杂度分析

同样只给一下类的复杂度分析。

class ▲	OCavg	OCmax	WMC
AddMethod	5.67	13	17
BuildPostFix	4.75	13	19
Exceptions	1.00	1	2
Factor	1.00	1	2
MainClass	8.60	23	43
MultiplyMethod	5.67	13	17
Node	2.22	12	20
NormalNumber	1.60	3	8
Power	3.00	4	12
SubtractMethod	5.67	13	17
Term	1.33	2	4
Tree	2.67	6	8
Triangle	9.25	29	37
TriangleMethod	5.00	13	15
Total			221
Average	4.17	10.43	15.79

- main函数：main函数复杂度更大是因为其更长了。。增加了两个WF的判断方法，因此本来就很长的main越来越长。实际上可以写到另外一个函数中。
- Triangle类：这个类本身应该继承自Factor，但是由于存在嵌套，因此在类本身中也需要进行WF形式的检查，大大增加了构造方法的代码长度。
- 其余Method：和第二次作业分析的情况类似，这一次复杂度变化不是很大。

2、bug经验

• WF考虑不全

这个是硬伤，毕竟我是按照暴力WF做的，不过好在强测中虽然也有WA，但是很多情况都考虑到了，这个地方对于我这种面对过程写的极其不友好，这个bug只能构造更多的数据去de，尽量考虑周全。在弱测和中测、以及互测中，都是通过WF进行de和hack的。

• 爆long

有一位互测玩家很聪明，直接拿40个9来测，对于我这种分类的自然分到了NormalNumber中，而且利用

`BigInteger.valueOf(Long.parseLong(str))`，然后直接爆long了，后来发现可以直接用`BigInteger(str)`。不过当时这位玩家直接用这个hack了好几位。

3、互测

互测关键在于如何构造隐蔽的WF形式来hack别人，或者对于正确形式别人判定成WF了，本次互测故意构造了一些边边角角的数据和多重嵌套，成功hack到别人。

三、总结体会

本次总结就到这里了，事实上我还是对自己觉得不太满意。一方面自己的构造真的很糟糕，想不出别的方法就构造一些（歪门邪道）来求解，没有很好用到面对对象的方法，也没有设计优化。。就感觉似乎写了一长串代码而已。这种感觉很糟糕。

总而言之，第一单元已经结束了，希望在之后的学习中能有所进步。