OO第一单元总结

前言

第一单元的三次作业，因为自己的拖延症，加上一些确实无法避免的事情耽搁，每每到周日甚至周一才开工，虽然前两次作业完成地比较顺利，但到了第三次作业就差点把自己玩儿掉里。第三次作业到周日下午才开工，花了整整一下午的时间去构思、设计，花了一整天的时间写代码，又花了半个小时说服自己完全抛弃了之前的工作，重构代码，历经千辛万苦~~(我知道是我自己作的)~~，终于在连熬了两天夜后完成了这次作业。现对这三次作业进行一个系统的总结，以此来审视自己在这三周的OO学习中的收获与不足。

第一次作业 (PolyDerivation)

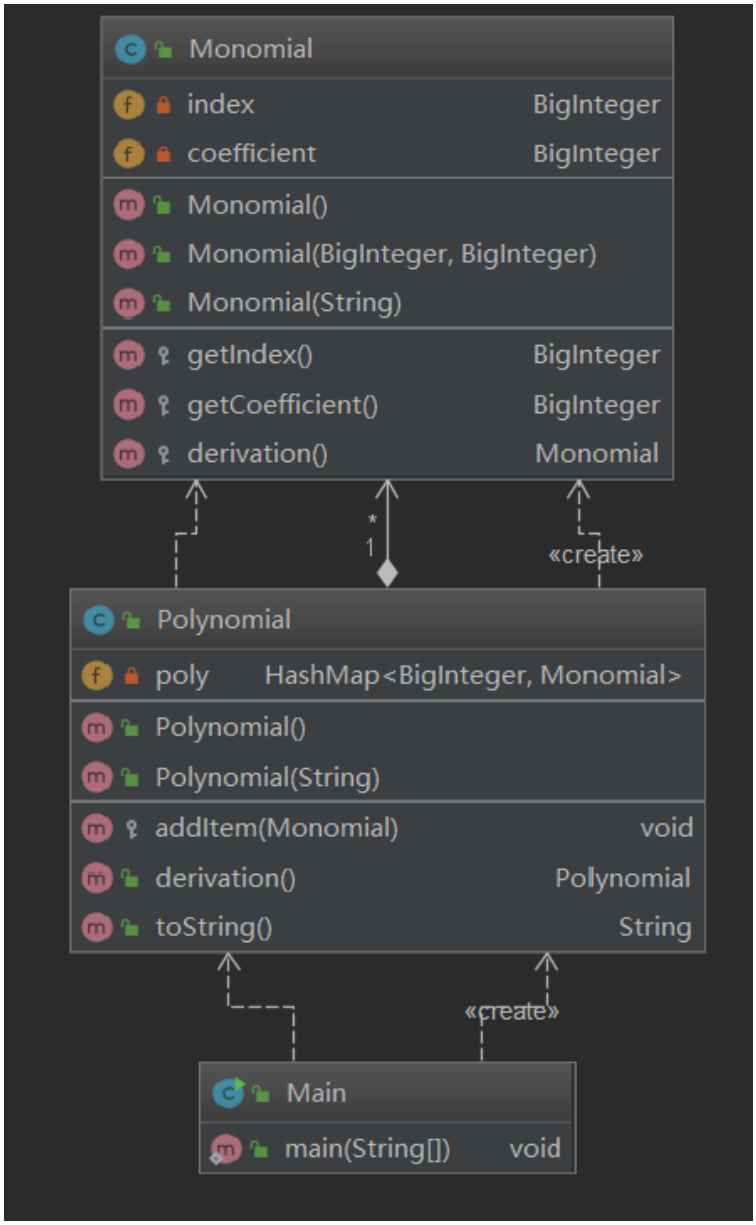
第一次作业需要完成对简单多项式的求导，算是个入门级任务，难度不高，工程量也不大(*小几百行的量，见下图*)，难点在于正则表达式的熟练运用。

思路

1. 首先对字符串进行预处理：
   - 扫描整个字符串，检测是否为**空串**，以及是否包含**非法字符**
2. 根据项与项之间的规律将多项式分割为若干个单项式
   - 经观察得到，每个项的结尾只可能为 `数字` 或 `x`，而每个项的开头只可能为 `+` 或 `-`（~~当然第一项例外~~），故可利用 `[x\\d][ \t]*[+-]` 进行正则匹配，在项与项之间添加一特殊标记符 `@`，即可利用 `str.split("@")` 得到各个单项式。
3. 逐个构造单项式
   - 单项式的格式根据**系数与指数**的不同共有五种情况，枚举并构造。

程序结构

利用IDEA自带插件生成UML图，程序结构大致如下，共包含三个类：Main, Polynomial, Monomial。



度量分析

使用statistic插件来统计代码量。

Source File	Total Lines	Source Code Lines
Main.java	13	10
Monomial.java	119	98
Polynomial.java	137	99
Test.java	57	14
Total:	326	221

使用MetricsReloaded插件来度量代码复杂度。

Method	ev(G)	iv(G)	v(G)
Main.main(String[])	1	1	1
Monomial.Monomial()	1	1	1
Monomial.Monomial(BigInteger,BigInteger)	1	1	1
Monomial.Monomial(String)	1	6	8
Monomial.derivation()	1	2	2
Monomial.getCoefficient()	1	1	1
Monomial.getIndex()	1	1	1
Polynomial.Polynomial()	1	1	1
Polynomial.Polynomial(String)	1	4	4
Polynomial.addItem(Monomial)	2	2	3
Polynomial.derivation()	1	2	2
Polynomial.toString()	3	10	12
Test.main(String[])	1	1	2

从表格中可以看出，`Polynomial.toString()`方法的模块复杂度与独立路径条数较高，我发现由于我在逻辑块中为了**优化输出**使用了大量的特判，导致代码不仅可读性差，模块复用的可能性基本为0，且更容易犯错。

互测

- 由于我是枚举五种情况进行匹配，而并非使用大正则，压力测试达到million级，因而不存在爆栈的可能。但我在`Polynomial`类中对输入字符串进行预处理时，先`.trim()`了一下，导致被各种`\f`、`\v`等非法空白符轰炸。
- 在分析roommates的程序bug时重点还是放在了输入处理上，如：遇到大正则就进行**压力测试**，挑正则表达式的刺等等，而并没有把重点放在对代码逻辑的理解上，这种面向分数debug的行为是极为幼稚且不可取的，在后两次作业中我也在尽量避免这种思想及行为。

自评

- 本次作业难度较低，对新手比较友好。但毕竟是第一次写，对Java这门语言仍处于**陌生**状态，对于**正则表达式**等概念及其具体原理也不甚理解，在代码的**工程性方面**更是并没有做过多的考量，写出来的代码可以说只是披着OO的外衣，十分的**面向过程**。

第二次作业 (PolyDeriPlus)

第二次作业仍是多项式求导，但新增**正余弦函数**，且加入了**乘法组合项**，新增因子的定义。虽然工程量有所增加，难度也有所提升，但更像是**暴风雨前的宁静**，话不多说，进入正题。

思路

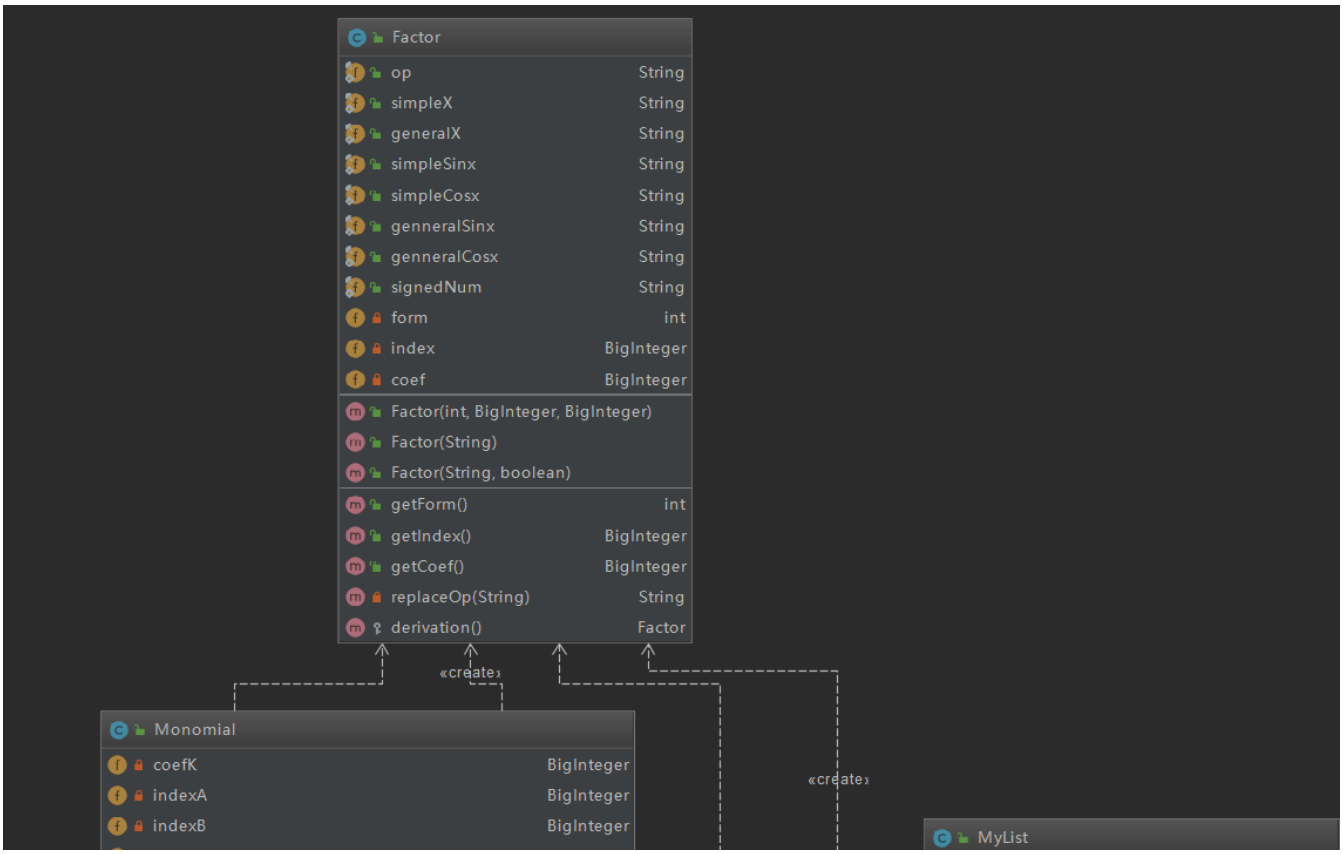
- 表达式的构造
  - 整体思路同第一次作业，在表达式的构造上，新增Factor类。由于各因子之间以`*`相连，故可利用`.split("*")`将项分割为若干个因子逐一进行构造。
- 项的存储方式
  - 由于任何项都是由若干个常数、幂函数以及三角函数构成，故可将项抽象为 $k * x^a * sin(x)^b * cos(x)^c$ ，即可以**四元组(k, a, b, c)**的形式进行存储。

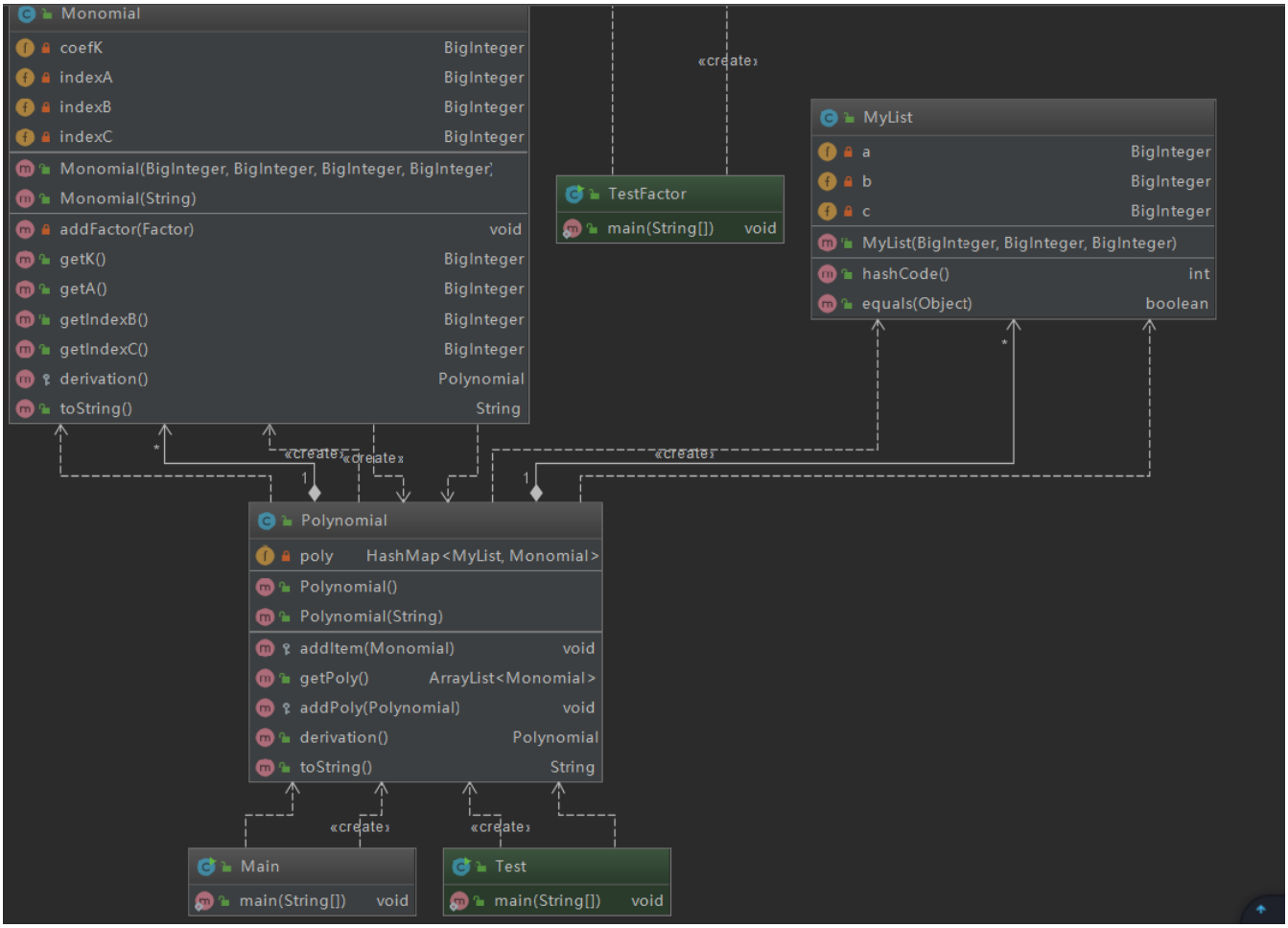
- 这样做不仅**合并同类项**十分方便，求导规则也一目了然，如下：

  $(k*x^a*sin(x)^b*cos(x)^c)' = k*a*x^{a-1}*sin(x)^b*cos(x)^c + k*b*x^a*sin(x)^{a-1}*cos(x)^{b+1} - k*c*x^a*sin(x)^{b+1}*cos(x)^{c-}$

程序结构

UML图如下，结构与第一次作业类似。`Mylist`是为了便于合并同类项而自定义的类，重写了`hashcode()`以及`equals()`方法。





度量分析

Source File	Total Lines	Source Code Lines
Factor.java	314	245
Main.java	24	15
Monomial.java	184	126
MyList.java	24	19
Polynomial.java	163	97
TempTrash.java	71	3
Test.java	20	14
TestFactor.java	8	6
TestMonomial.java	16	9
Total:	824	534

Method	ev(G)	iv(G)	v(G)
Factor.Factor(String)	1	9	10
Factor.Factor(String,boolean)	1	14	14
Factor.Factor(int,BigInteger,BigInteger)	1	1	1
Factor.derivation()	2	5	9
Factor.getCoef()	1	1	1
Factor.getForm()	1	1	1
Factor.getIndex()	1	1	1
Factor.replaceOp(String)	1	1	1
Main.main(String[])	1	2	2
Monomial.Monomial(BigInteger,BigInteger,BigInteger,BigInteger)	1	1	1
Monomial.Monomial(String)	1	3	4
Monomial.addFactor(Factor)	2	2	6
Monomial.derivation()	1	1	1
Monomial.getA()	1	1	1
Monomial.getIndexB()	1	1	1
Monomial.getIndexC()	1	1	1
Monomial.getK()	1	1	1
Monomial.toString()	2	11	12
MyList.MyList(BigInteger,BigInteger,BigInteger)	1	1	1
MyList.equals(Object)	1	1	1
MyList.hashCode()	1	1	1
Polynomial.Polynomial()	1	1	1
Polynomial.Polynomial(String)	1	4	4
Polynomial.addItem(Monomial)	2	2	3
Polynomial.addPoly(Polynomial)	1	2	2
Polynomial.derivation()	1	2	2
Polynomial.getPoly()	1	2	2
Polynomial.toString()	5	4	6
Test.main(String[])	1	2	2
TestFactor.main(String[])	1	1	1
TestMonomial.main(String[])	1	1	2

- 从表格可以看出，主要问题仍然集中在`toString()`方法中，同第一次作业，为了尽可能地优化输出，加入了许多特判逻辑，使得代码看起来非常不友好。

互测

- 本次互测被找出一个bug：在构造各个因子时，我直接`.split("*")`，但由于`split()`方法的特性，当遇到`**`或者类似`2*x*`的形式会误判为正确形式，对面的Assassin也有同样的bug。

- 此外，本次作业有一个坑点~~(也是最大的槽点)~~为形如 `+++1` 或 `++x` 的表达式竟然是合法的，且中间可以在任何位置加入任意数量的空白符。因而针对这个点进行了盲测，果不其然，中标。

自评

- 本次作业延用了第一次作业的结构，因而流程十分类似，仅仅是增加了Factor类，仍然十分的**面向过程。**
- 此外，在Factor类中，我为第一个因子单独写了一个构造方法，没有很好地利用好正则表达式，导致代码冗余又臃肿，十分难看。

第三次作业 (PolyDeriFinal)

第三次作业则加入了更多的组合规则，尤其是最令人头疼的**嵌套规则**，且极易混淆**表达式因子**与**嵌套因子。**难点在于如何递归地构造表达式树。

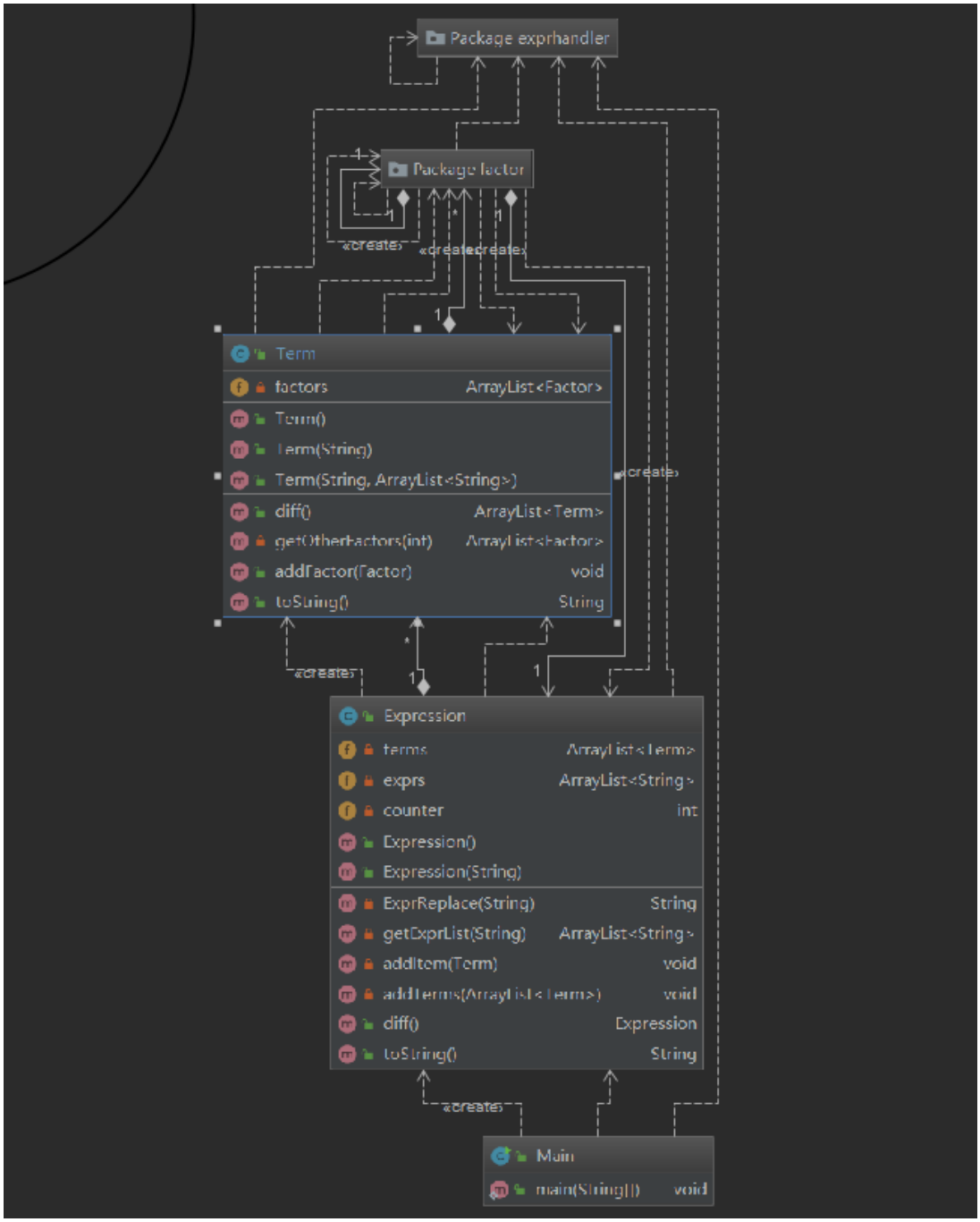
思路

1. 输入字符串预处理
2. 再次扫描字符串，将所有表达式因子及嵌套因子替换为 `(E)` 作为标记。
3. 构造项
4. 构造因子，若为表达式因子，则返回第一步。
5. 重复上述过程，得到一个由 `Arraylist` 存储的抽象的表达式树。
6. 递归求导

程序结构

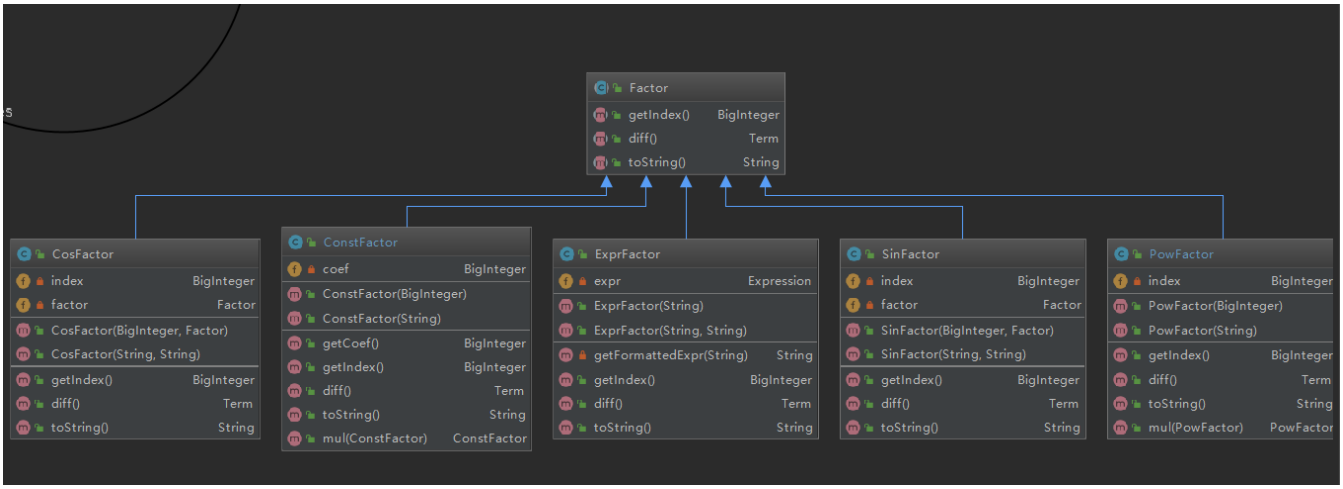
从UML图可以看到，程序主要包括四个模块：Expression，Term，Factor，exprhandler。

其中，exprhandler是抽象出来的一个层次，其它三个模块层次分明，呈循环结构，实现递归调用。



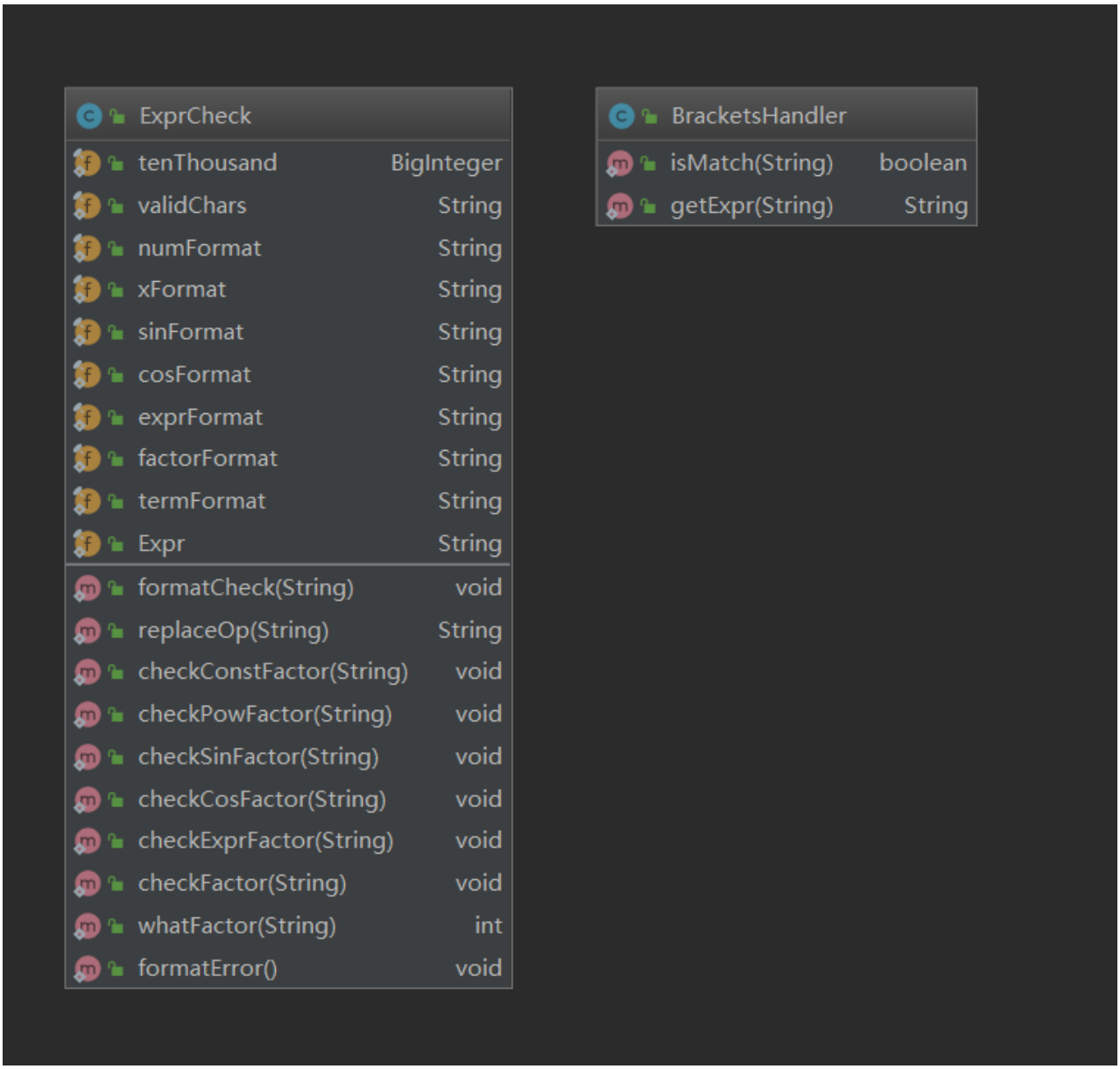
Factor是一个抽象类，包括`getIndex()`，`diff()`，以及`toString()`方法。

其子类有常数类，幂函数类，三角函数类以及表达式类。



exprhandler包内结构如下：

包括一些常量，字符串的预处理，括号匹配，以及格式检查等等。



度量分析

由下图可知，仅从代码量来看，第三次作业与第二次作业的800+ lines相比并无较大差别，而难度确实第二次作业无可比拟的，故从一定程度上可以看出代码写的更加干净，精简了。

Source File	Total Lines	Source Code Lines
BracketsHandler.java	71	47
ConstFactor.java	45	33
CosFactor.java	84	63
ExprCheck.java	165	127
Expression.java	110	87
ExprFactor.java	68	52
Factor.java	17	8
Main.java	22	16
PowFactor.java	76	60
SinFactor.java	84	62
Term.java	165	141
Total:	907	696

由下表可以看出，平均代码复杂度较前两次有所降低，但`Term`的构造方法复杂度却如同鹤立鸡群。经查，在Term中构造各个因子时，用了`swtich()`结构，而在阿里Java开发手册中有明确提到：

> 11.【强制】构造方法里面禁止加入任何业务逻辑，如果有初始化逻辑，请放在init方法中。

因而应尽量避免在构造函数中携带过多的计算逻辑。

Method	ev(G)	iv(G)	v(G)
expression.Expression.ExprReplace(String)	1	3	3
expression.Expression.Expression()	1	1	1
expression.Expression.Expression(String)	1	3	3
expression.Expression.addItem(Term)	1	1	1
expression.Expression.addTerms(ArrayList)	1	1	1
expression.Expression.diff()	1	2	2
expression.Expression.getExprList(String)	1	2	2
expression.Expression.toString()	1	2	2
expression.Main.main(String[])	1	2	2
expression.Term.Term()	1	1	1
expression.Term.Term(String)	1	2	2
expression.Term.Term(String,ArrayList)	2	11	16
expression.Term.addFactor(Factor)	2	2	2
expression.Term.diff()	1	3	3
expression.Term.getOtherFactors(int)	3	3	3
expression.Term.toString()	3	2	3
expression.exprhandler.BracketsHandler.getExpr(String)	5	5	5
expression.exprhandler.BracketsHandler.isMatch(String)	7	5	7
expression.exprhandler.ExprCheck.checkConstFactor(String)	1	2	2
expression.exprhandler.ExprCheck.checkCosFactor(String)	1	2	2
expression.exprhandler.ExprCheck.checkExprFactor(String)	1	2	2
expression.exprhandler.ExprCheck.checkFactor(String)	1	2	2
expression.exprhandler.ExprCheck.checkPowFactor(String)	1	2	2
expression.exprhandler.ExprCheck.checkSinFactor(String)	1	2	2
expression.exprhandler.ExprCheck.formatCheck(String)	1	4	4
expression.exprhandler.ExprCheck.formatError()	1	1	1
expression.exprhandler.ExprCheck.replaceOp(String)	1	1	1
expression.exprhandler.ExprCheck.whatFactor(String)	6	6	6
expression.factor.ConstFactor.ConstFactor(BigInteger)	1	1	1
expression.factor.ConstFactor.ConstFactor(String)	1	1	1
expression.factor.ConstFactor.diff()	1	1	1
expression.factor.ConstFactor.getCoef()	1	1	1
expression.factor.ConstFactor.getIndex()	1	1	1
expression.factor.ConstFactor.mul(ConstFactor)	1	1	1
expression.factor.ConstFactor.toString()	1	1	1
expression.factor.CosFactor.CosFactor(BigInteger,Factor)	1	2	2

Method	ev(G)	iv(G)	v(G)
expression.factor.CosFactor.CosFactor(String,String)	1	3	3
expression.factor.CosFactor.diff()	2	1	2
expression.factor.CosFactor.getIndex()	1	1	1
expression.factor.CosFactor.toString()	1	3	3
expression.factor.ExprFactor.ExprFactor(String)	1	1	1
expression.factor.ExprFactor.ExprFactor(String,String)	1	2	2
expression.factor.ExprFactor.diff()	1	1	1
expression.factor.ExprFactor.getFormattedExpr(String)	1	4	4
expression.factor.ExprFactor.getIndex()	1	1	1
expression.factor.ExprFactor.toString()	1	1	1
expression.factor.PowFactor.PowFactor(BigInteger)	1	1	1
expression.factor.PowFactor.PowFactor(String)	1	3	3
expression.factor.PowFactor.diff()	2	2	3
expression.factor.PowFactor.getIndex()	1	1	1
expression.factor.PowFactor.mul(PowFactor)	1	1	1
expression.factor.PowFactor.toString()	1	3	3
expression.factor.SinFactor.SinFactor(BigInteger,Factor)	1	2	2
expression.factor.SinFactor.SinFactor(String,String)	1	3	3
expression.factor.SinFactor.diff()	2	1	2
expression.factor.SinFactor.getIndex()	1	1	1
expression.factor.SinFactor.toString()	1	3	3

互测

- 很不幸，本次互测又被找出一个十分脑残的bug：在对表达式进行最终的格式判断时，忘记调用`trim()`去掉首尾空白符，在输入字符串行首有空白符时会导致正则匹配失败，错误输出`Wrong Format!`，而这个问题在前两次作业中都很好地避免了，所以这是十分不应该的一次粗心。
- 更不幸的是，我并没有找到roommates的bug，但同组的两位狼人代码风格却令我十分**惊艳**~~(WTF！还能这么写！？)~~，对异常的使用十分巧妙，且代码的工程性很好，可复用性强。能有幸学习到这样的代码，也算是不幸中的万幸了吧！

自评

- 优点：
  - 本次作业完全抛弃了前两次作业的结构，对整个逻辑框架进行了重构，第一次使用**抽象类**，使得类与类之间的关系与信息交互更加明了，层次清晰，代码可读性提高。
- 缺点：
  - 仍然不会灵活使用**异常处理**
  - 代码的工程性及可复用性仍有所欠缺：程序的很多地方仍有**代码质量问题**，还是有进一步改进的空间的。

总结

对于第二次作业互测被找出的bug，说来也巧，我在查阅阿里Java开发手册时发现有这么一行：

> 13.【推荐】使用索引访问用String的split方法得到的数组时，需做最后一个分隔符后有无内容的检查，否则会抛IndexOutOfBoundsException的风险。
>
> 说明：
>
> ```
> String str = "a,b,c,,";
> String[] ary = str.split(",");
> // 预期大于3，结果是3
> System.out.println(ary.length);
> ```

原来，自己可能会犯的错误已经有无数先烈踩过坑了，如果我能多花时间仔细钻研一下优秀的代码可能就能发现其中的玄机，从而最大可能地避免这种共性的错误。

总之，三周下来，确实能感到自己的能力在稳步提升，但还是没有完全从**面向过程编程**的思想中转变过来。我意识到，想要入门**面向对象**，仅仅完成作业是远远不够的，还要认真阅读相关专业书籍，仔细钻研优秀代码的框架与逻辑，这些都是要花大量时间的。本周就要进入OO第二单元——**线程安全**，学长学姐说这部分的内容对新手非常**不友好**，这更是需要我投入更多的时间与精力，我必须摆正自己的态度，认真对待。

在这里立个Flag，在以后的工程中，我要尽力做到：

- 仔细研读指导书，充分明确需求，在动工之前仔细研究框架的合理性和可扩展性。
- 通过阅读专业书籍以及模范代码，学习更多更巧妙的设计思想，切实地从面向过程的思维中跳出来。
- 严格遵循代码规范，培养良好的编码风格，不能每次都依靠CheckStyle信息去调整。