

**2019年春季学期  
计算机学院《软件构造》课程**

**Lab 2实验报告**

|  |  |
| --- | --- |
| 姓名 | 沈子鸣 |
| 学号 | 1170301007 |
| 班号 | 1703010 |
| 电子邮件 | [2508754153@qq.com](mailto:2508754153@qq.com) |
| 手机号码 | 18800421860 |

**目录**

[1 实验目标概述 1](#_Toc5568559)

[2 实验环境配置 1](#_Toc5568560)

[3 实验过程 2](#_Toc5568561)

[3.1 Poetic Walks 2](#_Toc5568562)

[3.1.1 Get the code and prepare Git repository 2](#_Toc5568563)

[3.1.2 Problem 1: Test Graph <String> 3](#_Toc5568564)

[3.1.3 Problem 2: Implement Graph <String> 6](#_Toc5568565)

[3.1.3.1 Implement ConcreteEdgesGraph 6](#_Toc5568566)

[3.1.3.2 Implement ConcreteVerticesGraph 8](#_Toc5568567)

[3.1.4 Problem 3: Implement generic Graph<L> 10](#_Toc5568568)

[3.1.4.1 Make the implementations generic 10](#_Toc5568569)

[3.1.4.2 Implement Graph.empty() 11](#_Toc5568570)

[3.1.5 Problem 4: Poetic walks 11](#_Toc5568571)

[3.1.5.1 Test GraphPoet 11](#_Toc5568572)

[3.1.5.2 Implement GraphPoet 12](#_Toc5568573)

[3.1.5.3 Graph poetry slam 13](#_Toc5568574)

[3.1.6 Before you’re done 13](#_Toc5568575)

[3.2 Re-implement the Social Network in Lab1 15](#_Toc5568576)

[3.2.1 FriendshipGraph类 16](#_Toc5568577)

[3.2.2 Person类 17](#_Toc5568578)

[3.2.3 客户端main() 17](#_Toc5568579)

[3.2.4 测试用例 17](#_Toc5568580)

[3.2.5 提交至Git仓库 18](#_Toc5568581)

[3.3 Playing Chess 19](#_Toc5568582)

[3.3.1 ADT设计/实现方案 19](#_Toc5568583)

[3.3.1.1 接口 19](#_Toc5568584)

[3.3.1.2 类 20](#_Toc5568585)

[3.3.1.3 ADT关系说明 27](#_Toc5568586)

[3.3.1.4 几个主要ADT的实现思想 28](#_Toc5568587)

[3.3.2 主程序MyChessAndGoGame设计/实现方案 32](#_Toc5568588)

[3.3.2.1 开始新游戏 32](#_Toc5568589)

[3.3.2.2 游戏操作 33](#_Toc5568590)

[3.3.2.3 辅助方法 43](#_Toc5568591)

[3.3.3 ADT和主程序的测试方案 44](#_Toc5568592)

[3.3.3.1 ADT test 44](#_Toc5568593)

[3.3.3.2 应用功能测试 46](#_Toc5568594)

[3.4 Multi-Startup Set (MIT) 49](#_Toc5568595)

[4 实验进度记录 49](#_Toc5568596)

[5 实验过程中遇到的困难与解决途径 50](#_Toc5568597)

[6 实验过程中收获的经验、教训、感想 51](#_Toc5568598)

[6.1 实验过程中收获的经验和教训 51](#_Toc5568599)

[6.2 针对以下方面的感受 51](#_Toc5568600)

# 实验目标概述

本次实验训练抽象数据类型（ADT）的设计、规约、测试，并使用面向对象编程（OOP）技术实现ADT。具体来说：

* 针对给定的应用问题，从问题描述中识别所需的ADT；
* 设计ADT规约（pre-condition、post-condition）并评估规约的质量；
* 根据ADT的规约设计测试用例；
* ADT的泛型化；
* 根据规约设计ADT的多种不同实现；针对每种实现，设计其表示 （representation）、表示不变性（rep invariant）、抽象过程（abstraction function）
* 使用 OOP实现ADT，并判定表示不变性是否违反、各实现是否存在表 示泄露（rep exposure）；
* 测试ADT的实现并评估测试覆盖度；
* 使用ADT及其实现，为应用问题开发程序；
* 在测试代码中，能够写出testing strategy并据此设计测试用例

# 实验环境配置

简要陈述你配置本次实验所需环境的过程，必要时可以给出屏幕截图。

特别是要记录配置过程中遇到的问题和困难，以及如何解决的。

在这里给出你的GitHub Lab2仓库的URL地址（Lab2-学号）。

本次实验环境配置与之前相同。

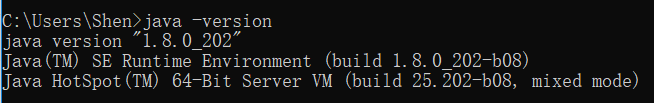
硬件环境：Intel Core i5-7200U，64位; 4G;

软件环境：Windows 10家庭中文版

IDE：Eclipse Java 2018-12

关于Java：JDK 8, JRE 8

关于Git：Git 2.20.1





GitHub仓库的URL地址：

<https://github.com/ComputerScienceHIT/Lab2-1170301007>

# 实验过程

## Poetic Walks

本次实验要求按照MIT 6.031的Problem Set 2的实验手册来完成。其主要目的是训练接口的类的实现、泛型的应用、测试优先的编程习惯以及规约的一些规范，并应用在诗句自动生成场景上。

### Get the code and prepare Git repository

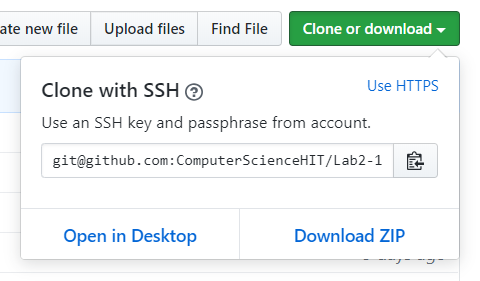
在GitHub上把实验二的相关代码clone到本地。

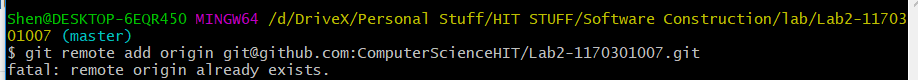


然后建立Eclipse Java项目，并创建git仓库，建立本地Git仓库和远程仓库的连接。



用这个SSH地址用作远程仓库的地址。用origin用作远程仓库的标识符，建立连接。





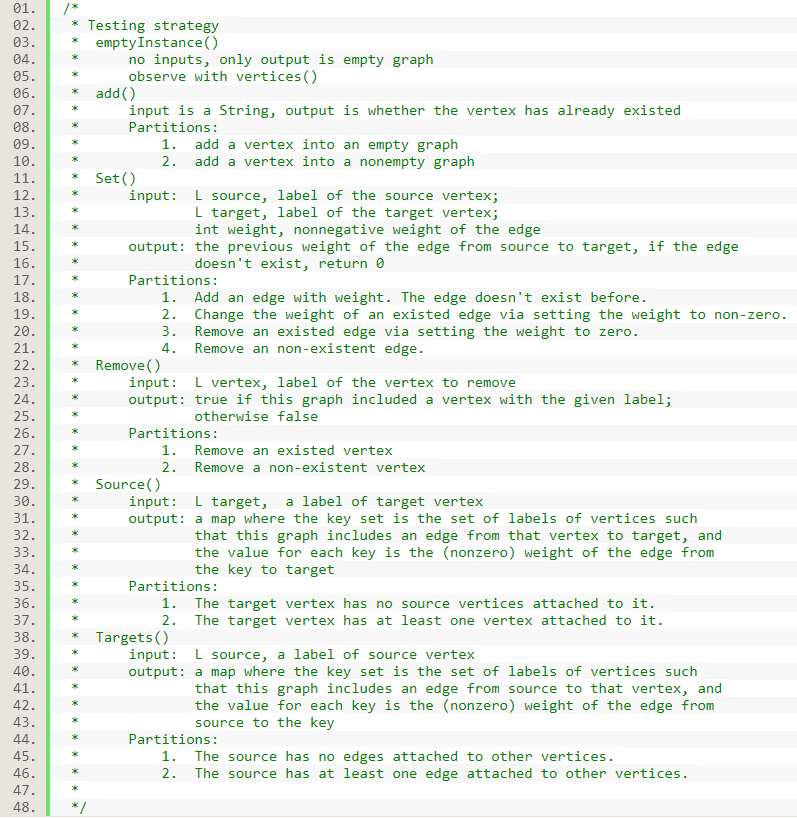
若之前已建立过连接，则会报错，提示远程origin标识符已存在。

### Problem 1: Test Graph <String>

这部分要设计并实现Graph<String>的测试代码，还要完善测试文档。有两个测试文件，一个是GraphStaticTest.java，一个是GraphInstanceTest.java。目前我们主要完善后者，将测试策略和对实例化方法的测试代码都写入GraphInstanceTest.java，在测试过程中，只能使用emptyInstance()方法，而不要去使用静态的Graph.empty()方法，那个是给GraphStaticTest.java用的。

因为GraphStaticTest.java中的测试调用的是Graph接口中的静态工厂方法，所以只要实现了这个方法，该测试就可以运行。但GraphInstanceTest.java中的测试调用的是emptyInstance()抽象方法，这个方法的实例化确切地说是在GraphInstanceTest的子类中，后续会对GraphInstanceTest实现两个子类测试文件，在子类测试文件中完善emptyInstance()抽象方法，到那时我们的GraphInstanceTest.java才能正常运行。

既然是测试优先编程训练，我们就老老实实先写测试代码吧，此时只能严格按照Graph接口的spec来设计测试案例。我的测试策略是这样的（因为我没有找到合适的排版策略，这部分的逻辑最好清晰呈现，就把测试策略的注释截图放过来了）：



我对Graph接口的5个方法分别做了等价类划分，此外还包括一个用来生成实例化Graph对象的emptyInstance()方法测试。

* add()

将一个顶点放入到图中，如果该顶点已存在，则返回false，否则返回true。

对此测试需要分2个等价类。

一类是该顶点不存在；

一类是该顶点已存在。

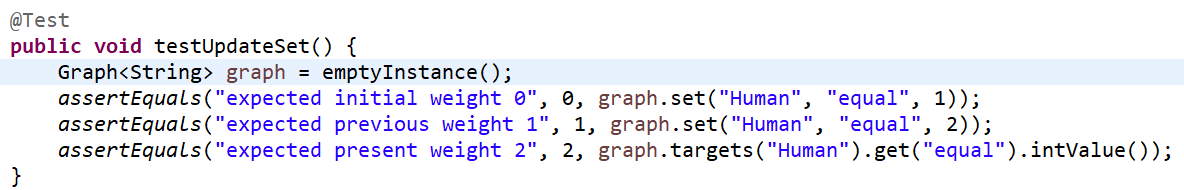
* Set()

要求输入源顶点和目标定点和一个非负整数权值weight，将源顶点到目标定点的边的权值置为weight。输出是该边的旧权值，如果边不存在，则返回0.

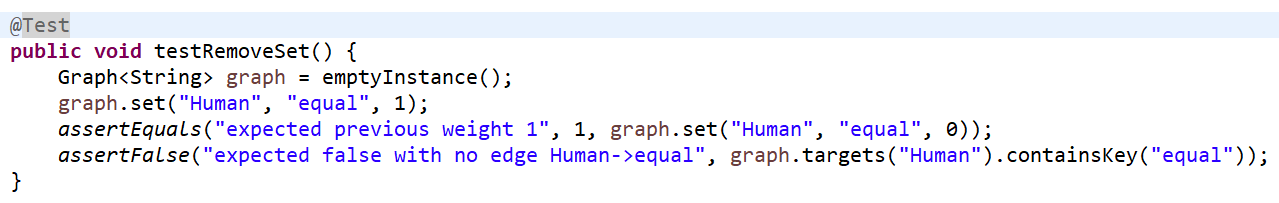
对此测试需要分4个等价类。

一类是边本来不存在，调用此方法后添加了一条权值为weight的边；

一类是边存在，调用此方法后修改了边的权值为正整数weight；



一类是边存在，调用此方法后通过将边的权值设为0，而去除这条边；



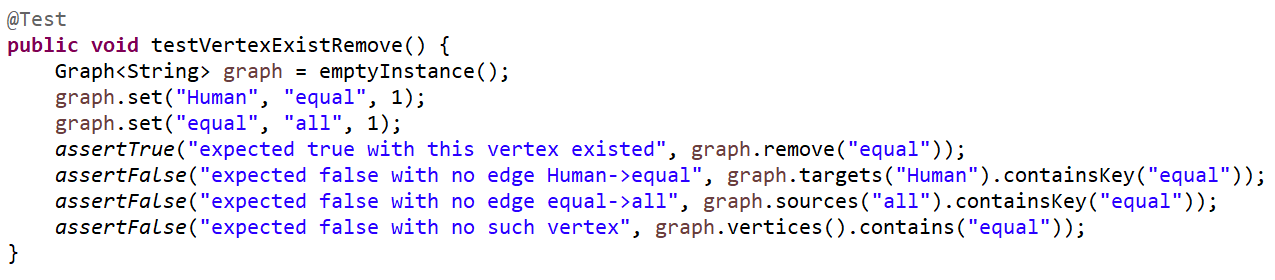
一类是边本来不存在，调用此方法希望通过将边权值设为0而去除它，但失败。

* Remove()

要求输入一个顶点，将该顶点从图中删除，同时删除邻接到它的所有边以及邻接于它的所有边。如果删除成功，即图中原来有这个顶点，那么返回true，否则返回false。

对此测试需要分2个等价类。

一类是顶点存在，测试返回值的同时，还要测试与该顶点相关的邻接边是否删除；



一类是顶点不存在。

* Source()

输入一个顶点，要求返回一个映射表，表中是图中所有邻接到该顶点的顶点到该边的权值的一个映射，权值应该为正整数。

对此测试需要分为2个等价类。

一类是无任何顶点邻接到该顶点；

一类是至少有一个顶点邻接到该顶点。

* Targets()

输入一个顶点，要求返回一个映射表，表中是图中所有邻接于该顶点的顶点到该边的权值的一个映射，权值应该为正整数。

对此测试需要分为2个等价类。

一类是无任何顶点邻接于该顶点；

一类是至少有一个顶点邻接于该顶点。

### Problem 2: Implement Graph <String>

这部分开始实现Graph接口了，暂时先把泛型定位String来实现，这样我们能更容易接受一点，后期再改成泛型。

从这里开始，以后的所有类都要有统一的规约规范。要有抽象过程和表示不变性，用注释的方式给出防止表示泄露的方法以证清白。还要实现checkRep方法来检查表示不变性确实在调用类方法时没有发生改变，还有实现toString方法，用来把抽象表示转换成可读性好的表示形式输出。

所有的类都必须要清晰的文档规约，除了重写的方法外，所有的方法都应该写上Java文档。重写的方法应该默认认为其规约与父类或结构中的规约相同。在这部分中，ConcreteEdgesGraph和ConcreteVerticesGraph都要实现。

#### Implement ConcreteEdgesGraph

按照要求，仅使用给定的表示，对于Edge类保证它是immutable的类，对任何重写的方法，记得加上@override标识符。

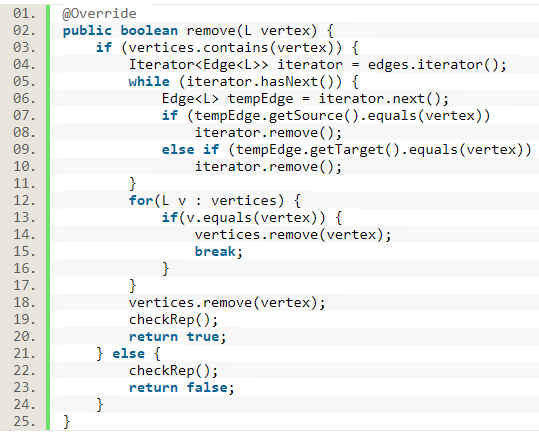
剩下的就是关于如何实现接口中的方法了。大体思想是针对测试中划分的等价类，用条件语句来对每种情况进行划分。该类中的仅能使用的两个rep是Set类型的顶点集合，和List类型的边列表，列表中存储的是Edge类型的边。这样我们可以使用顶点集和边表，对各个操作的实现来说，比较灵活方便。

public boolean add(L vertex)，向图中添加一个顶点，只有两种情况，已存在该顶点，返回false，该顶点是新加入的顶点，返回true。

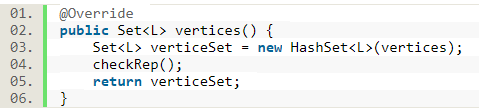
public int set(L source, L target, int weight)，根据等价类，有四种情况。首先以weight参数是否为0，分为两类，然后再根据修改权值的类型，划分两类，有可能是返回之前的旧权值，也有可能之前该边不存在，从而返回0.这个方法的代买截图如下：



public boolean remove(L vertex)，删除一个顶点即可，别忘了把关于该顶点的所有边也要从边表中抹去。



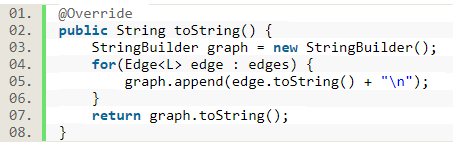
public Set<L> vertices()，这个方法是要返回顶点集合，但为了保证表示不被泄露，需要做一份防御式拷贝，再返回。



public Map<L, Integer> sources(L target)，这个方法要求返回一个映射表，表中是图中所有邻接到该顶点的顶点到该边的权值的一个映射，权值应该为正整数。对此还是在边表中遍历，抽取出符合要求的顶点，将其和对应权值加入到映射表。

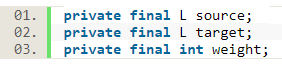
public Map<L, Integer> targets(L source)，这个方法要求返回一个映射表，表中是图中所有邻接于该顶点的顶点到该边的权值的一个映射，权值应该为正整数。实现思想和sources方法完全相同。

public String toString()，这个重写了Object父类的方法，这里决定让toString把图输出出来，就是把每个边都打印一边，需要调用Edge类中的toString方法。



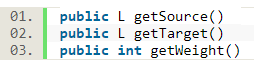
还有Edge类的实现方法。这个类要求必须是immutable的类，为了保证它是immutable的，所有的方法都最好修饰为private和final，表示必须都是immutable类型，不能有mutator方法。而之前的set()方法可能需要删除或更改某条边，既然我们不能更改边的属性，那对于更改操作，就把边删除，再重新生成一条边吧。

Edge类中要有三个rep，边的起点、终点和边的权值：



Edge类中只有Constructor方法和Observer方法，还有一个检查表示不变量的checkRep()，和一个重写父类的toString()。

三个Observer方法为：



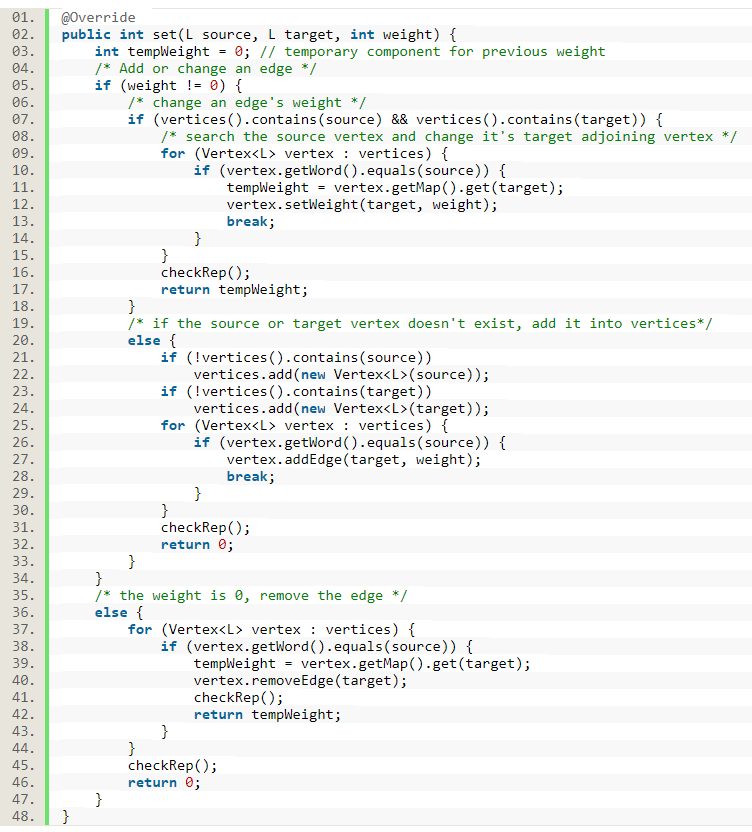
#### Implement ConcreteVerticesGraph

这部分要保证Vertex类是mutable的，mutable的类虽然不被看好使用，但限定条件较少，相比之前的immutable的Edge类，Vertex中可以添加mutator方法了。还需要注意的是，本部分给提供的rep只有一个顶点列表，表中存储的是Vertex类型的顶点，注意，没有给提供边集的表示。除此之外，该部分的实现过程跟3.1.3.1的差不多。

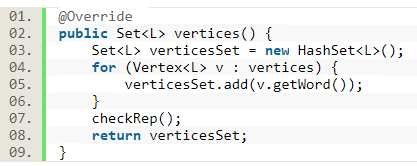
如果没有边集的表示的话，很多地方的实现和Edge Graph不太一样。比如在set方法中，对一个边的权值修改，因为没有边集表示，所以不能直接获取边并修改它，我的方法是在Vertex类中，对每一个顶点赋给一个邻接映射表属性



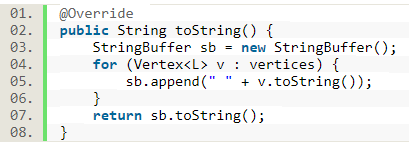
这个邻接映射表存储了邻接到该顶点的所有顶点和其对应边的权值。在搜索边的时候，只需要对相关顶点的邻接映射表遍历就好了。这段代码实现要比有边集表示的复杂一些，具体实现如下：



当然了，其它方法涉及到搜索特定边的，也是需要遍历顶点的邻接映射表来做到的。还有vertices方法，要返回一个顶点集合。可是顶点表示是一个列表类型，那就遍历一次，把顶点加入到一个集合中返回就好了。



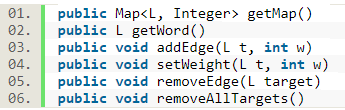
还有toString方法，也是希望能把这个图的相关信息打印出来，这里就只打印顶点。需要调用了顶点的toString方法。



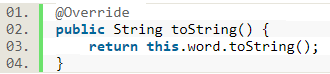
关于Vertex的具体实现，因为被要求是mutable的类，所以限制条件少一些。不过还是要注意表示泄露的问题。两个rep分别表示顶点的标签和邻接映射表。



除了一个Constructor方法，一个checkRep检查表示不变性的方法，还有两个Observer方法和四个Mutator方法。其中Observer方法和Mutator方法如下：



在4个Mutator方法中，addEdge，setWeight和removeEdge是对邻接映射表的操作，removeAllTarget也是对邻接映射表的操作，只不过这个方法是为了方便设计的，它清空该顶点的邻接映射表。toString方法如下：



### Problem 3: Implement generic Graph<L>

之前的实现都是基于类型String的，接下来我们要引入泛型，使我们的接口和类，使其不再依赖于类型String，而是可以扩展到所有的对象类型（默认规定我们的依赖类型是immutable）。

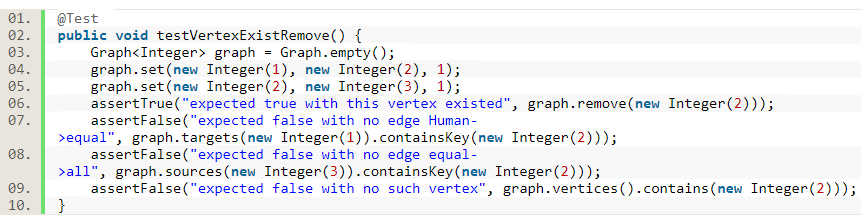
#### Make the implementations generic

先把所有的String类型，改为泛型L。这其中包括所有的构造器输入类型和列表或集合或映射表中的String类型都要改成泛型L。这些做完之后，再运行一遍测试代码，确保没有错误出现。否则，还得debug。

#### Implement Graph.empty()

Graph.empty()是在Graph接口中的一个静态工厂方法，用于用户使用时初始化一个Graph对象，对象类型取决于empty方法的实现。这里我们可以挑选Graph接口的两个实现子类中的任一个类来实现这个empty方法。我用的是ConcreteEdgesGraph类来实现的。

到这里就可以返回去完成GraphStaticTest的代码了。这里面的测试是对Graph的实现类的实现测试，看看我们写好的代码能否适应泛型的变化。之前都是用的String类型类测试泛型（在GraphInstanceTest中），现在我们用Integer(immutable 类型)再试一次。其中一个测试方法示例如下：



### Problem 4: Poetic walks

Graphs — what are they good for? Poetry!

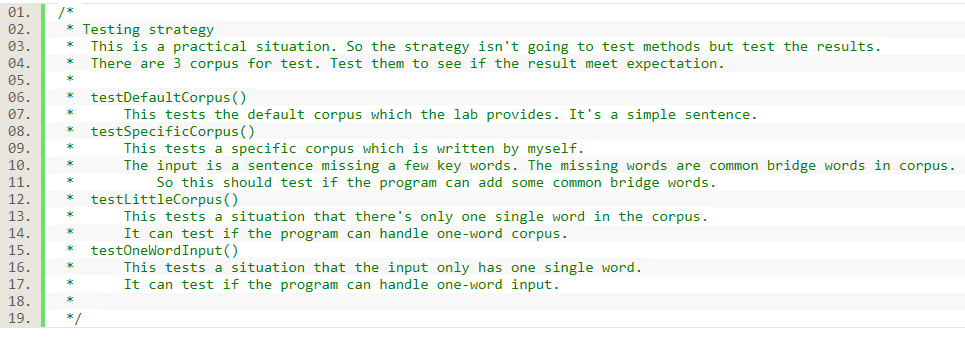
终于要把写的代码交付自己使用了！

这部分要求用自己设计的ADT来实现一个英语诗句补全的功能。其实这个功能也不算很实用，甚至有点无用。关于应用的功能要求在手册中写的很详细了，大概就是给定一个语料库corpus，根据corpus中的文本生成一个单词图，然后给定一条语句输入，在图中搜索词之间的关系，自动补全语句中可能可以完善的部分。

图的构建规则是，在corpus中，对每一个不一样的单词看作一个顶点，相邻的单词之间，建立一条有向边，相邻单词对出现的次数，作为这条有向边的权值。在输入信息补全时，对相邻单词A和B做检查，如果存在一个单词C，在图中可以由前一个单词A通过这个单词C到达单词B，那么就在A和B之间补全C，补全的优先级按照权值越大者优先。

#### Test GraphPoet

还是测试优先的编程规范。我的测试策略如下：



因为这是一个实际应用场景，所以测试的话不打算对每个方法测试，而是对不同情况的输入测试它的结果。在这里我测试了四种情况。

第一种是实验手册中给出的默认corpus，测试一条预计会补全单词的输入信息。

第二种是测试一个特定的corpus，其中包含了大量的重复出现的词组，主要是为了测试补全时的优先级。输入信息也是一条预计会有补全的信息。

第三种是极端情况，是corpus中只有一个单词，是为了测试单词关系图构造是否能成功。

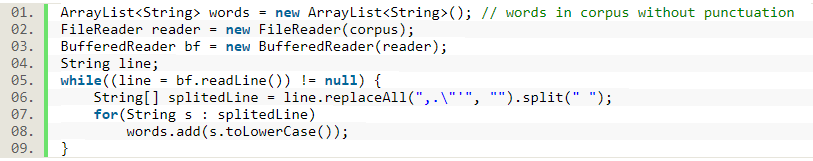
第四种是极端情况，是输入信息只有一条信息，是为了测试输入处理时是否能够接受这种输入。

四种情况都是给定corpus和输入信息，将输出信息和预想的信息对比。

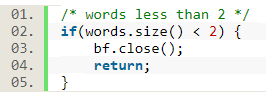
#### Implement GraphPoet

GraphPoet类中只有两个方法，一个是构造方法，输入参数是一个给定的corpus文件，用来构造关系图graph。一个是生成诗的poem方法，输入一条String输入信息，输出一条String补全信息。

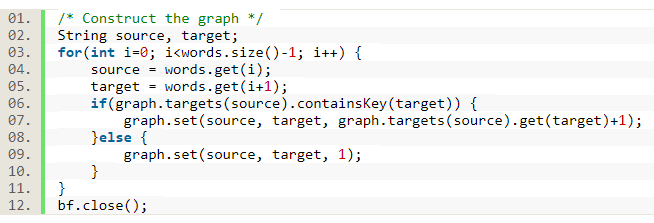
对于构造方法，因为我们的rep表示是Graph<String>的一个实例化（用的是ConcreteEdgesGraph<String>），所以具体操作都是引用这个实例化类中的方法。首先对corpus文件进行读取并做单词提取处理，提取出一个String列表words，其中包含了corpus中所有的剔除了标点符号的单词。



然后，如果corpus中单词少于2，直接退出。

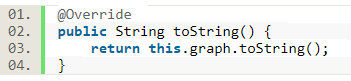


构造graph的环节，只要调用Graph中的实例方法就好了。



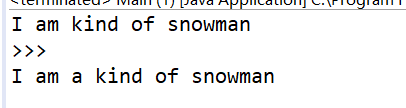
打开文件的时候，可能会发生异常，用try-catch处理一下就好了。

还有一个重写的toString方法，直接调用Graph的实例子类的toString方法。



#### Graph poetry slam

这部分在Main.java的main函数里面实际应用一下就好了。我找了一个有比较多的语句的corpus，把一些中文诗句用谷歌翻译，作为corpus。然后随便写一句可能有补全的话，结果如下：

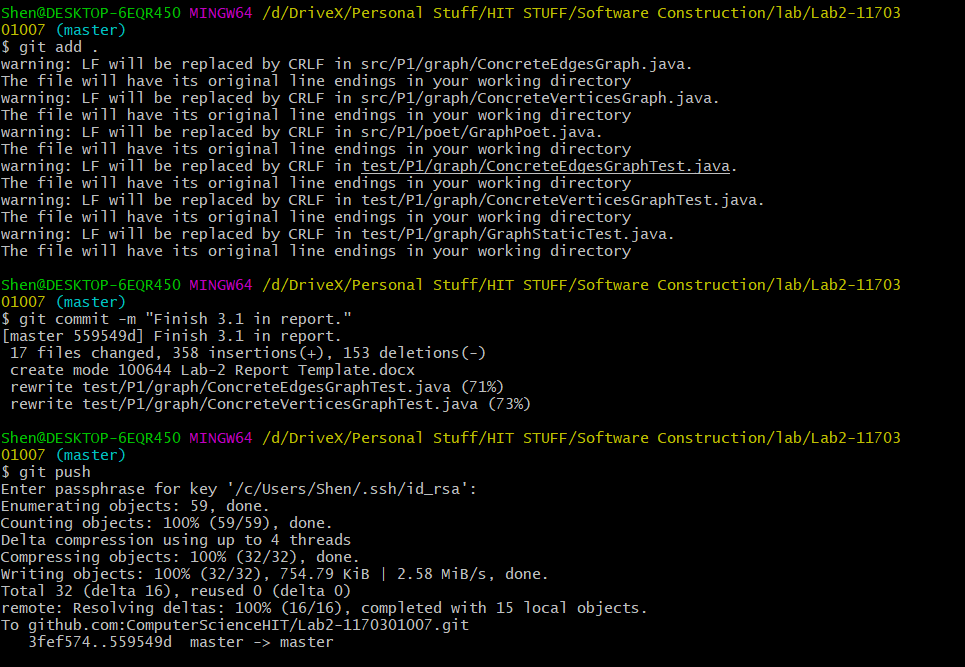


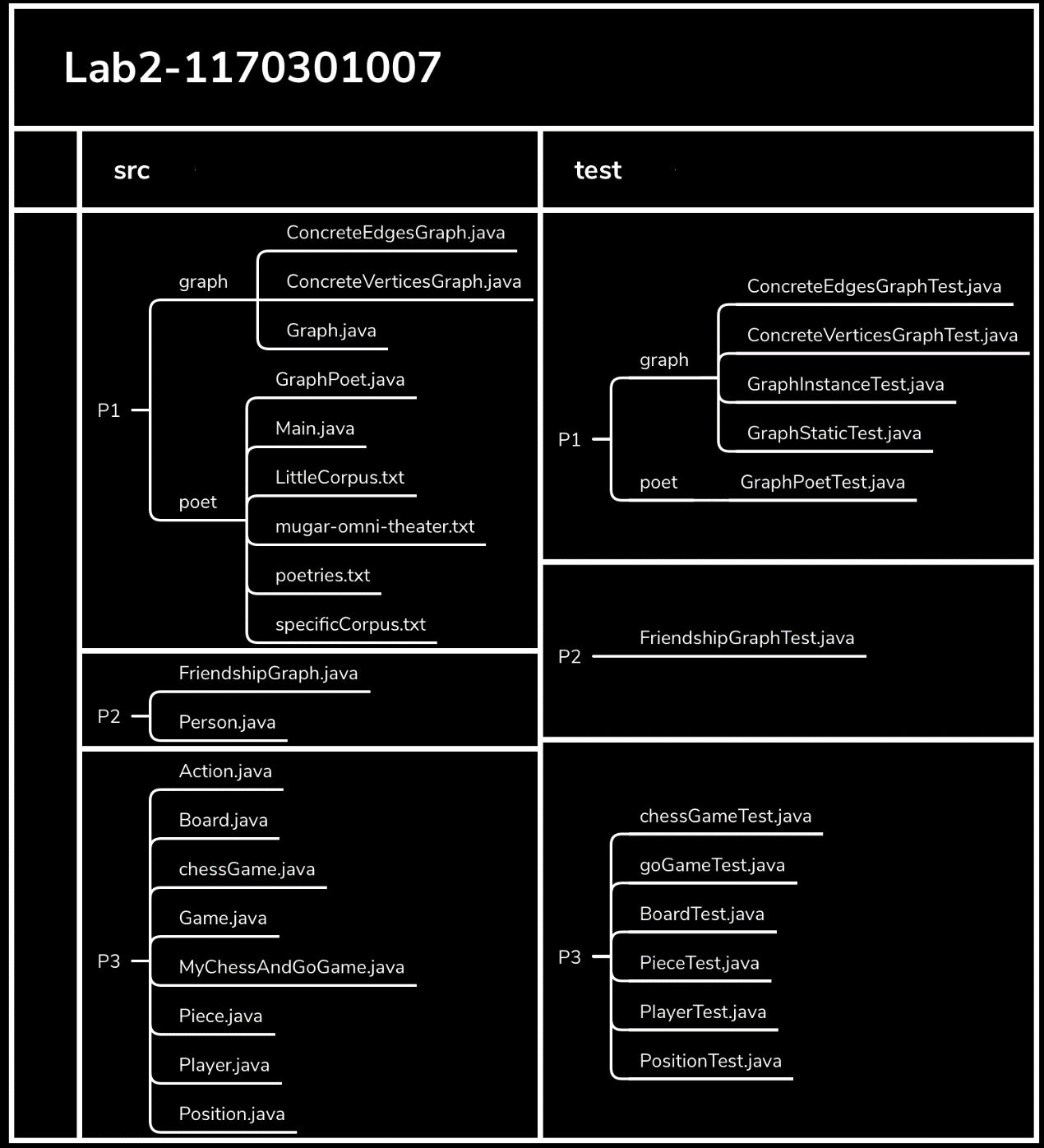
事实上，因为corpus中的信息对于人来说，检索起来并不直观，所以我也不知道什么样的语句输入可能会被补全，甚至有可能，单词关系图graph中的所有边的权值都不超过1，所以这个实际应用看起来还是相当不成熟的。

### Before you’re done

提交到GitHub上的命令

* git add .. // 将当前目录下的所有文件加入到缓冲区
* git commit -m *message* // 将缓冲区中所有文件上传到本地仓库，并以message作为本次提交的记录
* git push // 将本地仓库中的文件推送到远程仓库





## Re-implement the Social Network in Lab1

这部分任务就是用我们在3.1中写的ADT，把第一次实验中的FriendshipGraph重新实现一遍，图中的节点仍然是Person类型，所以泛型L一律为Person. 而对于我们已经写好的FriendshipGraph中的方法，我们要推倒重来，用3.1中的Graph ADT中的方法来实现它们。

最后完成之后，在main()客户端代码不变的情况下，理应正常运行，否则，可能需要debug。

### FriendshipGraph类

最开始采用常规方法，在FriendshipGraph类中设计一个ConcreteEdgesGraph类的rep，通过这个rep表示，来对图进行操作，后来在测试中发现，如果这么设计的话，则必然要暴露这个rep的表示，于是最后采用老师给建议的方法，零FriendshipGraph类继承ConcreteEdgesGraph类，从而它的rep表示，抽象过程和表示不变性以及检查，都继承ConcreteEdgesGrpah类，而表示泄露的问题，也是根本不会发生。接下来就设计实现如何利用ConcreteEdgesGraph ADT来实现FriendshipGraph中的三个方法的功能了。

public void addVertex(Person person) throws Exception，向图中添加一个顶点person。要注意虽然FriendshipGraph继承了ConcreteEdgesGraph，但是后者的rep还是不能直接调用（private），对于FriendshipGraph来说是不可见的，所以添加顶点还是得调用Mutator方法。在添加顶点时，要检查这个人是否已经存在了（以忽略大小写的人名作为评判标准），如果存在的话，就抛出一个异常，报出异常信息，直接终止程序。否则，就正常调用add方法来把这个人加入到顶点集中。

public void addEdge(Person p1, Person p2)，向图中添加一条p1到p2的边，默认设置其权值为1.这个非常简单了，根据我们ConcreteEdgesGraph的设计，即时这条边的两个顶点未在图中，也是默认先加入这两个顶点，再连接这条边。直接调用set方法即可。

public int getDistance(Person p1, Person p2)，这个方法是得到p1到p2的最短距离，算法用的是广度优先搜索，在广度优先搜索的过程中，用一个映射表来记录搜索过程顶点之间的前置后置关系。最后，根据前置后置映射表，从p2回溯到达p1，回溯所经过的顶点数，就是路径长。



### Person类

Person类也要注意表示泄露的问题。不过我们还是先来确定一下它的rep表示。每个人都有名字属性String类型，在广度搜索时还要判断这个人是否被搜索过，所以给加一个boolean类型的marked标志，如果为真，则被遍历过，否则未被遍历过。

构造方法中，用给定String参数来初始化名字，而marked表示则初始化为false。

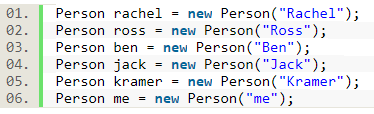
还有三个实例方法，两个Observer，分别获取rep信息，注意表示泄露的问题。还有一个Mutator，是用来更改marked表示的。

### 客户端main()

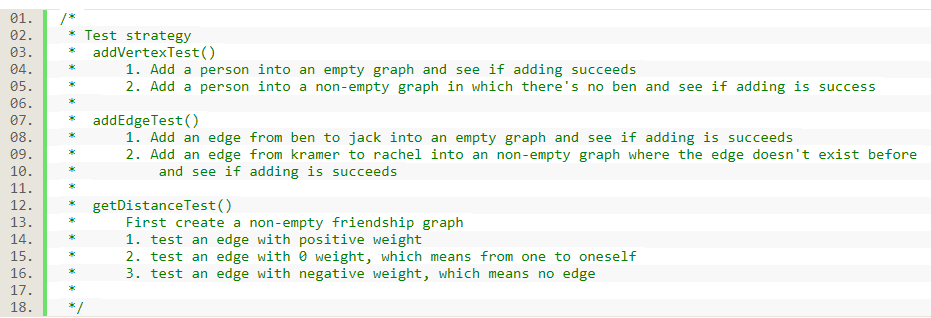
main()客户端方法，原封不动地拿过来，运行成功了，测试也通过了，那就可以了。此外还可以对一些比较复杂的图，双向边单向边混杂，路径有长有短的图来测试。这些其它测试我放在了测试用例中，见3.2.4. 本部分代码我是直接把实验一种的客户端代码拿过来了。

### 测试用例

这部分测试都需要用到一些Person对象，在类的rep中给出。



再给出我的测试策略，



对三个实例方法分别测试，分别划分等价类。对于前两个方法，是属于功能性测试，只要简单的划分等价类，并检查执行结果就可以了。而对于getDistanceTest，因为它测试的方法是一个应用性的方法，所以不太好划分等价类，那就构造一个包含尽可能多情况的特殊设计的图，用运行结果来测试。

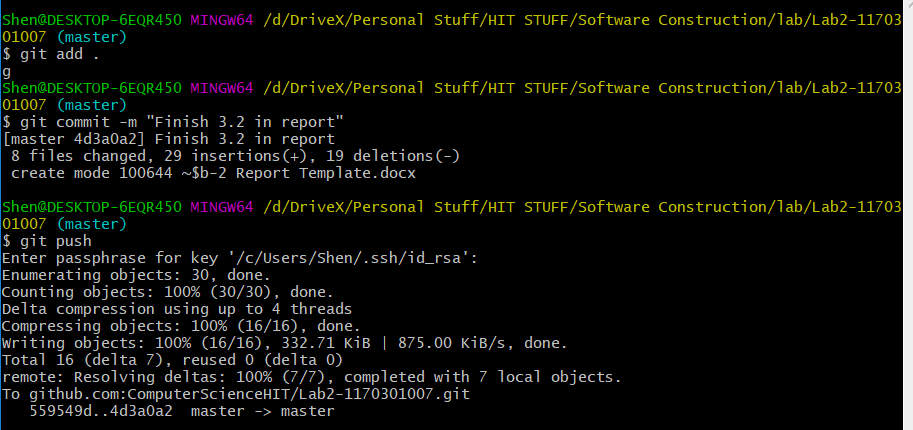
给出getDistanceTest的代码：



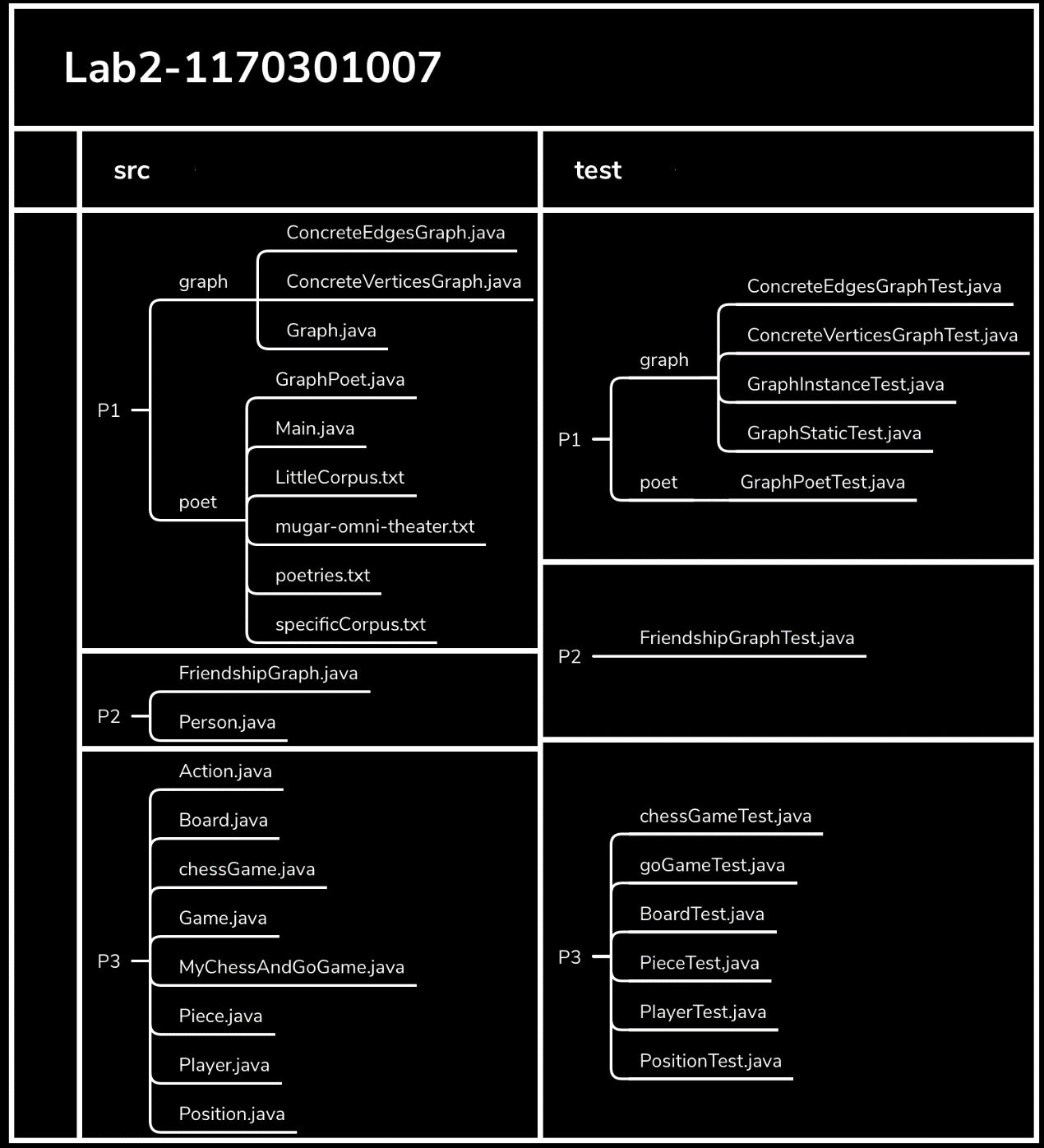
可以看出这里面有几种情况，二者之间有向路径的（-1），到自身的路径的（0），至少有两条路径且最短路径距离相同的（rachel->kramer, 3），还有至少有两条路径但有唯一最短路径的（ross->kramer, 1）。

### 提交至Git仓库

命令如3.1.6所示，上截图。



在这里给出你的项目的目录结构树状示意图。



## Playing Chess

### ADT设计/实现方案

其实如果不想看长篇累牍的叙述的表格展示的话，可以直接跳到3.3.1.3节。

#### 接口

|  |  |  |
| --- | --- | --- |
| 接口interface | | |
| 接口名 | 内部方法 | 方法简要叙述 |
| Action | public boolean move(Player player, Position source, Position target) | 若source不为空，则为国际象棋的移动操作；若source为空，则为围棋的移动操作。在国际象棋操作中，玩家player将自己在source位置的棋子，移动到target位置。在围棋操作中，玩家player在target位置下一枚己方棋子。  方法返回true，如果移动成功；否则返回false。 |
| public boolean kill(Player player, Position source, Position target) | 若source不为空，则为国际象棋的移动操作；若source为空，则为围棋的移动操作。在国际象棋操作中，玩家player用自己在source位置的棋子，杀死target位置上的对手棋子。在围棋操作中，玩家player提走target位置上的一枚对手棋子。  方法返回true，如果移动成功；否则返回false。 |
| Game extends Action | public boolean checkPosition(Position position) | 检查position位置上的棋子。  方法返回true，若position合法；否则返回false |
| public Board getBoard() | 获取一份当前游戏的棋盘拷贝 |
| public Player getPlayer1() | 获取当前游戏player1的玩家拷贝 |
| public Player getPlayer2() | 获取当前游戏player2的玩家拷贝 |
| public Player getOpponent(String name) | 获取当前游戏中，姓名为name的玩家的对手拷贝 |
| 该接口中继承了Action接口。 | |

#### 类

|  |  |  |  |
| --- | --- | --- | --- |
| 类Class | | | |
| 类名 | Rep | 内部方法 | 方法简要说明 |
| Board | private final Piece[][] board | public Board(int size) | Constructor.有一个整型输入参数size，构造一个大小为size\*size的棋盘，横纵坐标合法区域为[1,size]。 |
| public Piece[][] getBoard() | Observer.获得一份board的防御性拷贝 |
| public void setBoard(Position position, Piece piece) | Mutator.对棋盘上的position位置，设置为棋子piece |
| MUTABLE CLASS：该类中有Mutator方法  Abstract function：  AF(board) = a game board  Representation invariant:  Board应该是9\*9或者20\*20的二维数组，每个方法中都执行了checkRep  Safety from rep exposure：  唯一的表示已经被定义为private和final，虽然表示board是mutable类型，但对于它的Observer都已经做了防御式拷贝。 | | | |
| 类名 | Rep | 内部方法 | 方法简要说明 |
| Piece | private final String type | public Piece(String type, String color) | Constructor.构造一个Piece对象，并以type作为表示type的初始化，以color作为表示color的初始化。 |
| private final String color | public String getType() | Observer.返回表示type。 |
| public String getColor() | Observer.返回表示color。 |
| IMMUTABLE CLASS：该类中无Mutator方法，所有成员private且final，任何Observer都避免了表示泄露。  Abstract function：  AF(type, color) = a color's(player's) piece, the piece's type is type  Representation invariant:  Type应该是"pawn1", "pawn2", "pawn3", "pawn4", "pawn5", "pawn6", "pawn7", "pawn8", "rook1", "rook2", "knight1", "knight2", "bishop1", "bishop2", "king", "queen" 或 null其中之一  Safety from rep exposure：  所有的表示都被修饰为private和final；所有的表示类型都是immutable类型。 | | | |
| 类名 | Rep | 内部方法 | 方法简要说明 |
| Player | private final String name | public Player(String name) | Constructor.构造一个Player对象，以参数name初始化表示name |
| private final Set<Piece> pieces = new HashSet<Piece>() | public String getName() | Observer. 返回表示name |
| private final ArrayList<String> history = new ArrayList<String>() | public Set<Piece> getPieces() | Observer. 返回一份表示pieces的拷贝 |
| public void addPiece(Piece piece) | Mutator. 向pieces中添加棋子piece |
| public void removePiece(Piece piece) | Mutator. 从pieces中移除棋子piece |
| public ArrayList<String> getHistory() | Observer. 获取一份表示history的拷贝 |
| public void addHistory(String message) | Mutator. 向表示history中添加一条消息message |
| MUTABLE CLASS：该类中有Mutator方法。  Abstract function：  AF(name, pieces, history) = a player named name who has a set of pieces and his action history  Representation invariant:  表示name不应该为null  Safety from rep exposure：  所有的表示都被修饰为private和final；所有的mutable类型的观察器都做了防御式拷贝。 | | | |
| 类名 | Rep | 内部方法 | 方法简要说明 |
| Position | private final int x, y | public Position(int x, int y) | Constructor. 构造一个Position对象，并以x初始化表示x，y初始化表示y。 |
| @Override  public String toString() | 重写Object父类的toString方法。以(x,y)形式输出Position的表示。 |
| public int getX() | Observer. 返回表示x |
| public int getY() | Observer. 返回表示y |
| IMMUTABLE CLASS：该类中无Mutator方法，所有成员private且final，任何Observer都避免了表示泄露。  Abstract function：  AF(x,y) = a coordinate with position (x,y)  Representation invariant:  表示x和y应该是非负整数  Safety from rep exposure：  所有的表示都被修饰为private和final；所有的表示都是immutable的 | | | |
| 类名 | Rep | 内部方法 | 方法简要说明 |
| chessGame implements Game | private final Board board = new Board(8) | public chessGame(String player1, String player2) | Constructor. 构造一个chessGame对象，以player1作为名字参数初始化表示player1，以player2zuo为名字参数初始化表示player2. |
| private final Player player1, player2 | private boolean commonPartBetweenMoveAndKill(Boolean moveFlag, Player executor, Position source, Position target) | 在国际象棋游戏中，移动棋子和吃子行为是有高度重合的，特此写一个private函数，以作为代码复用，在kill和move函数中调用此函数即可。该方法返回true，如果操作是合法的；否则返回false |
| private boolean outOfBoardWarning(String warning, Position position) | 内部使用的功能方法。判断一个位置是否超出期盼范围。返回true，如果位置position超出棋盘范围，同时打印出期望的警告信息warning；否则，返回false。 |
| private void setPieces() | 内部使用的功能方法。初始化棋盘上的双方棋子。 |
| 重写了Game接口和Action接口中的所有方法 | |
| MUTABLE CLASS：该类中包含了mutable的rep，存在方法操作改变mutable的rep的属性。  Abstract function：  AF(board, player1, player2) = a chess game with two players who are player1 and player2 and a chess board  Representation invariant:  两个玩家不应该有相同的名字（不区分大小写）  Safety from rep exposure：  所有的表示都被修饰为private和final；所有的mutable表示的观察器都做了防御性拷贝 | | | |
| 类名 | Rep | 内部方法 | 方法简要说明 |
| goGame implements Game | private final Board board = new Board(19) | public goGame(String player1, String player2) | Constructor. 构造一个goGame对象，以player1作为名字参数初始化表示player1，以player2zuo为名字参数初始化表示player2. |
| private final Player player1, player2 | private boolean outOfBoardWarning(Position position) | 内部使用的功能方法。判断一个位置是否超出期盼范围。返回true，如果位置position超出棋盘范围；否则返回false。 |
| 重写了Game接口和Action接口中的所有方法 | |
| MUTABLE CLASS：该类中包含了mutable的rep，存在方法操作改变mutable的rep的属性。  Abstract function：  AF(board, player1, player2) = a go game with two players who are player1 and player2 and a go board  Representation invariant:  两个玩家不应该有相同的名字（不区分大小写）  Safety from rep exposure：  所有的表示都被修饰为private和final；所有的mutable表示的观察器都做了防御性拷贝 | | | |

#### ADT关系说明

本部分一共给出了Action、Board、Game、Piece、Player、Position共6个ADT，让我们设计可以满足需求的棋类游戏ADT。因为我们需要设计两个棋类应用，所以我想，把Game作为接口，用chessGame和goGame两个类来应用这个接口，实现具体方法的应用。至于Board，Piece和Position三个类，更像是一种和属性相关的数据类型，其中只有很少的数据操作，尤其是Position，它只是把二维坐标定义成了一个数据类型，把x和y坐标封装了一起，没有操作。还有和Game类似的，Action也是多种棋类游戏中操作高度相似的，所以把Action也定位为接口，并且用Game这个游戏接口去扩展Action接口，这样所有的具体实现Game的类都需要重写Game和Action中的方法。除了Game和Action之外，Player也是实现游戏的一个主要的ADT。我令Player中管理每个玩家的棋子Piece，还有玩家的走棋历史，这样任何和棋子有关的操作，就涉及到了Player。此外，Board上记录着棋子，如果某种操作改变当前局游戏的棋子，那么Player中管理的棋子集合，和Board上的记录，可能是要都发生改变的。

#### 几个主要ADT的实现思想

对棋局有主要影响的操作只有移动（走棋）和吃子（提子）操作，这两种操作涉及了很多其它的ADT。

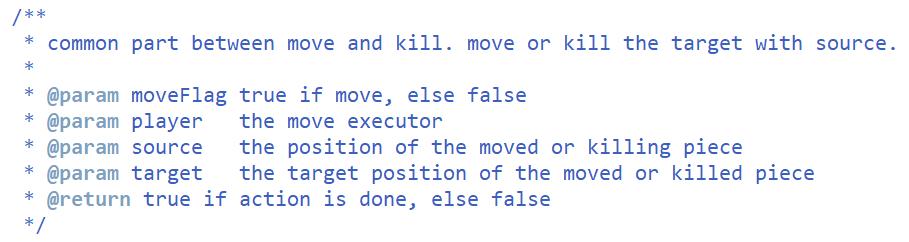
##### commonPartBetweenMoveAndKill

由于国际象棋中的移动和吃子行为高度重合，所以用一个用来复用的函数来实现主要功能。

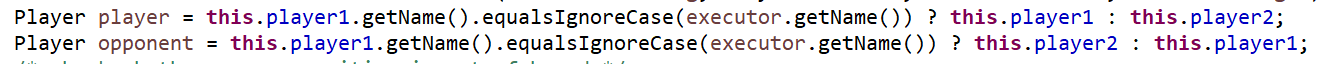
该函数的声明如下：



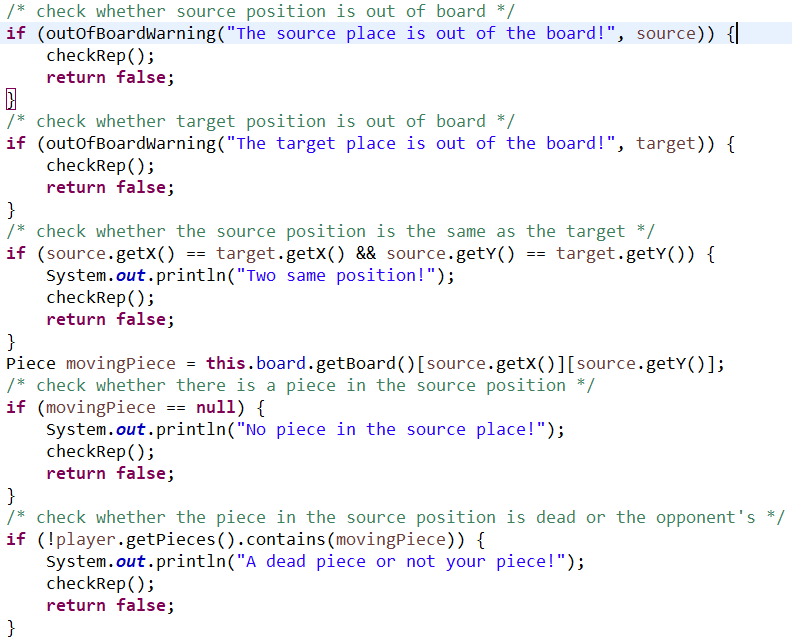
规约spec如下：



关于具体实现，因为有可能涉及吃子操作，所以可能要调用两个玩家变量。而输入参数中只有执行玩家，所以先把ADT内部的执行玩家和对手玩家获取得到。



然后对输入参数做各种合法性检查。



然后开始对kill和move操作分类检查合法性。这期间夹杂了在合法操作时的走棋历史的记录。



##### move(goGame)

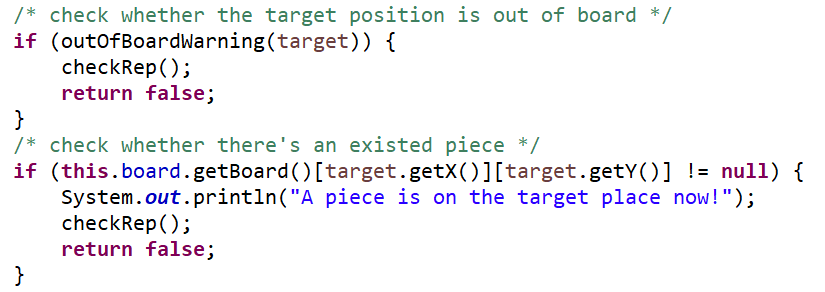
由于围棋中的走棋和提子的行为差的很多，所以就分开写了，不能像国际象棋那样复用同一个方法了。需要注意的是，由于我们重写的是Game接口的方法，而根据国际象棋需求，不得不加上三个参数，而围棋其实只要两个参数就好了。所以move方法中的第二个参数source是null。



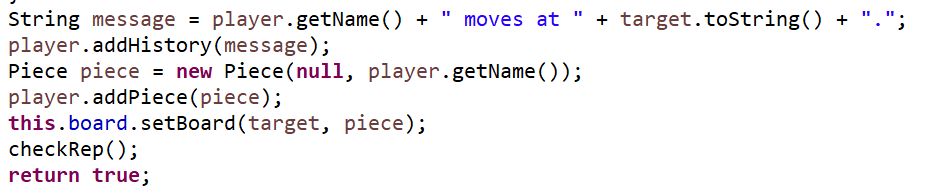
注意参数传进来的玩家player，并不是ADT内部的真正的player，这是我们做防御性拷贝时的后果。所以先获得内部的执行玩家。



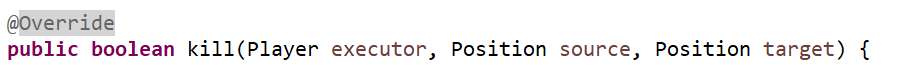
然后是输入参数合法性检查。



在输入都合法的情况下，对棋盘和玩家进行棋子修改，然后记录走棋历史。

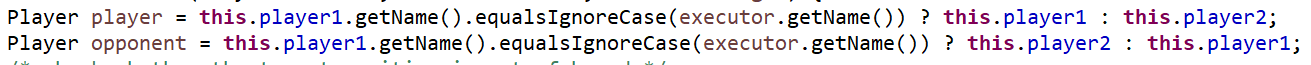


##### kill(goGame)

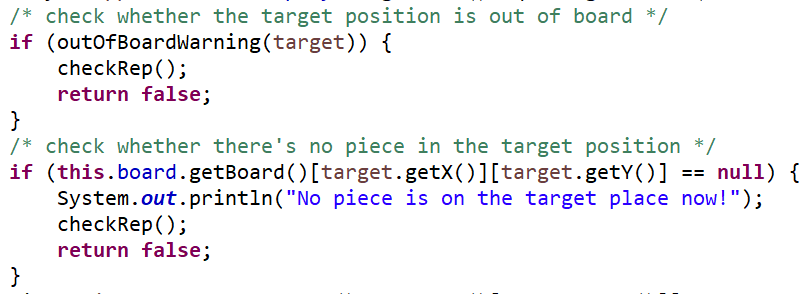


还是，因为是围棋操作，所以这里面的source参数其实是null的。

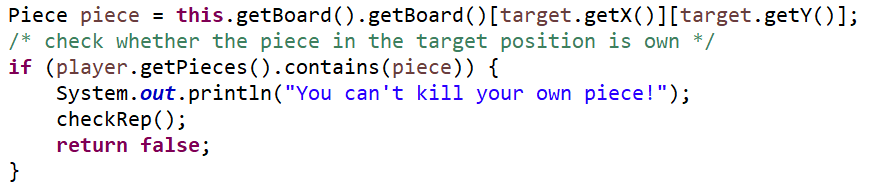
由于防御性拷贝的后果，得先获得内部的执行玩家，和提子相关的，还得获取对手玩家。



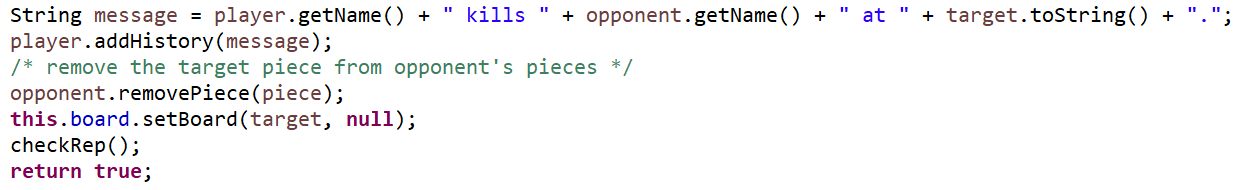
然后是输入参数合法性检查。



然后对kill操作做合法性检查。



所有的输入和位置都合法情况下，对棋盘和玩家修改，然后记录走棋历史。

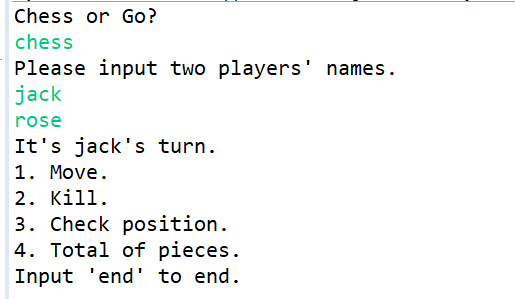


### 主程序MyChessAndGoGame设计/实现方案

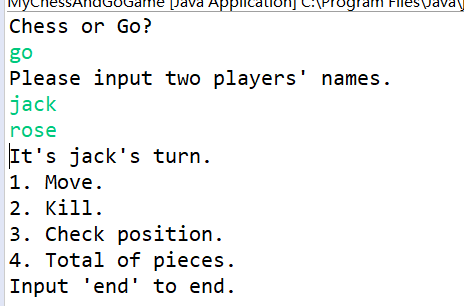
本部分的讲解将从多个功能、异常的处理来说明。

#### 开始新游戏

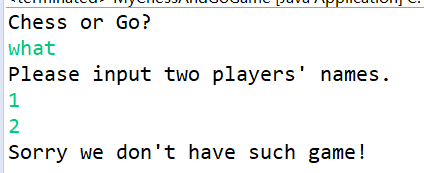
程序运行后，询问client游戏类型，client输入存储在一个String对象中，紧接着要求输入两个玩家姓名，如果根据client输入，用分支语句判断游戏类型，不区分大小写，其它输入会报出错误提示信息并结束运行。



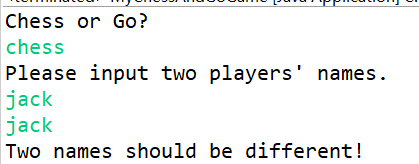
国际象棋游戏



围棋游戏

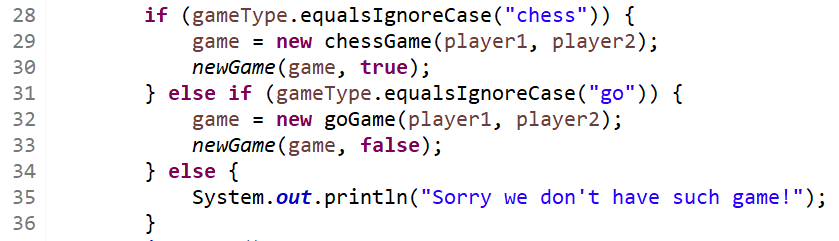


非法游戏



非法姓名

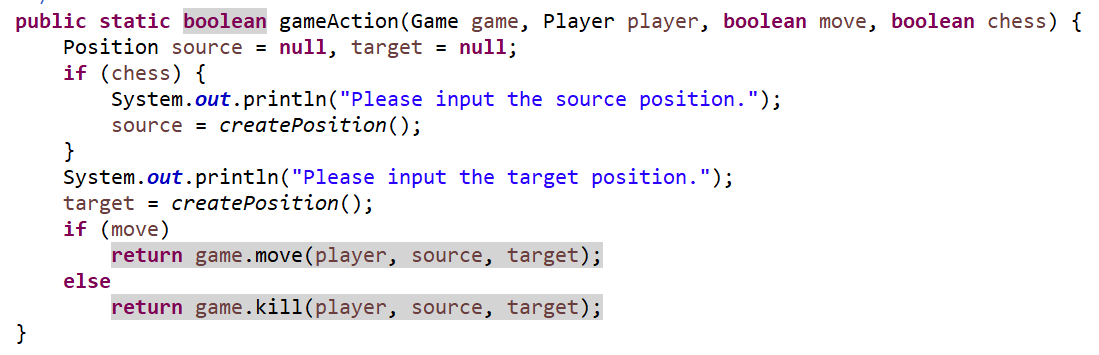
如果所有输入均为合法，将进入正确的分支，并调用newGame构造器，新建一个对应游戏类型的对象，如果是国际象棋，则newGame的第二个参数为true，否则为false。



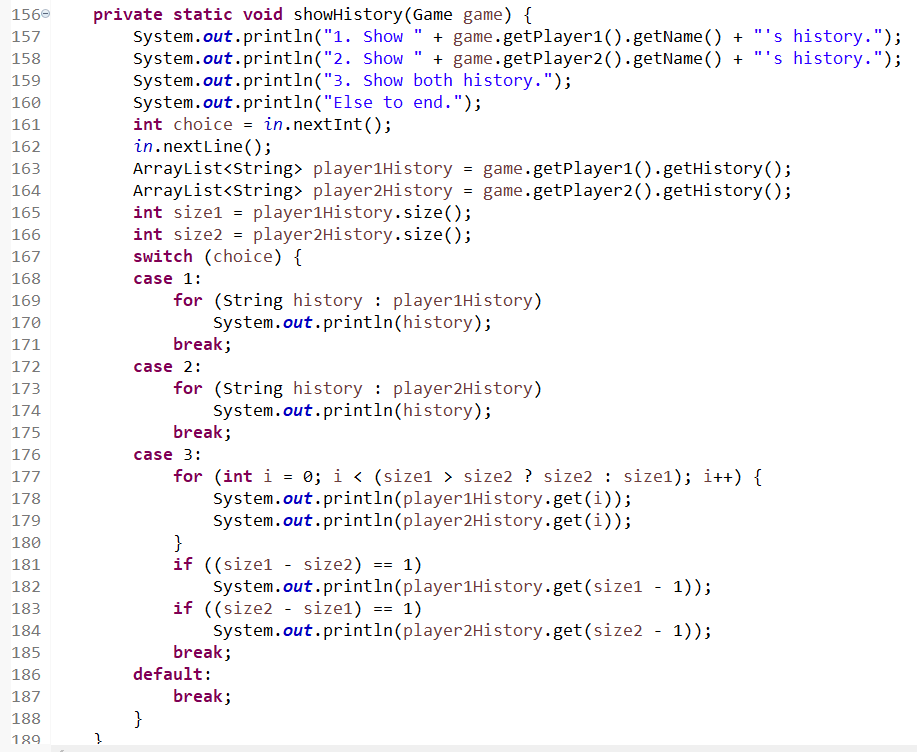
#### 游戏操作

游戏中的移动（走棋）和吃子（提子）操作是调用gameAction来实现的。如果是国际象棋，则该方法调用的第四个参数应该为true，如果是围棋，则该方法调用的第四个参数应该为false。具体操作的选择由client从键盘输入，输入存储在一个String对象中输入合法性由菜单规定。当前执行玩家存储在turn变量中，执行相应操作结束后，turn会更改。

移动（走棋）和吃子（提子）操作都是调用了Game接口中具体实现类中的move或kill方法来做的。

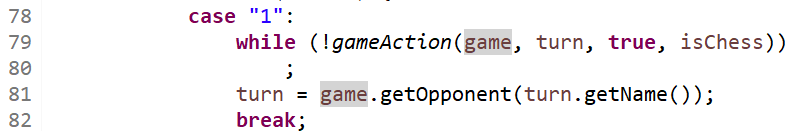


游戏结束时，可以访问走棋记录，这是通过调用showHistory来实现的。

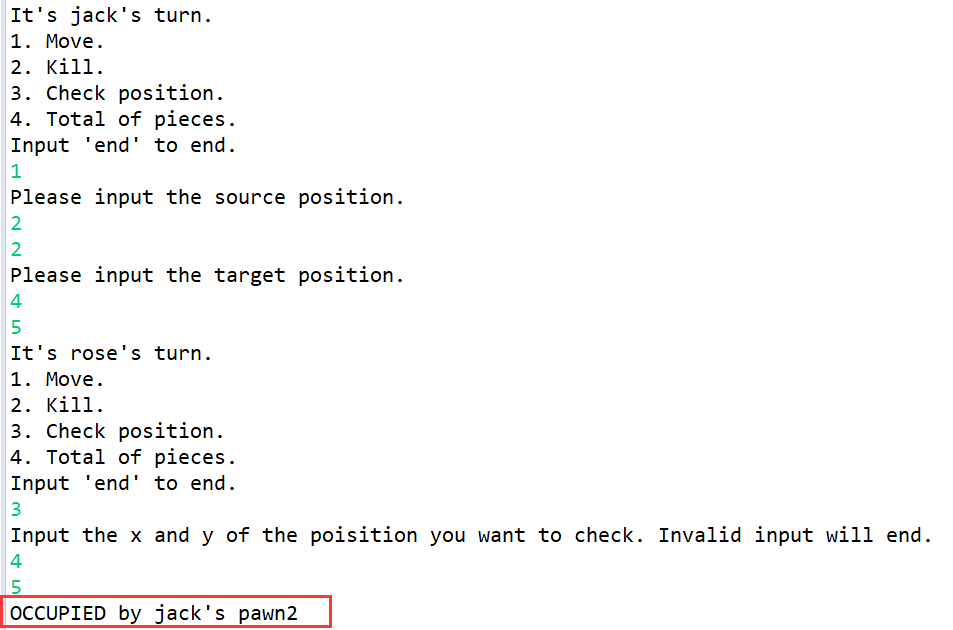


##### 国际象棋移动

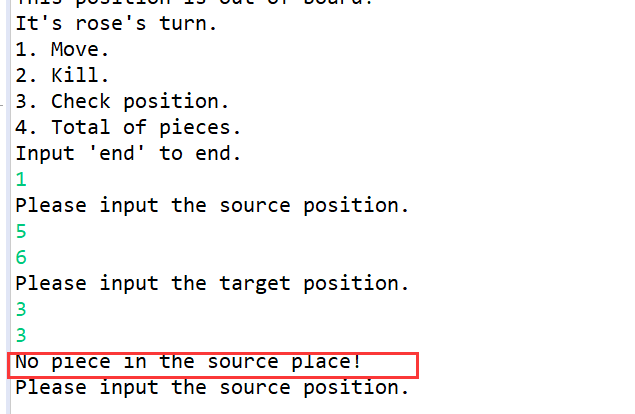
client输入1，则执行移动操作。使用循环反复检查移动的合法性，直到出现一次合法的移动操作，则交换执行权，跳出循环，进入对手回合。



一次合法的移动操作以及检查如下：



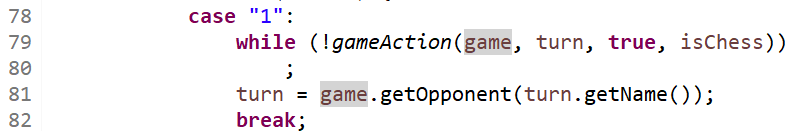
一次非法的移动操作：



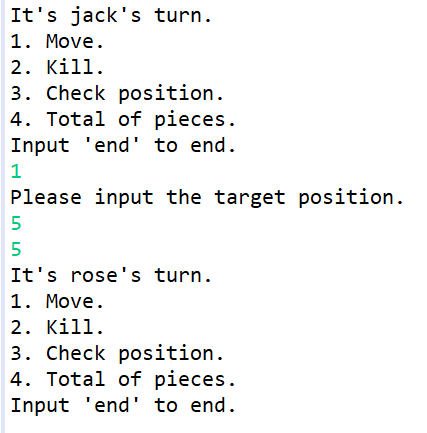
还有其它多种异常处理。直到有一次合法移动操作为止。移动操作无法撤回。

##### 围棋移动

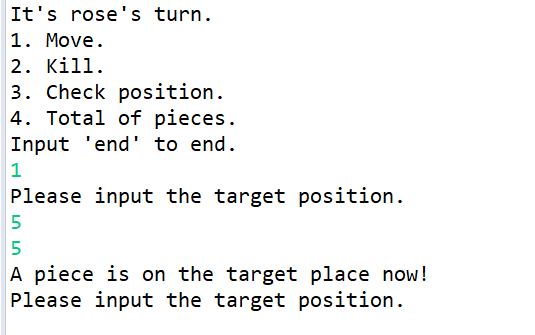
client输入1，则执行移动操作。使用循环反复检查移动的合法性，直到出现一次合法的移动操作，则交换执行权，跳出循环，进入对手回合。



一次合法的移动操作：



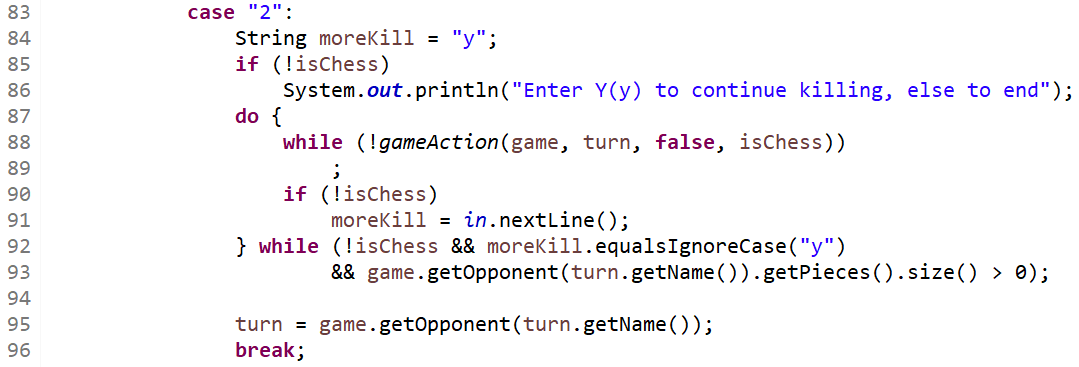
一次非法的移动操作（发生在上述jack移动之后）：



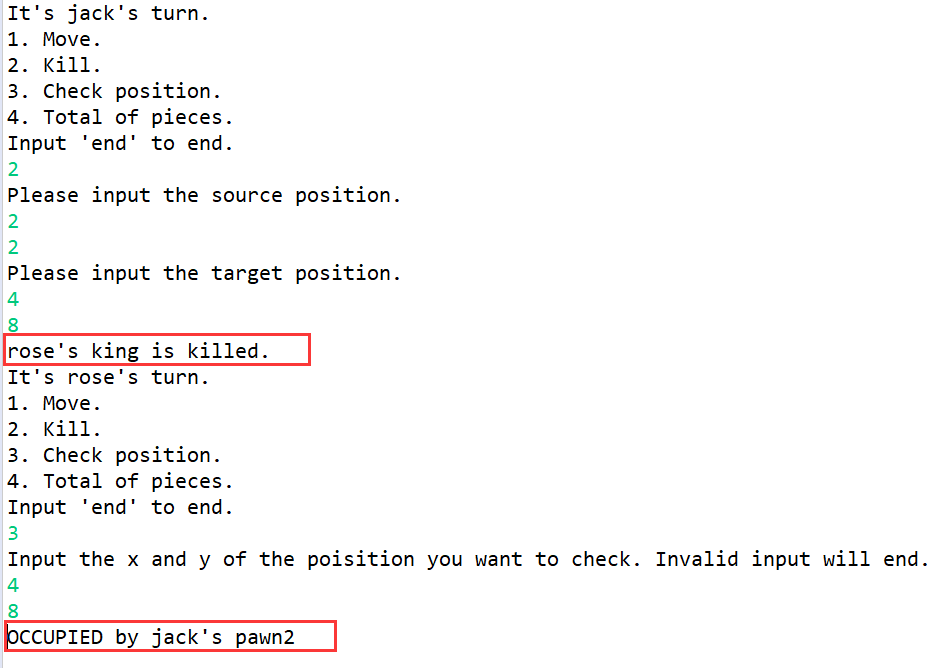
还有其它多种异常处理。直到有一次合法移动操作为止。移动操作无法撤回。

##### 国际象棋吃子

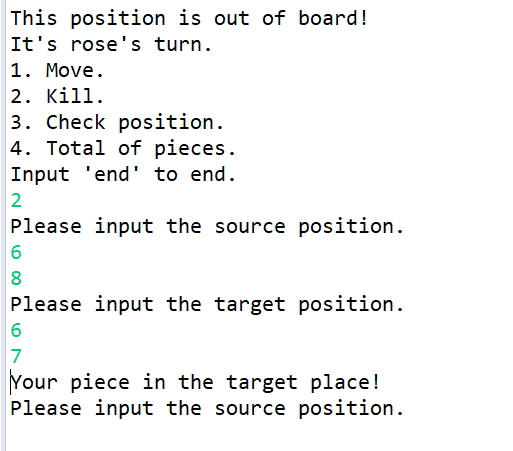
client输入2，执行吃子操作，这里会使用两层循环，第一次层在国际象棋游戏中会执行，内层循环是对吃子操作的合法性检查，直到有一次合法的吃子操作，才会跳出循环，交换执行权，进入对手回合。



一次合法的吃子操作：



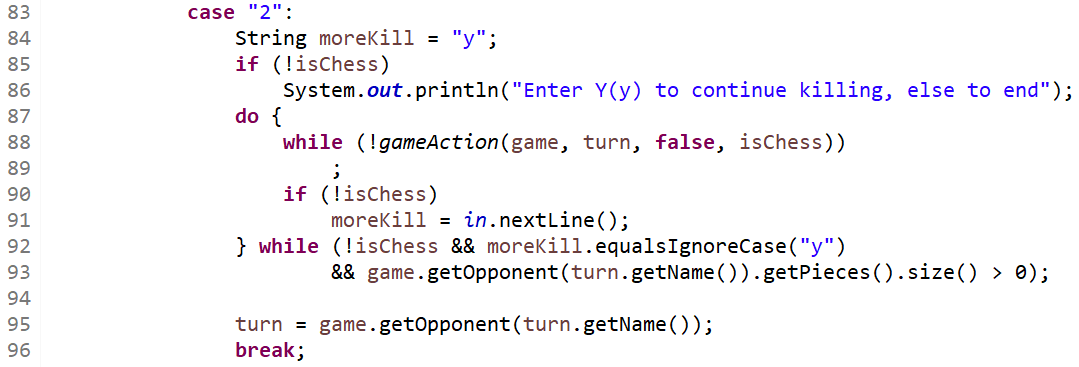
一次非法的吃子操作：



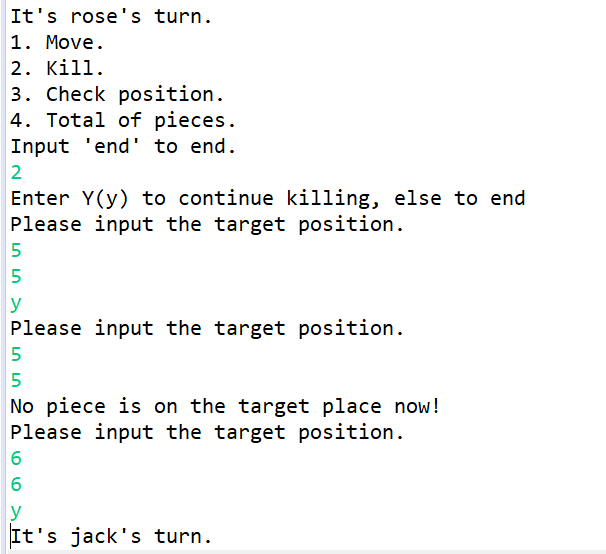
还有其它多种异常处理。直到有一次合法吃子操作为止。吃子操作无法撤回。

##### 围棋提子

client输入2，执行吃子操作，这里会使用两层循环，第一次层在围棋游戏中会执行，是为了判断是否还有棋子需要提走，因为围棋中一次提子，可能要提走多个对方棋子。内层循环是对吃子操作的合法性检查，直到有一次合法的吃子操作，才会跳出循环，交换执行权，进入对手回合。



一次提子操作：

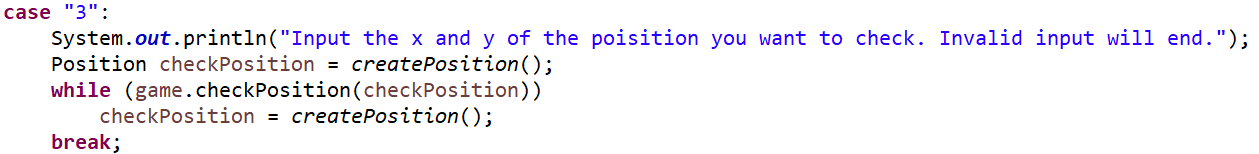


注意，本次提子操作是在jack在（5，5）和（6，6），rose在（8，8）下子之后，本次操作中包含了合法和非法提子操作，而且在rose提走了jack仅剩下的2个棋子之后，即时输入y继续执行提子，但因为jack已经没有棋子，所以直接跳出提子操作，交换执行权。

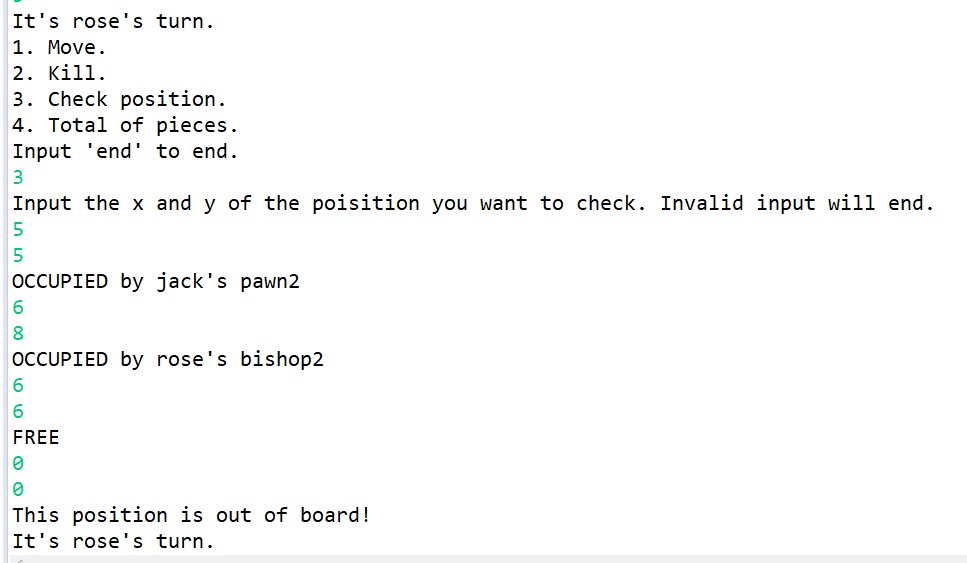
还有其它多种异常处理。直到有一次合法吃子操作为止。吃子操作无法撤回。

##### 检查棋盘指定位置

国际象棋和围棋的这一功能是相同的，只不过信息输出的形式不一样。



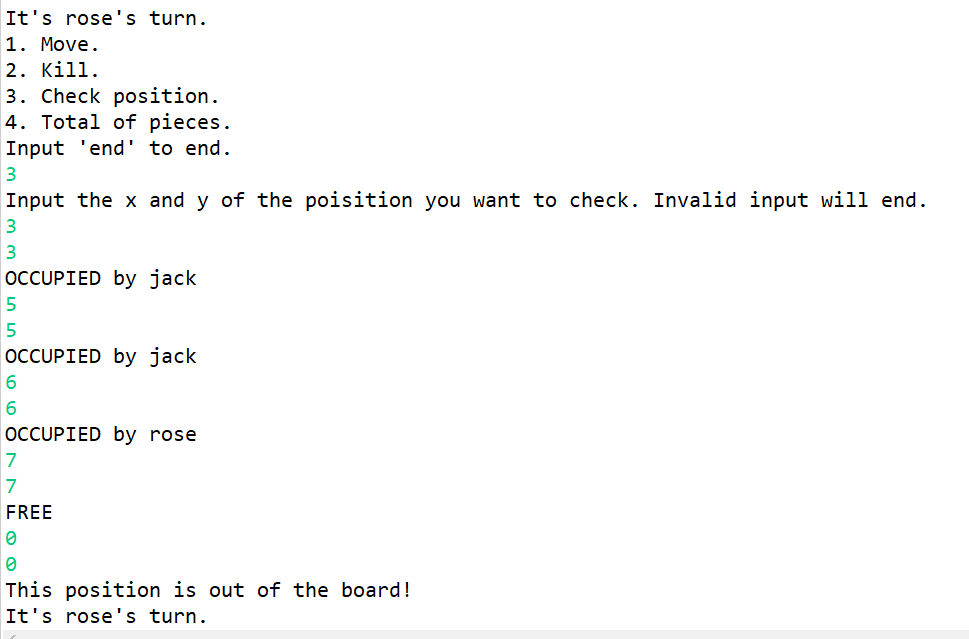
一次国际象棋中的位置检查如下：



这次的位置检查发生在jack将（2，2）的pawn2移动到（5，5）之后，可以看到rose检查了三个位置，（5，5），（6，8），（6，6），最后一次输入是结束本次位置检查的触发机制，如果输入了某个非法位置，那本次位置检查停止。

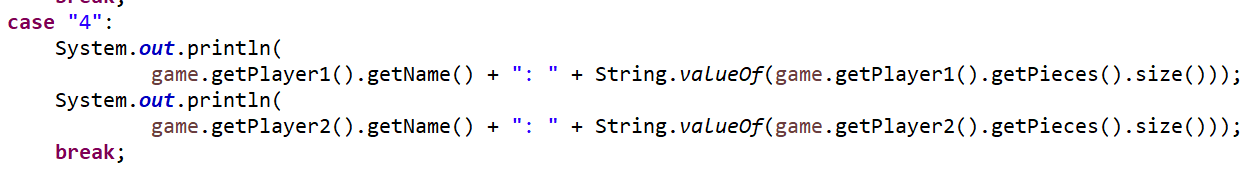
位置检查结束后，执行权不交换，所以，如果误触了一个非法位置导致当此位置检查结束，是可以再次检查的。

一次围棋中的位置检查如下：

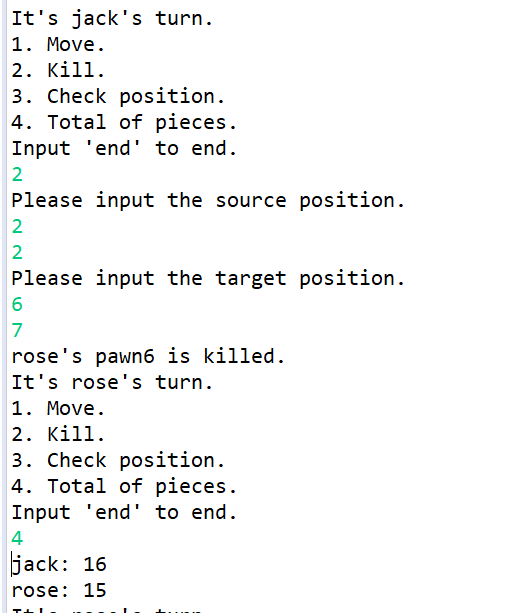


##### 获取当前双方棋子数

该功能在国际象棋和围棋中也是完全一样的。实现思想只要获取游戏中两个玩家，并访问一下它们管理的棋子集合计数就好了。



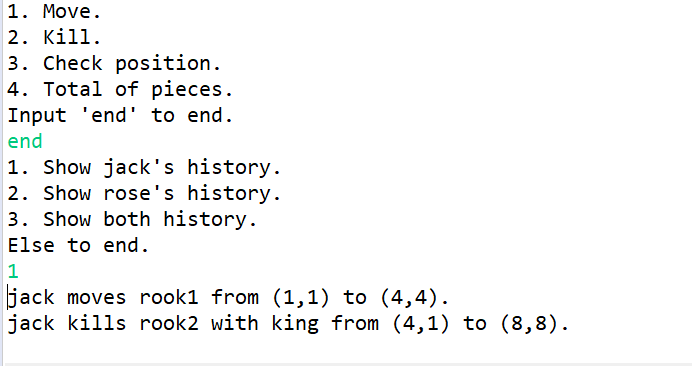
这个功能应该说没有什么合法性检查，只是获取两个数字。



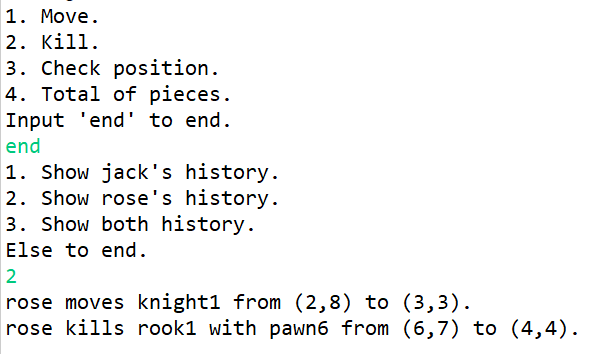
##### 访问走棋历史

这个功能只有在任一方输入end后，当前棋局结束之后，才会提供访问走棋历史功能。访问走棋历史功能提供三种选项，三种选项只能选择一种，然后程序结束。下列均以国际象棋为示例，三种举例中的行棋历史是相同的。国际象棋和象棋功能相同，只是查询的输出形式不同。

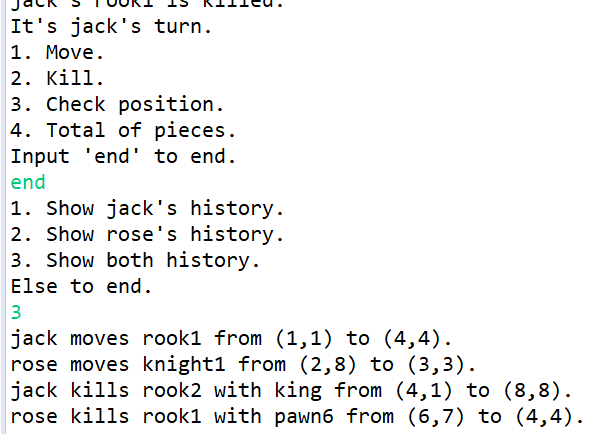
访问player1的走棋历史。



访问player2的走棋历史。

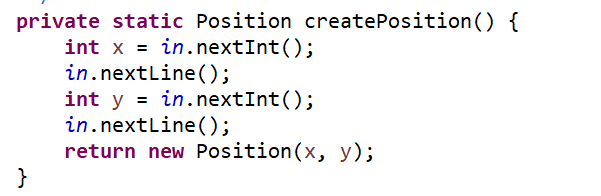


访问双方的走棋历史。

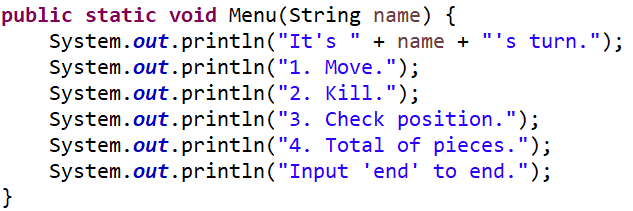


#### 辅助方法

辅助方法只有一个createPosition，是为了把二维坐标x和y封装到Position对象中。



当然还有菜单函数也没啥用。

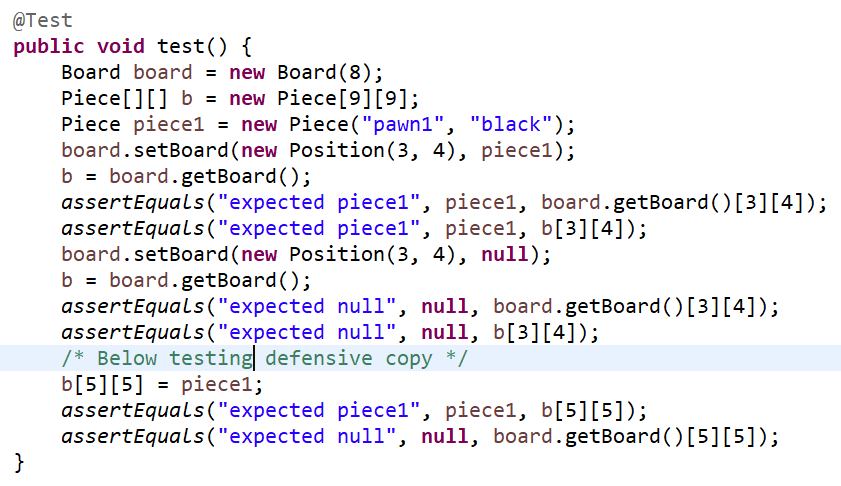


### ADT和主程序的测试方案

#### ADT test

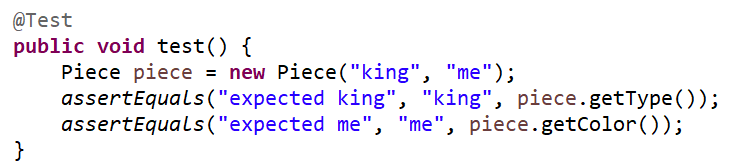
##### Board test

这部分测试，主要测试的是Board类中的setBoard Mutator方法，和getBoard Observer方法。一个是验证Mutator方法执行的结果正确性，一个是验证Observer方法的防御性拷贝是否成功。



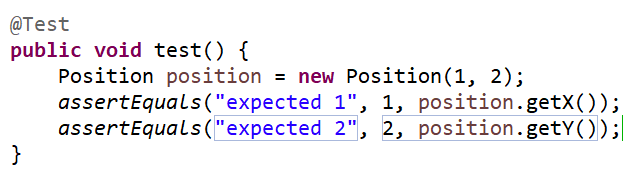
##### Piece test

Piece类的测试就好做多了，这个类里面只有两个Observer方法，那就检验一下Observer结果就行了。而且这个类的rep都是immutable的，还不用检查防御性拷贝。



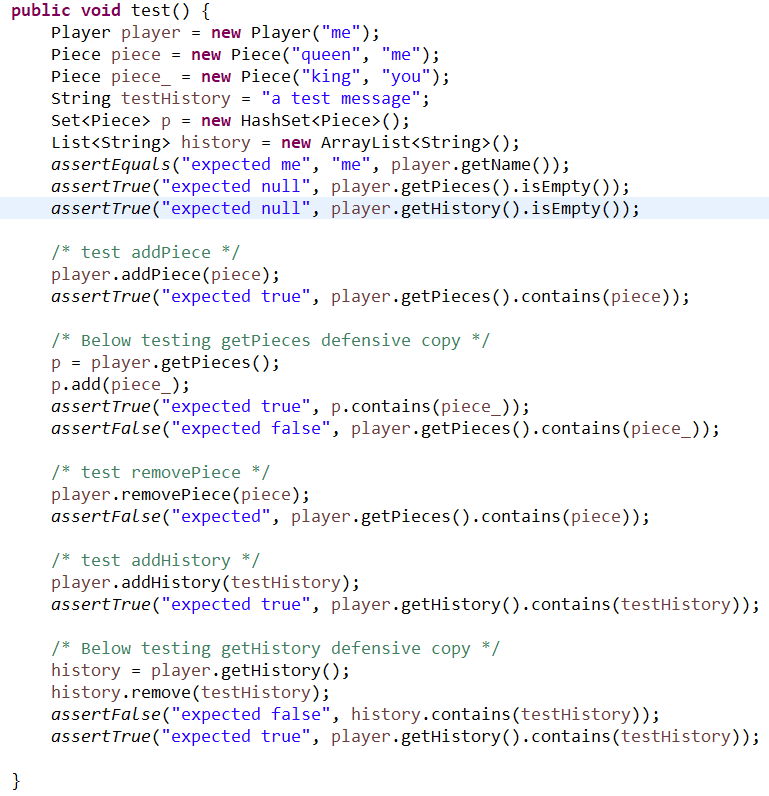
##### Position test

对Position类的测试几乎不用测试了，我对它的定位仅仅是一个数据结构，类似于C中的结构体，封装了x和y坐标。



##### Player test

Player测试相比之前有点多。因为它有两个mutable的rep，而这两个rep都有Observer方法，最好对防御式拷贝也做测试。还有一些Mutator方法。

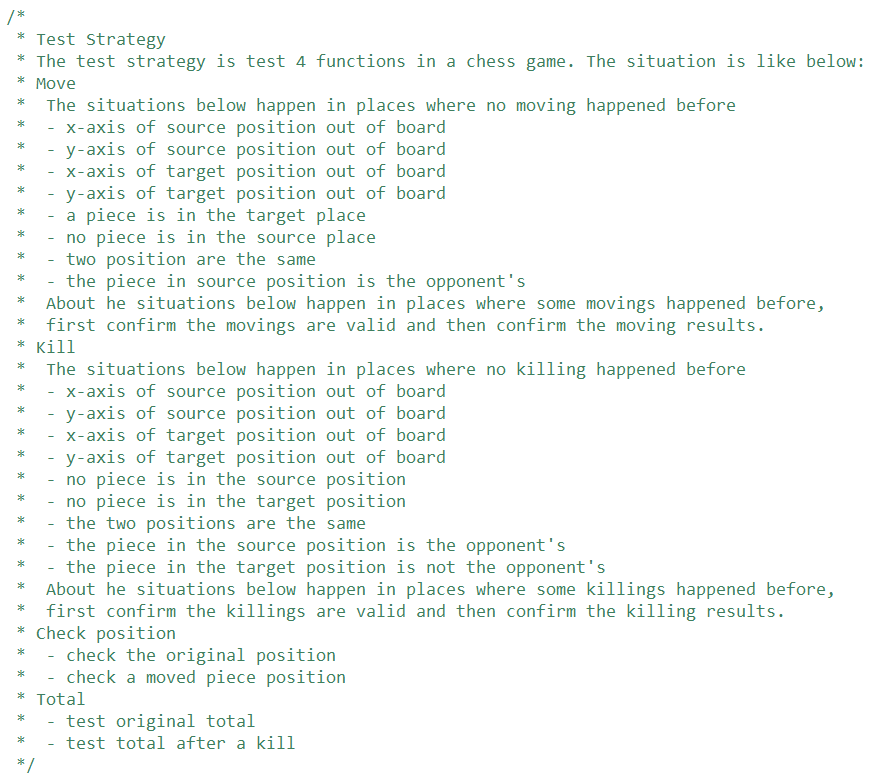


#### 应用功能测试

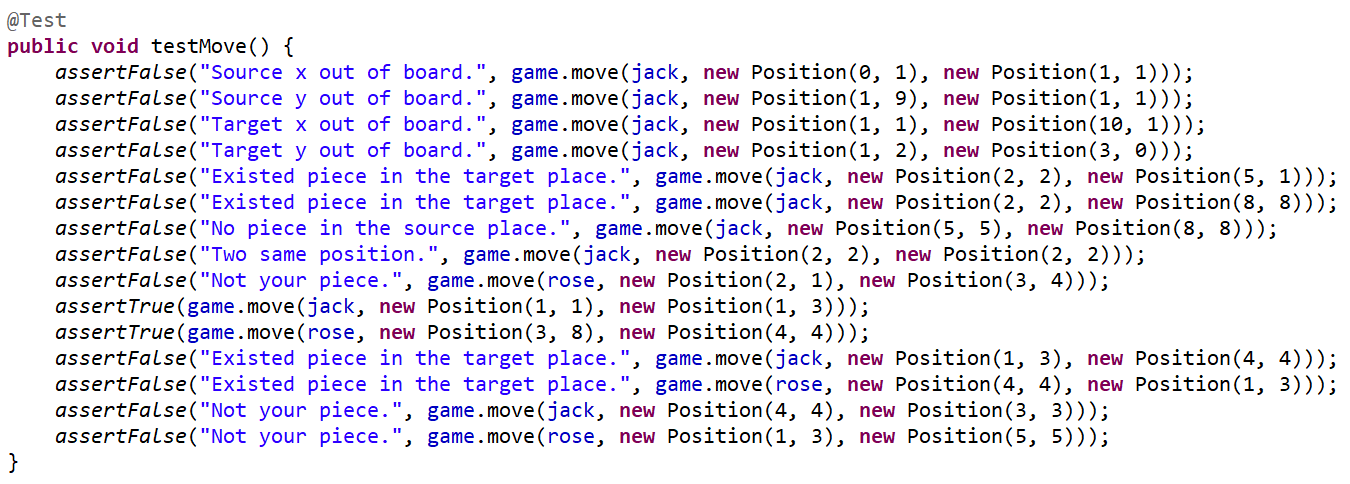
因为我的Game是接口，而chessGame和goGame是具体实现接口的两个类，而且这两个类中包括了所有的其它ADT。所以对这两个类分别测试ADT。而且在测试时，也遵从按照客户端运行的逻辑，以功能为单位，分别测试。

##### 国际象棋

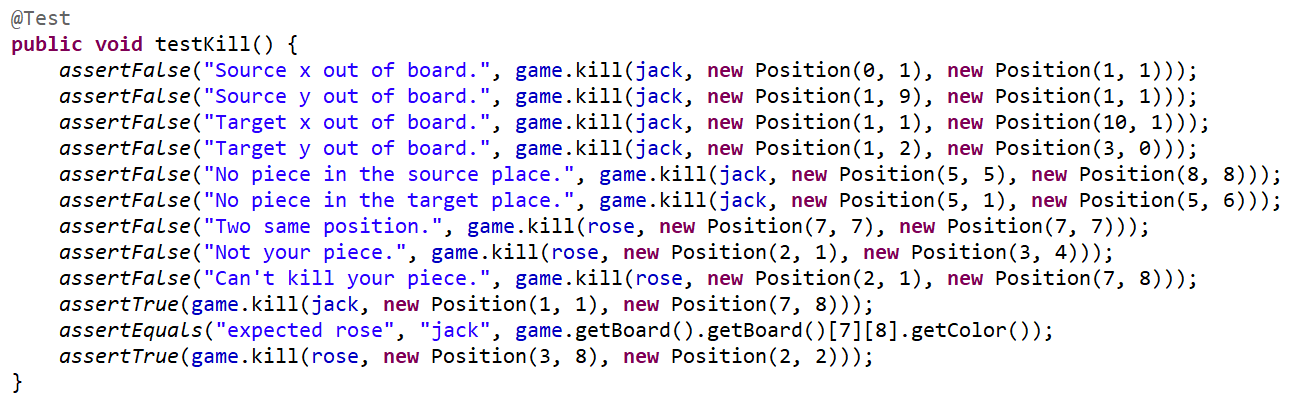
先放上test strategy。



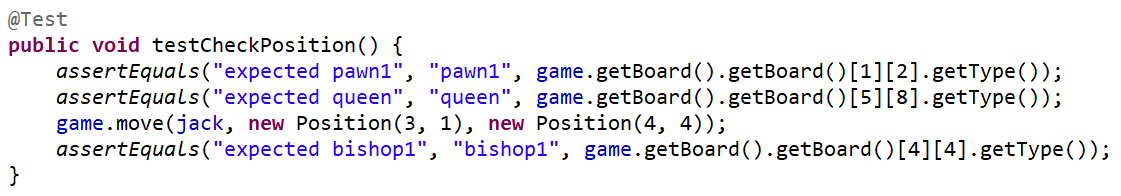
先测试move相关功能。按照move输入的参数逐个分析，对可能出现异常的多种输入组合测试。在测试完可能异常的情况之后，还要对合法的移动操作，做操作结果测试。所有测试的情况说明，已经写在了断言错误报告中（assert…函数的第一个参数中）。



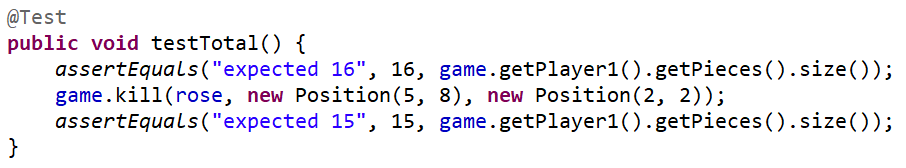
再测试kill相关功能。和move的测试思想一样，在测试完可能异常的情况后，对合法的吃子操作做结果检查。



再测试检查位置功能。由于这个功能其实是一个类似观察器的功能，所以结合其它修改功能来检查。

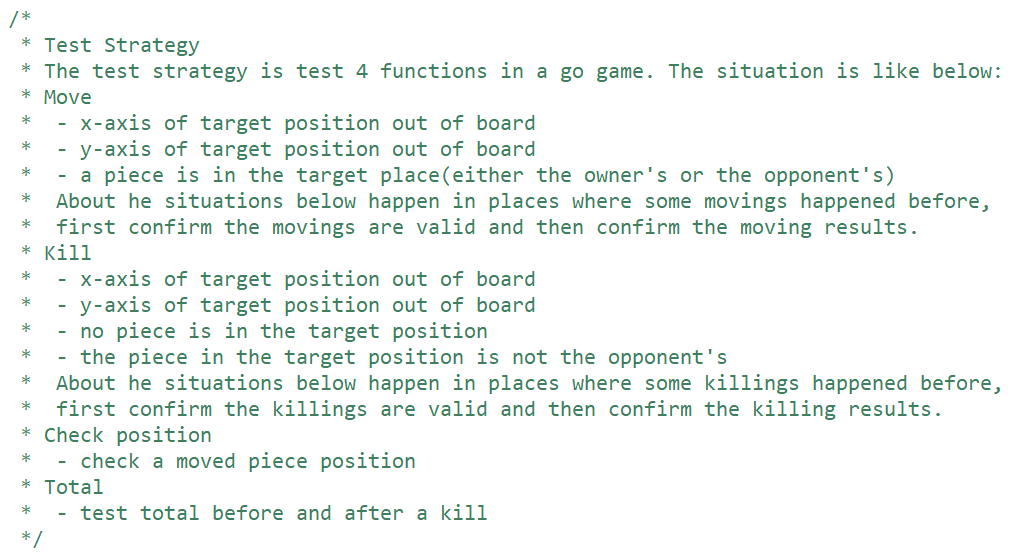


获取双方各自棋子总数，和检查位置功能一样，配合其它修改功能来检查。



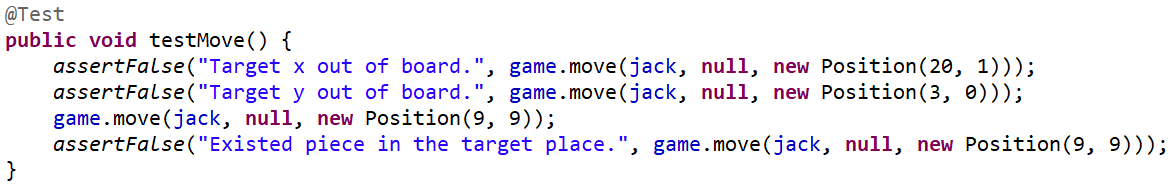
##### 围棋

先放上test strategy。

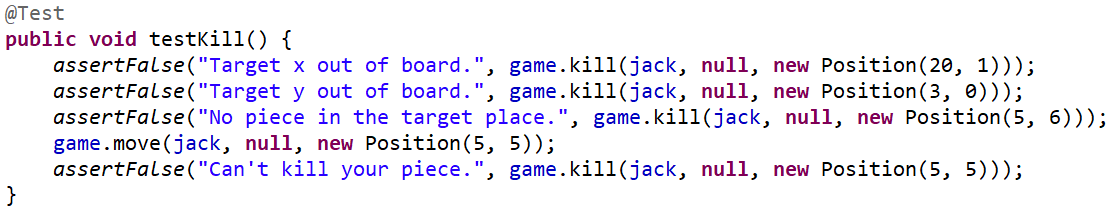


测试思想和国际象棋的测试差不多，只不过move和kill的测试更简洁，因为没有source参数。而检查位置和总数功能的测试需要更多地配合move和kill来检查。

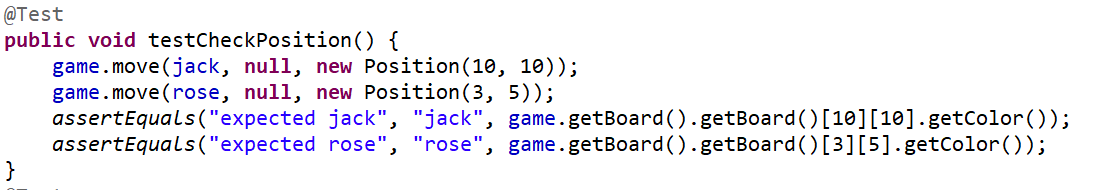
走棋功能测试。



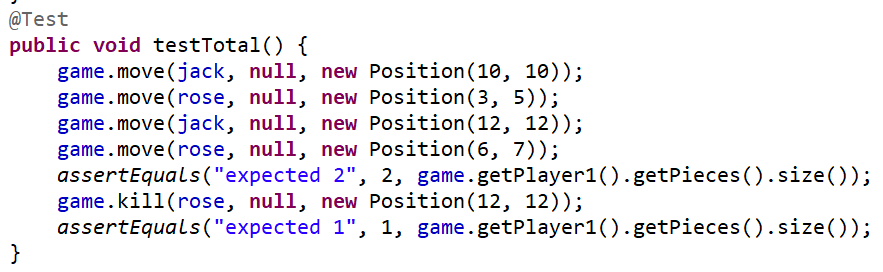
提子功能测试。



检查位置测试。



总数测试。



## Multi-Startup Set (MIT)

未做

# 实验进度记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 2019-3-11 | 18:50-23:00 | 写了GraphInstanceTest.java里的test，写了ConcreteEdgesGraph.java里的方法代码 | 完成 |
| 2019-3-12 | 19:00-22:00 | 写了ConcreteVerticesGraph.java里的方法，反复修改类的方法和Vertex的域，并应用le 泛型 | 完成 |
| 2019-3-13 | 18:30-21:50 | 基本完成了Poetic Walks的代码和测试代码，注释还未完善。 | 完成 |
| 2019-3-16 | 18:30-22:00 | 初步完成了P3的chess中的一些功能，健壮性和数据结构有待提高和修改 | 完成 |
| … | … | … | … |
| 2019-4-5至2019-4-7 |  | 完成实验报告 |  |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| 在做P1时，ConcreteVerticesGraph中仅给了一个rep，实现起来有些蹩脚 | 因为图的表示需要点集和邻接表，而要求又只能用一个点集rep，所以索性把每个顶点的邻接链表放到Vertices类中。 |
| P1中Edge类要求是一个immutable类，其中不能有mutator方法，而具体实现方法中很多操作要修改边相关。 | 凡是需要修改边的地方，全部改为清除后再添加 |
| 测试优先的编程习惯非常不适应 | 没办法了，既然测试优先是个好习惯，只能硬着头皮上了。最开始写P1测试时很蹩脚，不过后来就把测试当成数学里的分类讨论题来做了 |
| P1中GraphPoet的实现很蹩脚，有很多概念spec中叙述的模糊不清，而且凡是涉及文件文本处理的都是一个很繁琐的过程 | 降低了数据处理的标准，用正则表达式将文本中的常见标点符号全部去掉，并仅以换行符（回车符）和空格作为划分标志，将语料库文本分割成存储进单词列表。然后再利用之前的ADT构造这个图。 |
| P3的ADT关系设计在最开始也是非常困难的 | 对六个给定的ADT在纸上画了好几遍关系图，才确定了包含和所属关系，最后确定了Game和Action是接口，其他是子类的设计。而且最开始写的也是磕磕撞撞，改了三次才写好。 |
| 本次实验最难的也是最深刻应该是spec规约的更高要求了吧，AF，RI，表示泄露等 | 这些spec中的要求AF, RI, 表示泄露这些东西直到开了实验两周之后，课上才讲，所以最开始看到的时候真的是一头雾水。刚开始写的时候也是对AF和RI摸不清头脑，觉得很多东西都没必要写。不过最后还是一条一条的写出来了，可能写的也有不规范。至于RI，把checkRep写好之后，感觉条理能更清晰一点。至于test strategy，就把它当成是高中数学中的分类讨论题来做就好了 |

# 实验过程中收获的经验、教训、感想

## 实验过程中收获的经验和教训

以后采用测试优先的编程习惯，先写测试，然后再不断地完成主体代码，每写完一个阶段，就运行一下测试代码，可以及时发现bug。

从零开始写一个可复用的ADT时，多思考各个类之间的关系，理清关系再写。

在完善ADT的过程中，一定要时时刻刻注意防止表示泄露，不要到完善客户端代码时再去注意这个问题，不然到时候再发现，需要改的东西会很多。而且如果没有在客户端处发现这个问题，不代表其它client使用代码时不会遇到这个问题。总之要严格防止表示泄露。而且，尽可能多的使用immutable类型。

## 针对以下方面的感受

1. 面向ADT的编程和直接面向应用场景编程，你体会到二者有何差异？

前者主要把时间花在了设计上，后者主要把时间花在了逻辑结构实现上。前者在具体实现上会变得很轻松，而后者很有可能，在具体实现的过程遇到一个致命问题，就全局推倒重来了，可供修改的弹性很小。

1. 使用泛型和不使用泛型的编程，对你来说有何差异？

对我来说差异可能主要就是注意一下immutable类型和mutable类型，此外没有什么明显差异。主要差异在客户端那边，即使用者复用使用泛型的编程ADT效率更高。

1. 在给出ADT的规约后就开始编写测试用例，优势是什么？你是否能够适应这种测试方式？

在后续编程中频繁地运行测试代码，可以及时找出潜在的bug。如果测试用例写得好，那么从最开始就奠定了ADT代码编写的标准，不管是完善代码还是debug都有更明确的目标。不至于在编写过程中突然发现了一个bug，而debug的过程中发现更多bug，推倒重来，这种情况可以大大避免。刚开始不适应，后来就适应了。

1. P1设计的ADT在多个应用场景下使用，这种复用带来什么好处？

对客户端来说，调用的频率高，效率高。对于维护者来说，维护起来更加容易，修改的弹性更大。

1. P3要求你从0开始设计ADT并使用它们完成一个具体应用，你是否已适应从具体应用场景到ADT的“抽象映射”？相比起P1给出了ADT非常明确的rep和方法、ADT之间的逻辑关系，P3要求你自主设计这些内容，你的感受如何？

刚开始做起来有点蹩脚，后来能熟悉这种编程习惯。这种编程模式有点像杂物归类，给物品贴标签，而归类的方式每个人都不同。就好比棋子既可以是由棋局管理，也可以是由棋盘管理，也可以是由玩家管理，但很难理解棋子由“行为”管理，那么这样的话Piece类就不太可能被用到Action类中。逻辑范围越大的ADT，越有可能是接口，比如说Game类，在本次实验中有两个应用场景，不如就把它作为接口，用具体类去实现它。

1. 为ADT撰写specification, invariants, RI, AF，时刻注意ADT是否有rep exposure，这些工作的意义是什么？你是否愿意在以后编程中坚持这么做？

我觉得有些类中写这些有意义，有些类中写这些实在没什么意思。这种工作习惯有助于团队开发，项目开发责任的交接，让其他人更快的理解自己的代码，同时也是对客户负责。

今后的编程中，防止表示泄露我会一直践行下去，spec也是必须要写的，几乎所有方法都要写spec。但是RI和AF，有些类会写，有些类实在没什么好写的。

1. 关于本实验的工作量、难度、deadline。

从分值上也能理解工作量加大了吧。

最坑的是，强烈谴责图模型这门课！难度太大！这门课几乎把我的实验二以及其它专业课的学习打停了将近三周！实在是苦不堪言！我的实验报告完成质量显然没有实验一高了，博客也无暇兼顾，还有本报告中4、5节的总结部分没有好好完成，都是因为图模型那门课切割了我的实验完成进度。

本次实验刚开始上手有点难，后来才整明白，可老师却迟迟不讲AF, RI等知识，拖了两周才介绍，这些内容其实后来又改了好久。希望下次如果遇到了，还是提前讲吧，这毕竟算是一次实验中的主要内容啊，不像Git这些简单看看就行了。

Deadline没什么好评价的，够长是够长了，就是我把时间都放在图模型上了。

1. 《软件构造》课程进展到目前，你对该课程有何体会和建议？

建议老师在实验开始前，讲完实验主要内容涉及的知识点，比如本次需要用到的AF, RI等，一些边边角角的东西我们自己看就算了，这些主要内容不提前告知我们，后来再改很麻烦的，如果这些主要内容也需要我们提前自学的话，那工作量又大了。

建议以后的实验，从每次实验的第二次实验课开始，再切换到新的实验TA，因为每次实验的第一次实验课我们都在看实验手册，这个过程没啥好问TA的。