

**2024年春季学期  
计算学部《软件构造》课程**

**Lab 2实验报告**

|  |  |
| --- | --- |
| 姓名 | 罗昊然 |
| 学号 | 2022113415 |
| 班号 | 2237101 |
| 电子邮件 | [hungercarrots@qq.com](mailto:hungercarrots@qq.com) |
| 手机号码 | 13383324846 |

**目录**

[1 实验目标概述 1](#_Toc164848256)

[2 实验环境配置 1](#_Toc164848257)

[3 实验过程 4](#_Toc164848258)

[3.1 Poetic Walks 4](#_Toc164848259)

[3.1.1 Get the code and prepare Git repository 4](#_Toc164848260)

[3.1.2 Problem 1: Test Graph <String> 5](#_Toc164848261)

[3.1.3 Problem 2: Implement Graph <String> 7](#_Toc164848262)

[3.1.3.1 Implement ConcreteEdgesGraph 7](#_Toc164848263)

[3.1.3.2 Implement ConcreteVerticesGraph 10](#_Toc164848264)

[3.1.4 Problem 3: Implement generic Graph<L> 13](#_Toc164848265)

[3.1.4.1 Make the implementations generic 13](#_Toc164848266)

[3.1.4.2 Implement Graph.empty() 14](#_Toc164848267)

[3.1.5 Problem 4: Poetic walks 16](#_Toc164848268)

[3.1.5.1 Test GraphPoet 17](#_Toc164848269)

[3.1.5.2 Implement GraphPoet 17](#_Toc164848270)

[3.1.5.3 Graph poetry slam 18](#_Toc164848271)

[3.1.6 使用Eclemma/Code Coverage for Java检查测试的代码覆盖度 19](#_Toc164848272)

[3.1.7 Before you’re done 20](#_Toc164848273)

[3.2 Re-implement the Social Network in Lab1 21](#_Toc164848274)

[3.2.1 FriendshipGraph类 21](#_Toc164848275)

[3.2.2 Person类 23](#_Toc164848276)

[3.2.3 测试用例 23](#_Toc164848277)

[3.2.4 提交至Git仓库 24](#_Toc164848278)

[4 实验进度记录 24](#_Toc164848279)

[5 实验过程中遇到的困难与解决途径 25](#_Toc164848280)

[6 实验过程中收获的经验、教训、感想 25](#_Toc164848281)

[6.1 实验过程中收获的经验和教训（必答） 26](#_Toc164848282)

[6.2 针对以下方面的感受（必答） 26](#_Toc164848283)

# 实验目标概述

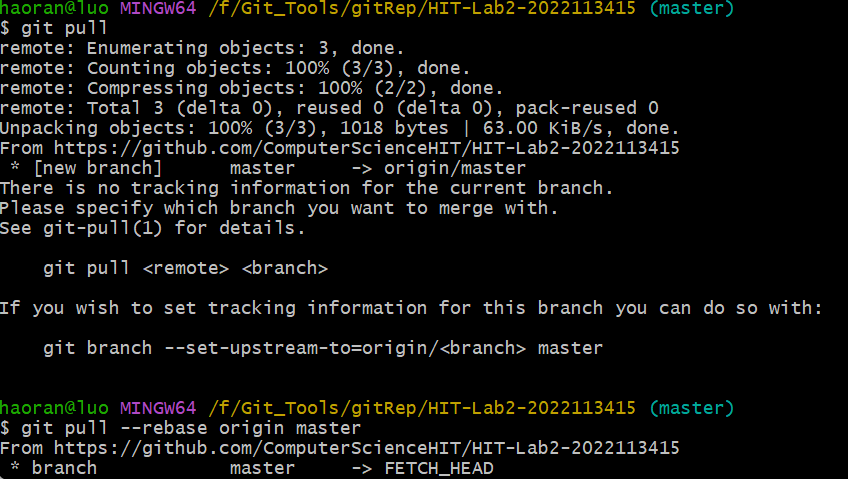
本次实验训练抽象数据类型（ADT）的设计、规约、测试，并使用面向对象编程（OOP）技术实现 ADT。具体来说：

* 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 的多种不同的实现；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 的实现并评估测试的覆盖度；
* 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。

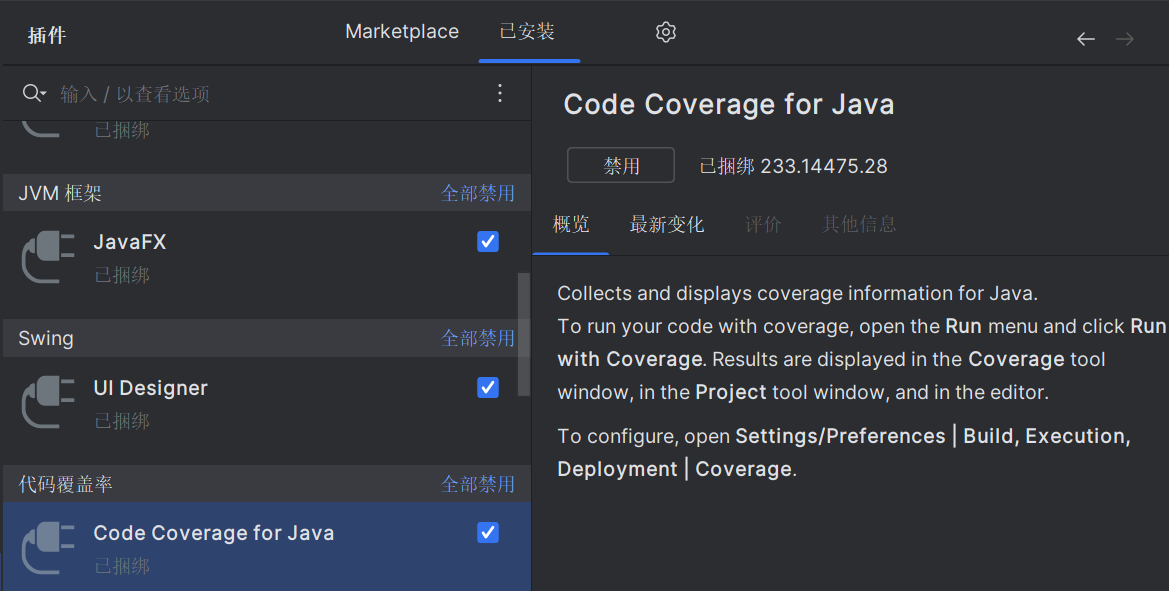
# 实验环境配置

JDK、IDE与Git工具的配置在Lab1中已经完成，不在赘述。

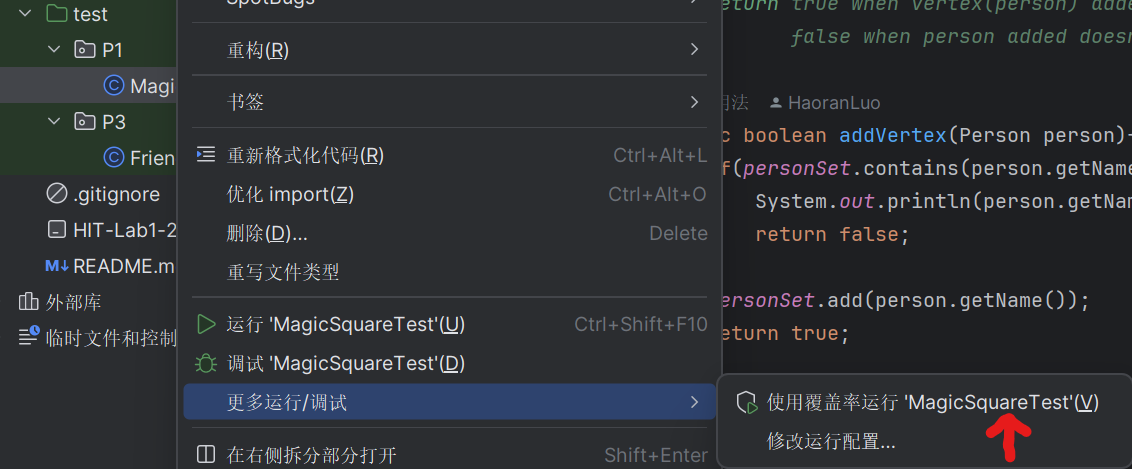
同Lab1一致，在链接至远程仓库后，由于远程仓库在创建时内部已经存在文件，故无法直接上传本地库的进度，需要从远程仓库pull一次与本地同步后才可上传。尝试git pull命令无果，在pull指令中添加—rebase参数后执行，成功。

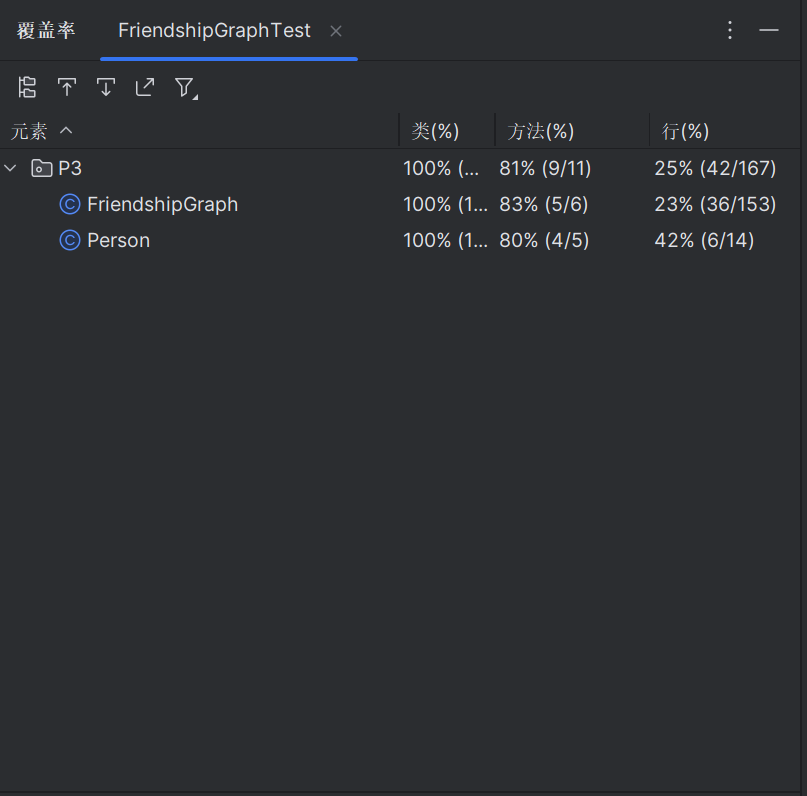


本报告中实验在IDEA环境下完成，故使用IDEA自带的配置测试覆盖度插件Code Coverage for Java，不在另行安装配置其他插件。

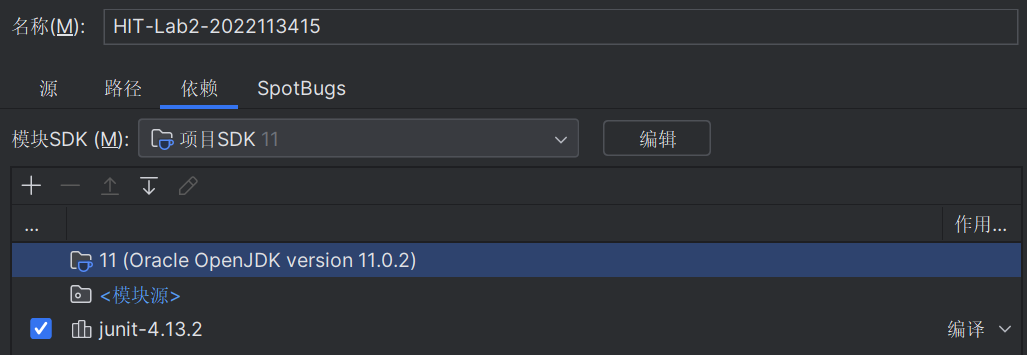


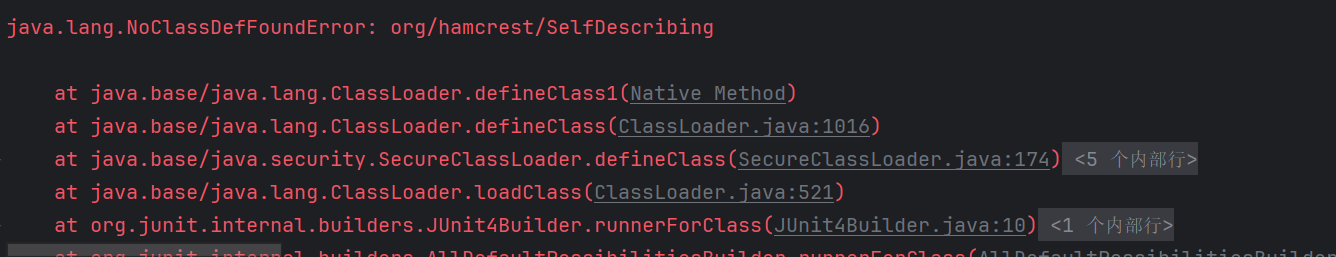
如图，这里在已经完成的Lab1项目中进行测试。对于已经实现的可以运行的类文件，通过右击要运行的类文件，在“更多运行/调试”中找到“使用覆盖率运行”即可。在类、方法与代码行三个维度的覆盖度测试结果将在IDEA界面的右侧显示。

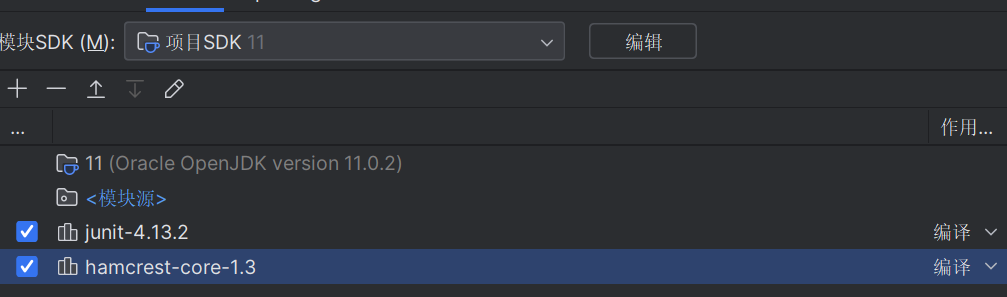


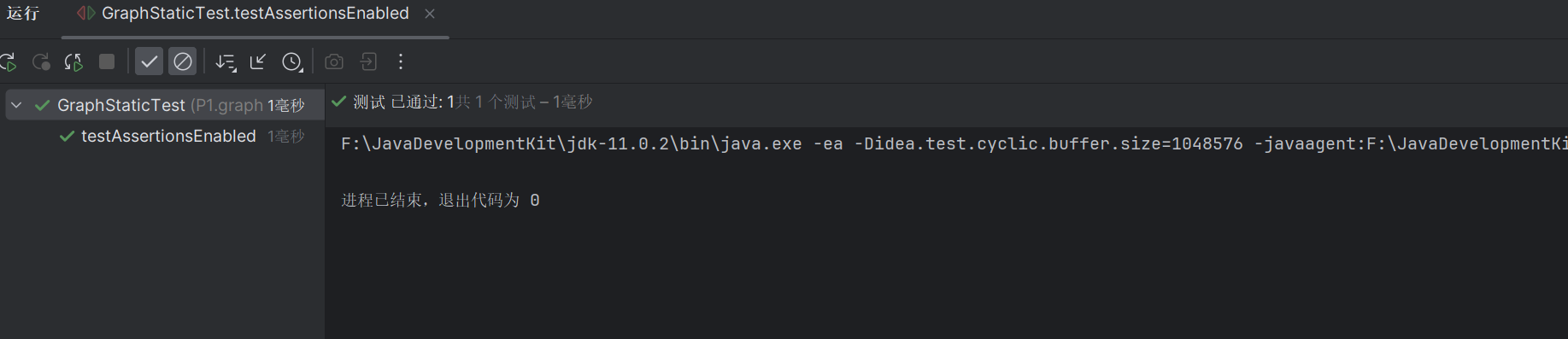


对新建的Lab2项目配置项目结构，导入Junit库用于测试，步骤同Lab1。运行Junit测试类是出现如下报错，查阅资料发现系手动导入Junit库时缺少hamcrest-core包导致，联网下载后导入后，Junit库正常运行。









最后给出本报告对应的GitHub Lab2仓库的URL地址：

<https://github.com/ComputerScienceHIT/HIT-Lab2-2022113415>

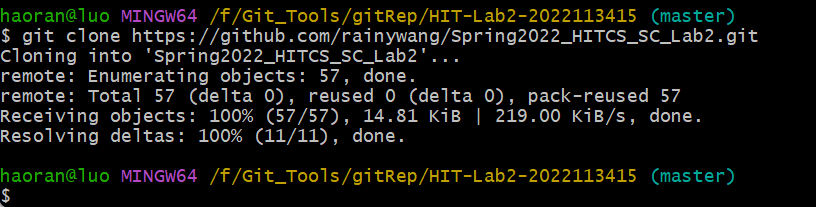
# 实验过程

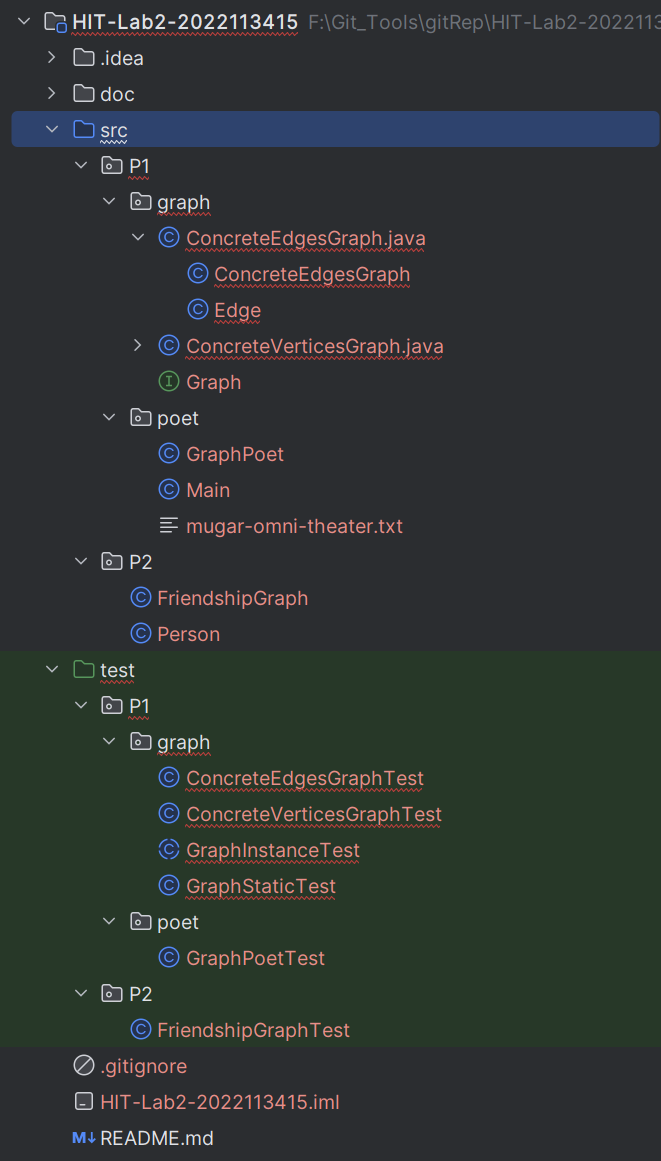
## Poetic Walks

任务要求我们实现Graph接口下的两个实现类ConcreteEdgesGraph和ConcreteVerticesGraph中的各个方法，编写有关测试用例，撰写AF、RI、预防表示泄露措施等，之后编写实现代码，并将这两个实现类扩展到泛型<L>，之后应用编写的抽象数据类型解决PoeticWalks问题。

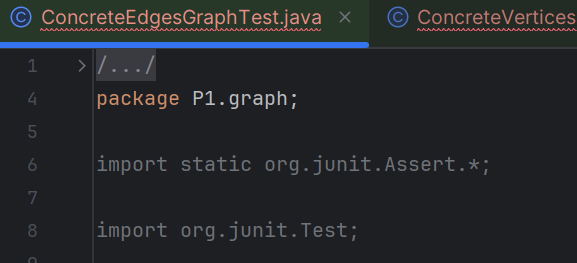
### Get the code and prepare Git repository

启动git Bash，cd进入当前项目目录，使用git clone命令从指定的链接获取初始代码，后按照要求的项目结构调整项目目录。





需要注意的是，clone的项目中各个类的package路径与本地项目的实际package路径不一致，需要人为调整修改“package xx”为“package P1.xx”。



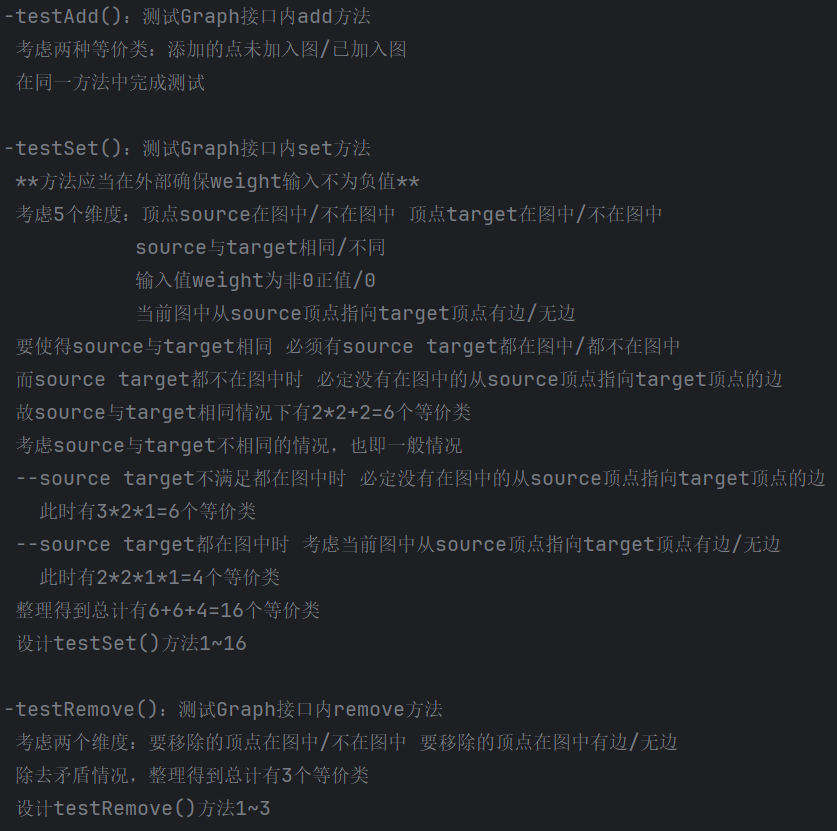
至此，完成clone远程库到本地库的配置与调整。

### Problem 1: Test Graph <String>

MIT页面要求先仅按照String的顶点类型编写测试用例，在之后再扩展到其他的数据类型，并且对静态方法的测试用例编写（也即GraphStaticTest.java的编写）在本问中涉及的部分已经提供了实现，故只编写GraphInstanceTest.java。

需要注意的是这里必须通过emptyInstance()方法获取空的graph类。

给出GraphInstanceTest.java测试类编写的策略：





完成后提交至Git。

### Problem 2: Implement Graph <String>

以下各部分，请按照MIT页面上相应部分的要求，逐项列出你的设计和实现思路/过程/结果。

Problem2要求实验者以两种不同的方式实现Graph<String>类，一种基于Edge类来架构的，一种基于Vertex类来架构，两种实现不得存在代码依赖或者共享关系。

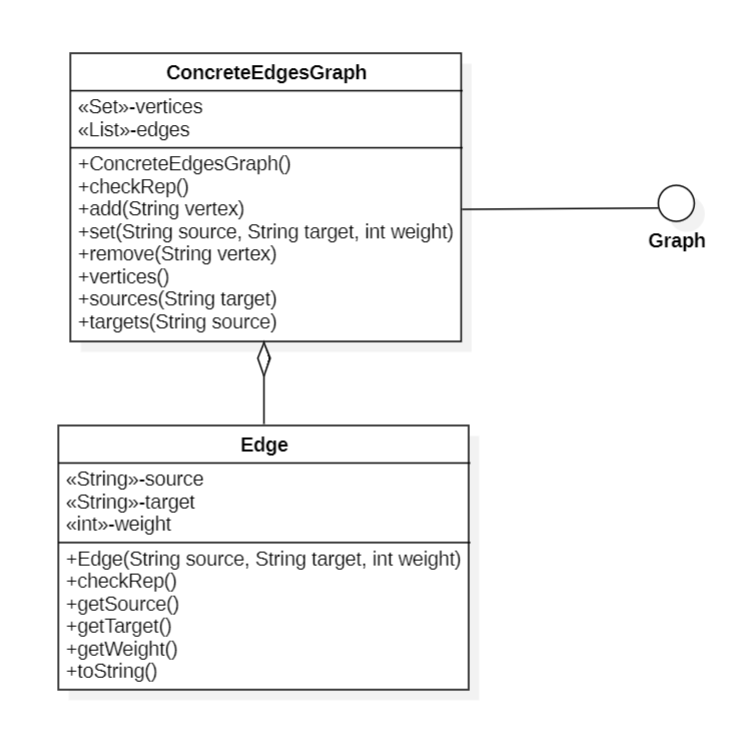
在两种实现的代码中需要：

* 记录类抽象函数与表示不变性
* 记录预防表示暴露的措施
* 实现对表示不变性检查的方法checkRep
* 实现toSring方法，返回一个人类可理解的抽象值表示

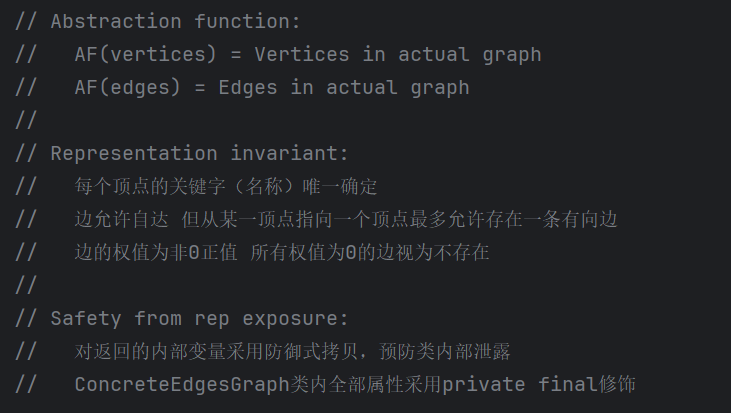
值得注意的是，MIT页面指出在当前问题中我们并不被强制要求重写equals和hashCode方法，这是由于在代码的实现中我们并不被要求判断不同的Edge或者Vertex对象是否相等。

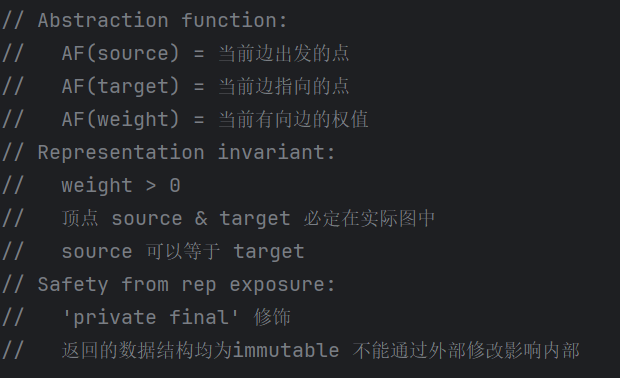
#### Implement ConcreteEdgesGraph

先进行Edge类的设计，大致绘制其与ConcreteEdgesGraph类的类图如下：

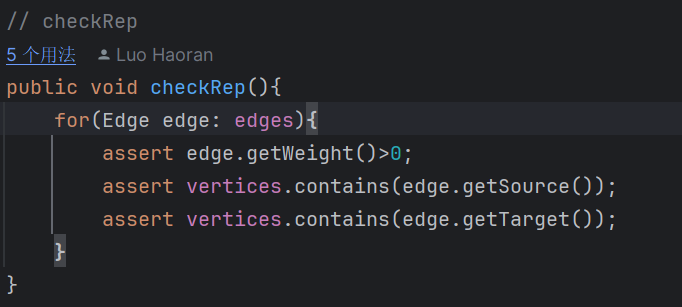


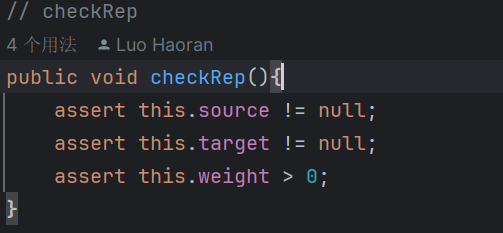
撰写ConcreteEdgesGraph类与Edge类的AF，RI及预防表示泄露措施的说明（上下两张图分别对应ConcreteEdgesGraph类与Edge类）：



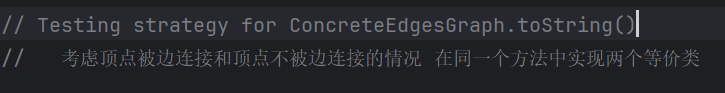


基于上面两个类中的RI要求，分别编写checkRep方法的实现，确保在ConcreteEdgesGraph类与Edge类内执行各方法后的结果均满足表达不变性。





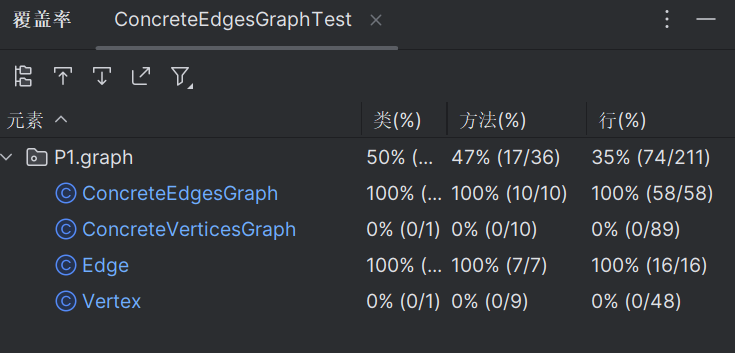
之后针对Edge类型进行其测试用例的进行并将设计结果实现到test包中的ConcreteEdgesGraphTest类中，给出用例设计策略如下：





最后实现各个方法，对类内的属性与非Override的各个方法给出JavaDoc说明（见源码）。

使用覆盖率运行ConcreteEdgesGraphTest类，由于它是GraphInstanceTest类的extend，故运行时会一并运行GraphInstanceTest测试类中的测试方法，得到测试结果以及覆盖率状况如下：

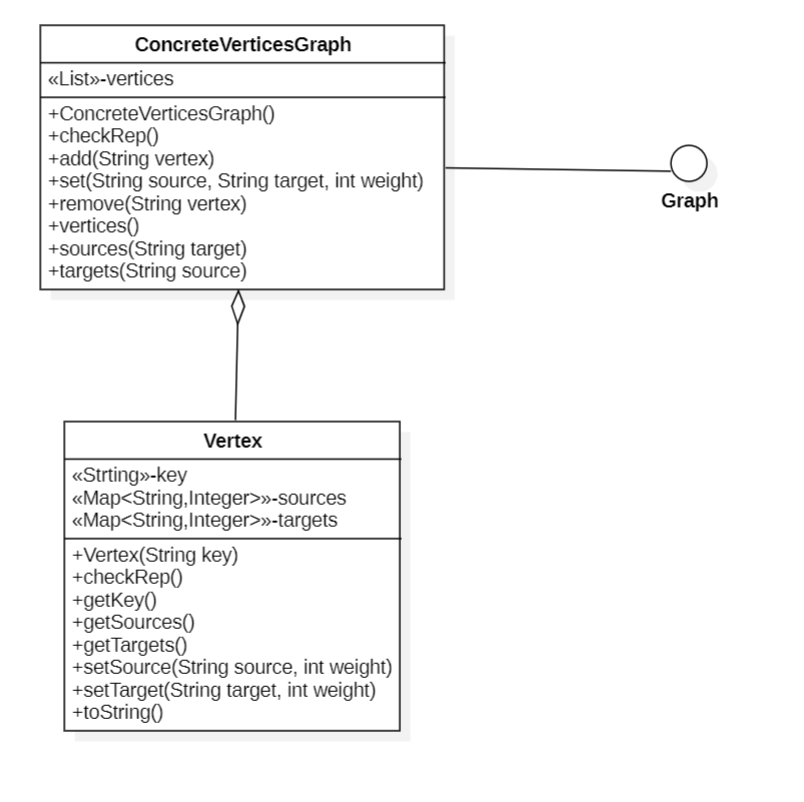




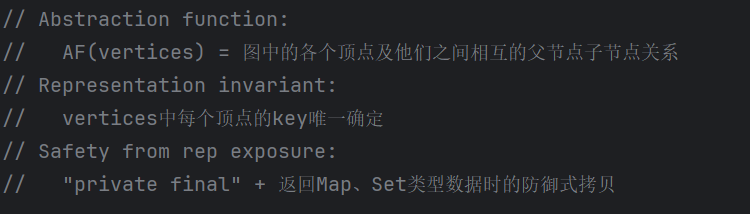
此时可以认为String类型下的Graph接口中的ConcreteEdgesGraph实现已全部完成。

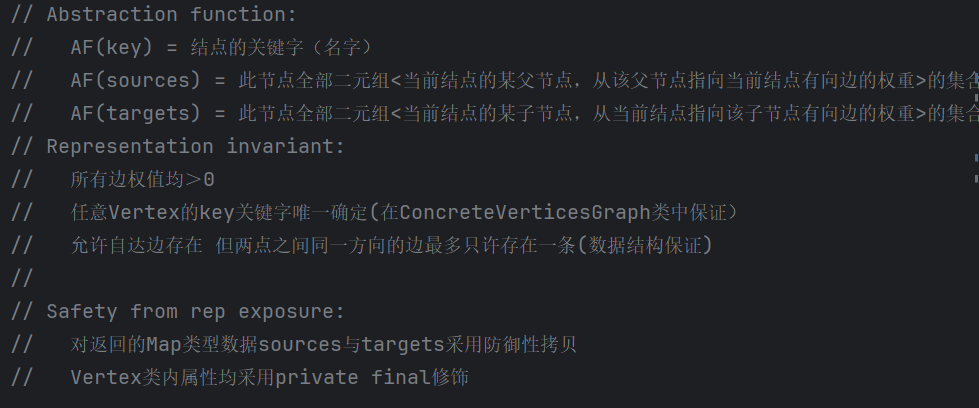
#### Implement ConcreteVerticesGraph

大致过程与ConcreteEdgesGraph类类似，先分析类的结构，大致绘制ConcreteVerticesGraph类与Vertex类的类图如下：



之后进行ConcreteVerticesGraph类与Vertex类的AF，RI及预防表示泄露措施说明的撰写（上下两张图分别对应ConcreteVerticesGraph类与Vertex类）：



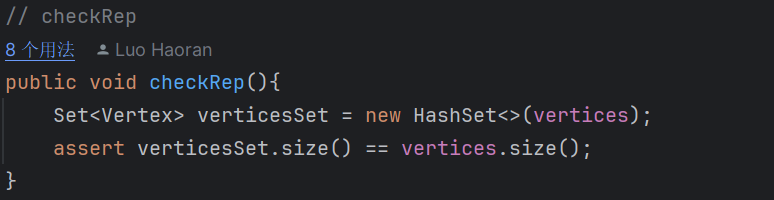


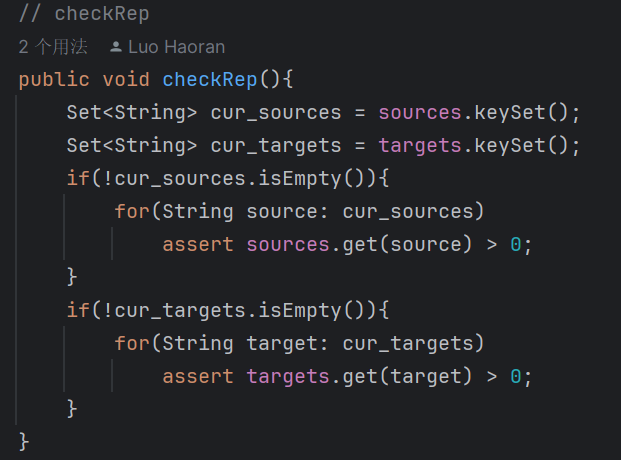
接着进行Vertex类的测试用例设计并将设计结果实现到test包中的ConcreteVerticesGraphTest类中，给出用例设计策略如下：





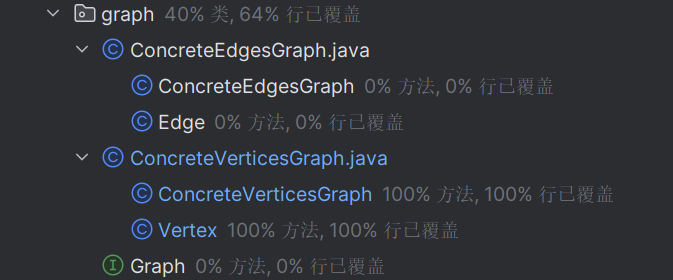
基于上面两个类中的RI要求，分别编写checkRep方法的实现，确保在ConcreteVerticesGraph类与Vertex类内执行各方法后的结果均满足表达不变性。





最后实现各个方法，对类内的属性与非Override的各个方法给出JavaDoc说明（见源码）。

完成代码实现后，同样地，使用覆盖率运行ConcreteVerticesGraphTest类，得到测试结果以及覆盖率状况如下：





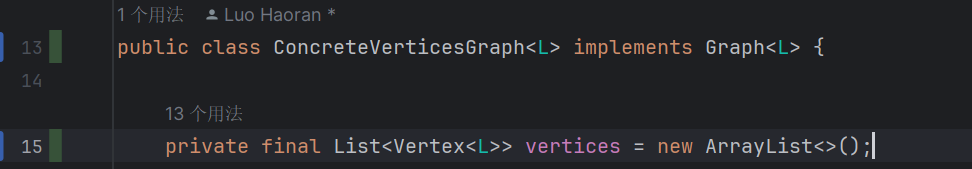
此时可以认为String类型下的Graph接口中的ConcreteVerticesGraph实现已全部完成。

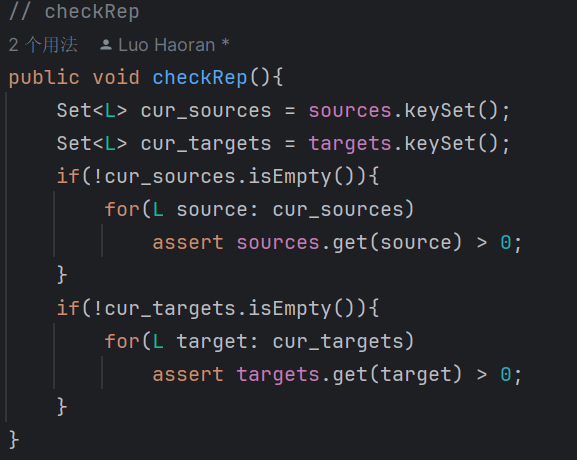
### Problem 3: Implement generic Graph<L>

现在根据实验要求，将原有的String数据类型下的代码扩展到泛型<L>，为后序的实际问题应用做准备。

#### Make the implementations generic

先将代码中全部用于标记数据类型的String替换为泛型的标识符L，同时为全部的Set和List添加泛型。





同时要注意为了避免测试类报错，在原本实现代码完成泛型替换后，相应地需要对测试类代码(ConcreteEdgesGraphTest与ConcreteVerticesGraphTest）进行修改处理。以ConcreteEdgesGraphTest类内的部分方法为例：

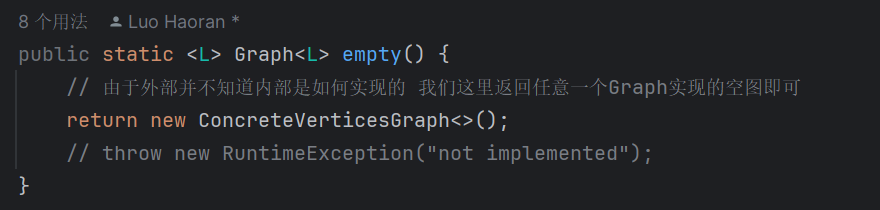


#### Implement Graph.empty()

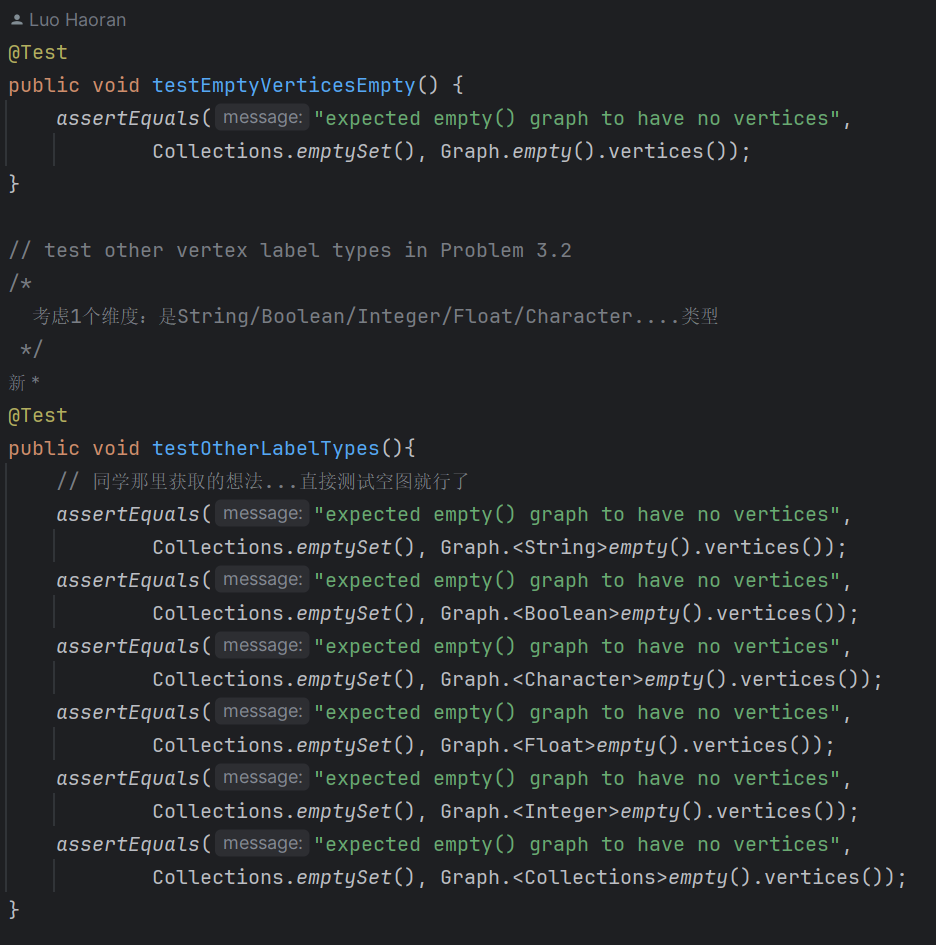
Graph接口中的empty方法是静态方法，是先于main方法加载而为所有对象共享使用的，常常通过类调用。我们将在test包下的GraphStaticTest中完成相应测试类的编写。

由于Graph的实现并不关心标签的实际数据类型，故我们不需要再针对String外的其他数据类型进行类似之前GraphInstanceTest里的复杂测试类设计，仅需要为确保泛型类的正常运行创建一些不同数据类型标签的简单测试类。

先在GraphStaticTest完成测试用例编写，再实现empty静态方法。显然外部并不知道内部的结构，故要返回一个empty的空图时，我们任意选取一种Graph的实现，然后返回这个实现下的一个空图即可。



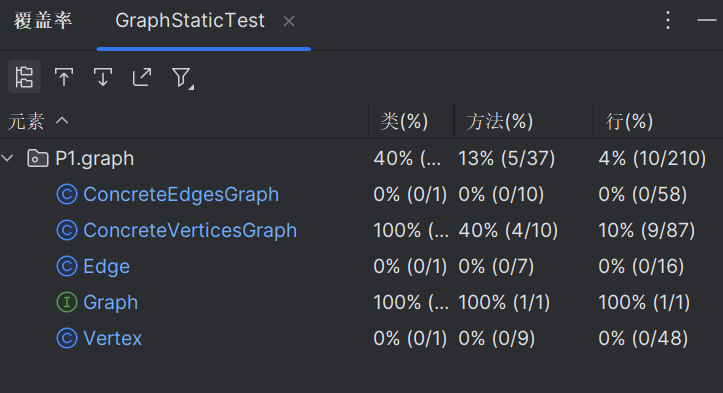
Empty方法实现



GraphStaticTest内部分测试方法

以覆盖率运行测试类，结果如下：





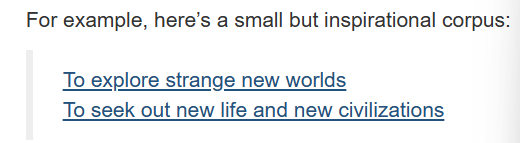
至此完成泛型<L>实现。

### Problem 4: Poetic walks

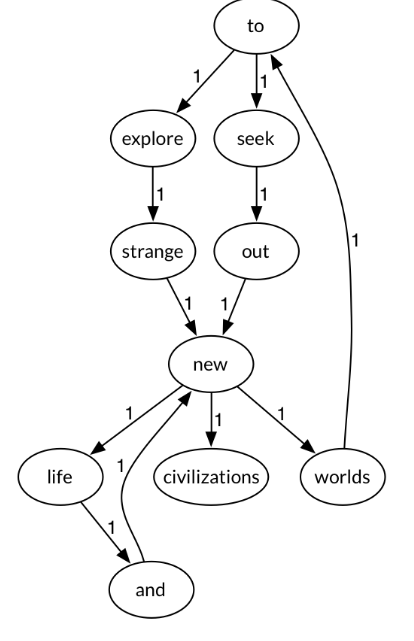
题目要求我们根据从文件流读入的英文语料库生成一个“单词亲和图”，并实现根据此“单词亲和图”对输入的任意字符串进行“扩充”的功能。

“单词亲和图”中各个顶点为大小写不敏感的单词，它的有向边则代表前一个单词与后一个单词的连接关系，此有向边的边权为前一个单词后面直接邻接后一个单词的情况出现的总次数。

以MIT页面的语料库——两行短诗为例：



可以生成如下“单词亲和图”：



如图“to”指向“explore”有向边边权为1表示在语料库中出现过一次“explore”紧跟“to”的情况，“explore”指向“strange”有向边边权为1表示在预料中出现过1次“strange”紧跟“explore”的情况，以此类推。

完成“单词亲和图”的建构后，我们要实现根据此“单词亲和图”对输入的任意字符串进行“扩充”的功能。具体来说是：提取输入字符串中两两相邻的英文单词（记为head和tail），并在“单词亲和图”中进行搜索，在搜索到图中存在恰好位于head与tail之间的顶点b时，将单词b插入原来字符串内的head与tail之间。

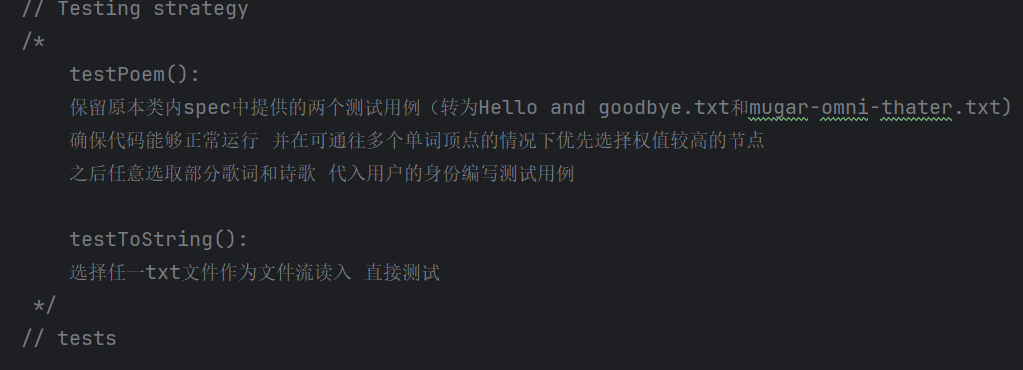
有如下规则：

* Head与tail之间必须恰好间隔一个顶点b，如果间隔更多顶点，则不在原字符串head与tail之间加入新的单词。
* 如果head与tail之间有多个“恰好间隔一个顶点”的情况，在其中选择这样的一个顶点b，使得在“单词亲和图”中head->b->tail的总权值大于其他的情况。
* 输出的字符串中单词大小写与标点符号不做任何编辑。

明确任务要求后，开始解决报告给出的各个问题。

#### Test GraphPoet

在GraphPoet中有poem和toString两个方法需要测试，给出如下测试策略：



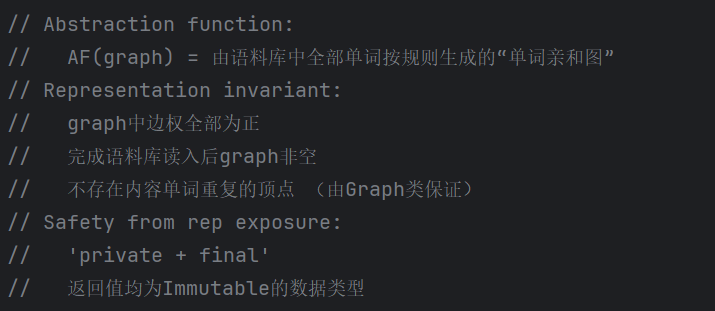
在poet目录中放入7个txt文件用作测试用例使用的语料库，编写测试方法testPoem1()~testPoem7()。

另外使用其中一个语料库文件验证toString方法的有效性，编写测试方法testToString（事实上这里Override的toString直接委派给graph对象执行了，相当于这个方法已经在Graph接口的测试类GraphInstanceTest.java中测试过，个人认为不再编写测试方法也是可行的，不过为了测试过程的规范性，这里仍编写testToString方法）。

#### Implement GraphPoet

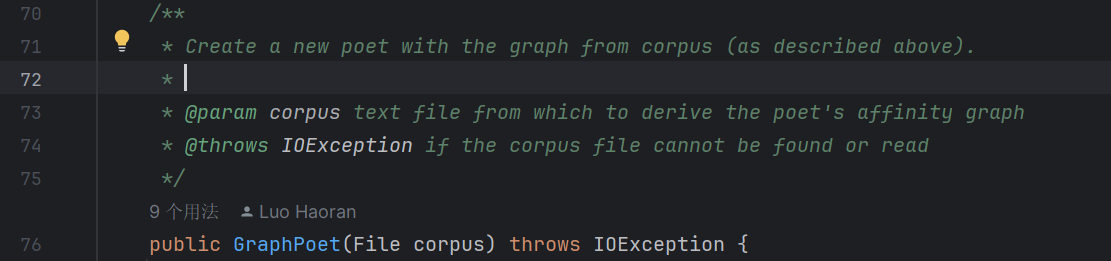
下面实现GraphPoet类。由于仅GraphPoet一个主要类，结构比较简单，不再绘制类图。

撰写AF，RI与表示泄露预防措施：

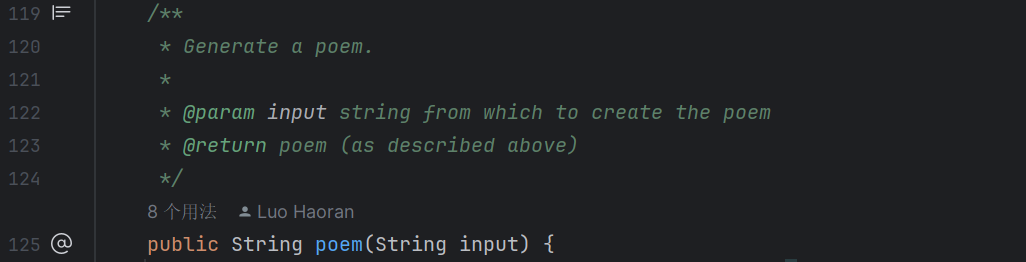


GraphPoet中包含一个Graph类对象graph，在实例化时即调用静态方法Graph.empty()而被初始化为空图。

在构造方法GraphPoet中，实现从文件流逐行读入语料库，并进行单词划分，单词转小写，剔除标点符号等操作，最后生成“单词亲和图”。



在Poem方法中，实现由已经生成“单词亲和图”，就给定的字符串输入input，在“单词亲和图”中按要求进行搜索匹配，并将全部符合要求的“bridge words”加入到input字符串中返回。

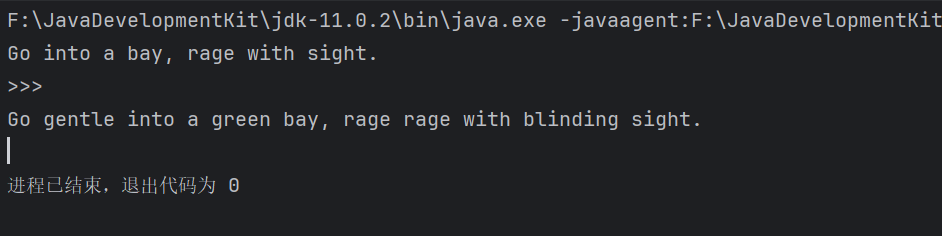


对重写的toString方法，由于我们只关心单词前后的邻接关系，而这恰好也能直接通过Graph接口下两个实现中的toString方法表示出来，故直接将其委派给graph对象执行即可。

#### Graph poetry slam

MIT页面要求我们更新Main类中main方法下的参数，对GraphPoet类尝试一些“很酷的例子”（本质上就是测试GraphPoet类的功能），这里我选取诗歌《不要温和的走入那个良夜》作为语料库，修改main方法后运行得到结果如下：

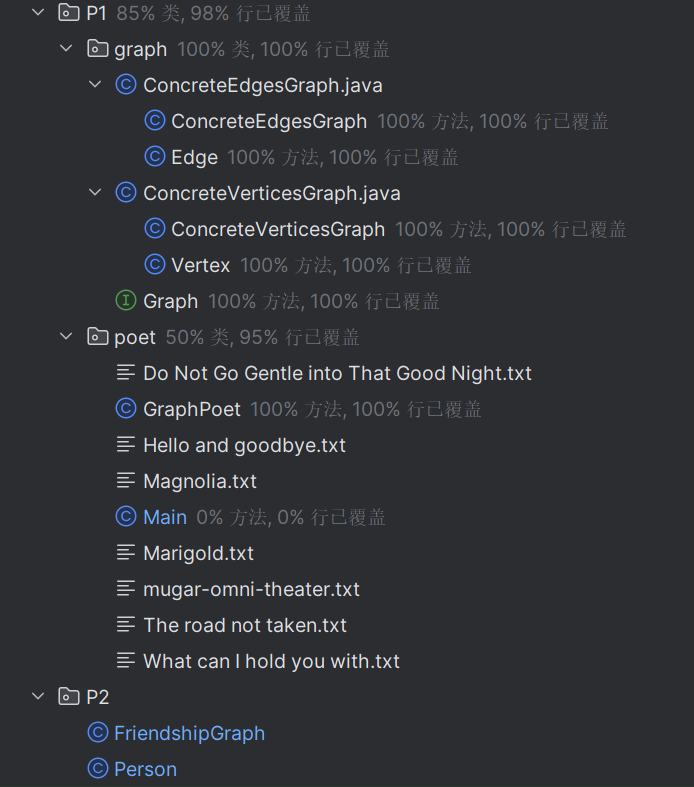




如果能找到一些庞大的语料库，或许这个例子会“更酷”？

### 使用Eclemma/Code Coverage for Java检查测试的代码覆盖度

在IDEA环境下直接使用覆盖率运行P1目录下的全部测试类，得到结果如下：



代码行整体的覆盖度基本达到了95%以上，可以认为测试是充分的。

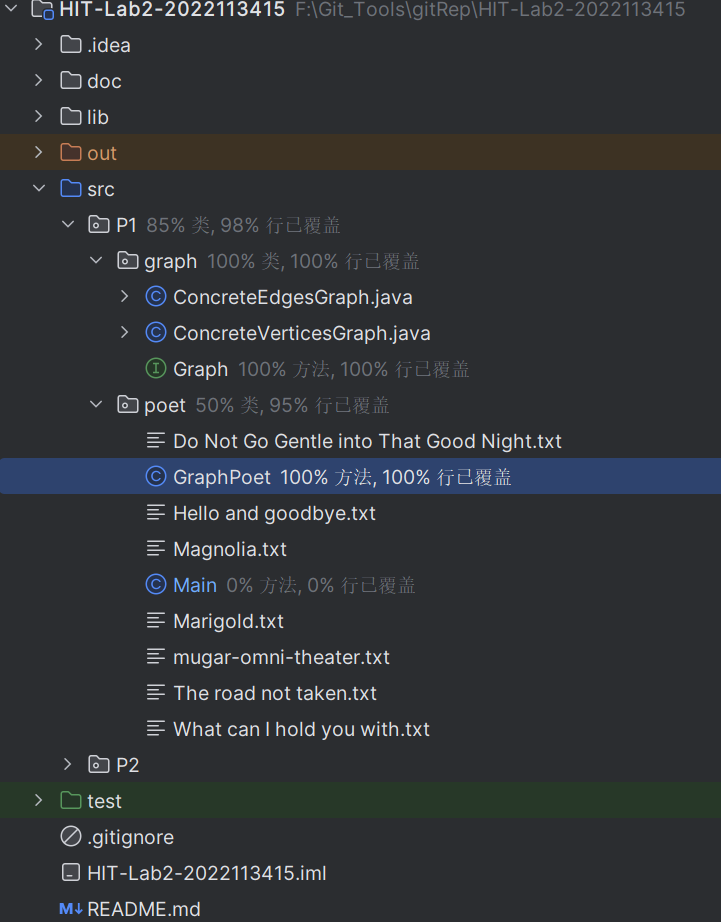
### Before you’re done

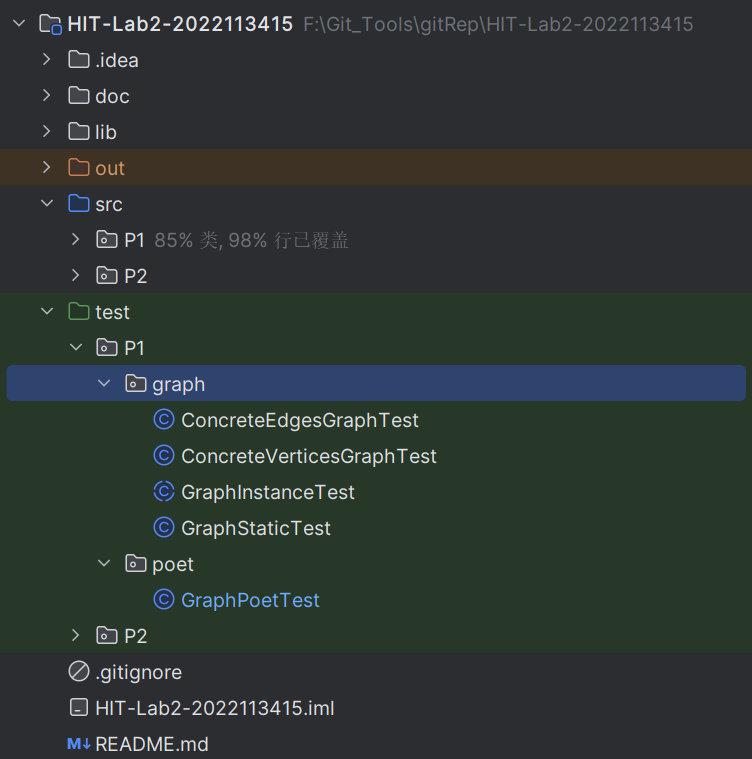
按照[http://web.mit.edu/6.031/www/sp17/psets/ps2/#before\_youre\_done](http://web.mit.edu/6.031/www/sp17/psets/ps2/" \l "before_youre_done)的说明，检查程序，在IDE界面并未提示项目文件中有任何由代码本身导致的Warning报错。

如何通过Git提交当前版本到GitHub上你的Lab2仓库。

启动Git Bash，和Lab1中的流程一致，git add各个需要上传的文件目录，git commit后git push即可。

在这里给出项目的P1目录结构树状示意图：





## Re-implement the Social Network in Lab1

Problem2要求我们基于在前面步骤中定义的Graph<L>及其两种实现，将泛型L替换为具体的Person类，按照Lab1中Social NetWork的要求，复刻Lab1 Problem 3中FriendshipGraph的各种功能，并且较为充分地复用我们在前文构造的类中已经实现的方法。最后，运行同样的main()，执行Lab1中的Junit测试用例，确保我们重新实现的Social NetWork能够正常运行。

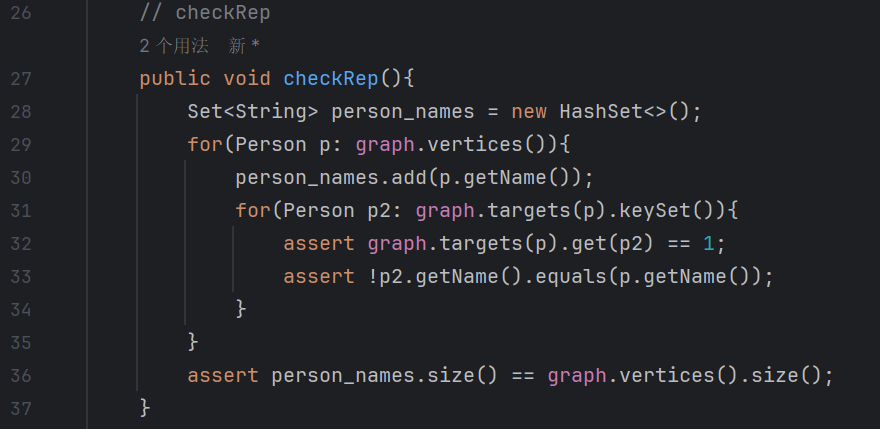
### FriendshipGraph类

FriendshipGraph类的属性中包含一个Graph接口下存储加入图中Person对象的ConcreteEdgesGraph类对象graph，用于存储关系图内的顶点与边的信息。

先撰写FriendshipGraph类的AF、RI及预防表示泄露措施如下：



根据如上内容，为FriendshipGraph类编写checkRep方法：



对类中方法说明如下：

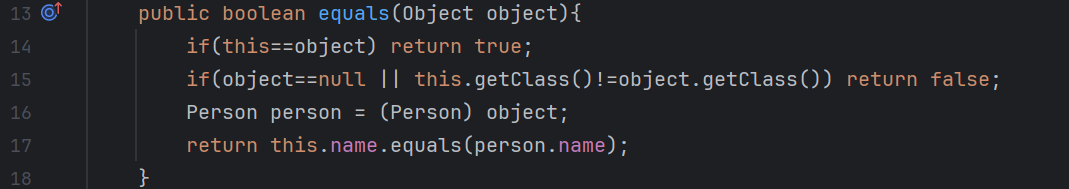
* Public Boolean addVertex(Person person)：该方法在社交网络图中增加一个新的节点，参数是要加入的Person类。方法先调用vertices方法获取graph中的全部Person对象，再将加入的person与这些Person对象逐一比对，在确认没有重复后添加person到graph中。在添加成功时返回true，否则返回false。实际调用graph对象add方法即可。
* Public Boolean addEdge(Person p1, Person p2)：该方法在社交网络图中添加两个Person对象之间的联系。程序考虑到了“单向社交的情况”，调用addEdge方法仅会在p1的属性child集合中添加p2，即仅添加在图中添加有向边<p1,p2> 。若为无向图，默认全部社交为双向，只需要再颠倒p1,p2位置调用一次方法即可。在边添加成功时，方法返回true，否则返回false。实际调用graph对象set方法即可
* Public int getDistance(Person p1, Person p2): 计算任意两个Person之间的“距离”，若没有任何社交关系则输出“-1”。方法具体实现基于BFS算法，借助graph对象的targets方法，默认边权为1，在搜索到边时将准备返回的值加1即可，搜索到目标点时退出。

### Person类

Person类用于实现一个具体的人——Person对象——FriendshipGraph类图中顶点的对应。

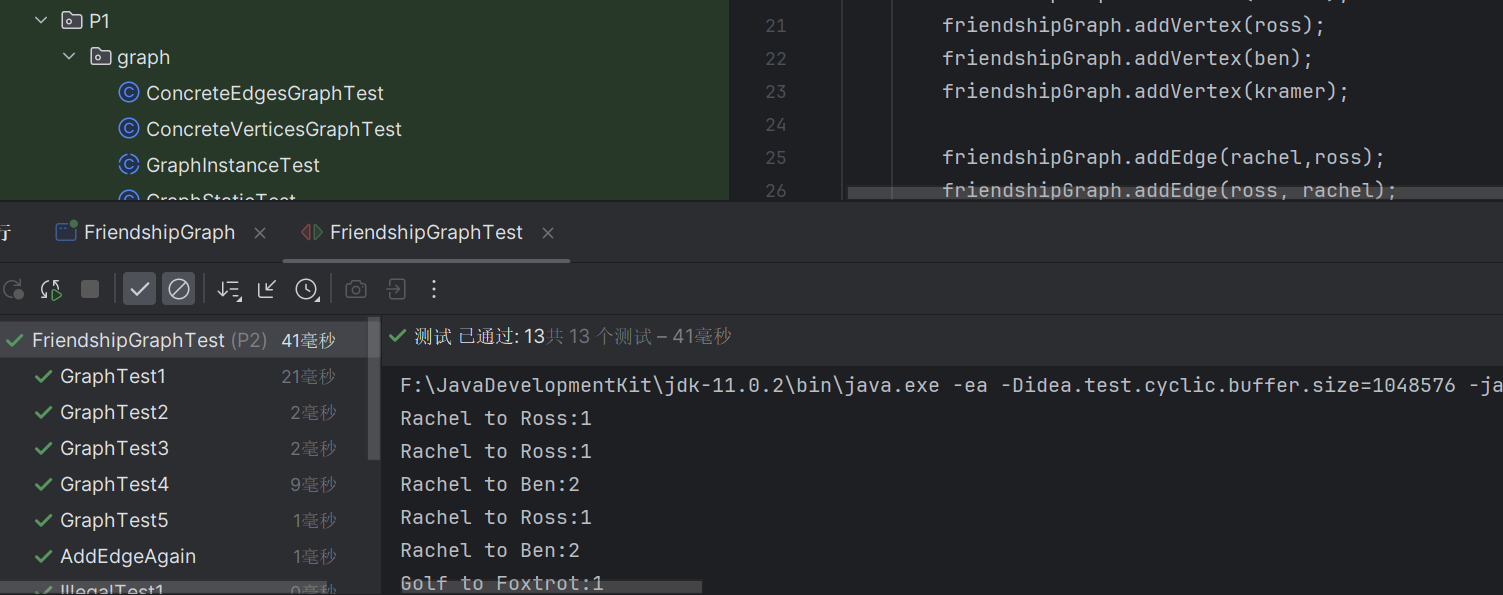
Person类较Lab1不同：Lab1中实验者将Person在关系图中的邻接信息存储于Person类内，这里则放在了FriendshipGraph类的graph对象中，因而需要对Person类进行修改，删去原有的ChildList属性，generateChildList方法等。现在Person类内仅包含一个存储Person对象名称的字符串name，Person类的构造方法以及获取Person对象name的方法getName。

值得注意的是，在FriendshipGraph中涉及Person类的比较，需要对equals方法进行重写:



### 测试用例

在test/P2/FreindshipGraphTest.java测试类中，实验者将Lab1中编写的13个对FriendshipGraph类的测试用例直接复制了过来，运行结果如下：



方法GraphTest1到GraphTest4用于测试大/小规模稀疏/稠密图中任何重复异常输入时两顶点之间距离＞0，距离＝0与不连通的情况。由于两顶点之间距离＞0，距离＝0与不连通的情况可以在同一个测试方法内验证，故不再分别独立设计测试方法。

GraphTest5方法测试特殊的所有顶点成环的情况，以验证BFS算法的准确性。

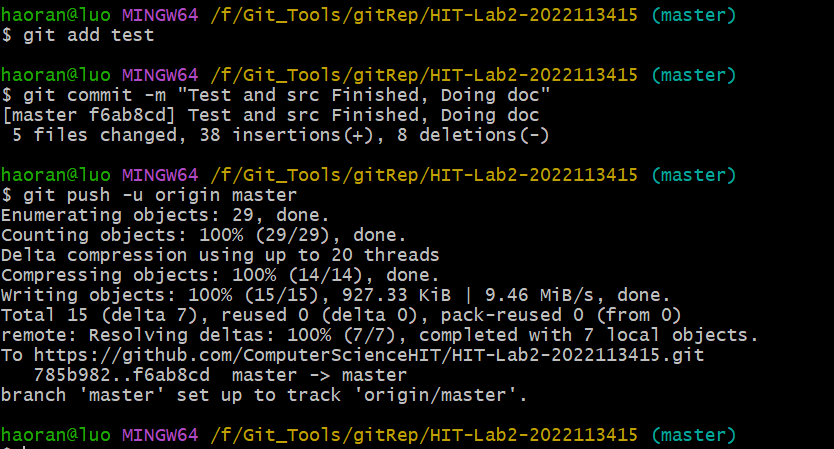
方法IllegalTest1到IllegalTest6分别用于测试有/无重复Person对象名字，大/小规模图，有/无自连边的情况。这对应着“有/无重复Person对象名字”，“大/小规模图”，“有/无自连边”三个维度的笛卡尔乘积中的六个情况，最后两种情况与GraphTest1到GraphTest4中“大/小规模图 无任何重复异常输入”的情况重合，故不再单独设置Junit测试用例。

最后两个测试方法DirectedTest与AddEdgeAgain的性质相对前面11个测试方法较为特殊：DirectedTest用于检测调用addEdge方法而只添加单向边时getDistance方法的运行效果，而AddEdgeAgain则用于验证重复添加边并不影响FriendshipGraph类的运行结果（期望addEdge方法在重复添加边时返回正值）。

客户端main函数与Lab1一致，这里不再赘述。

### 提交至Git仓库

与之前一致，在git bash内add全部项目文件再git commit与git push即可。



# 实验进度记录

使用表格方式记录你的进度情况，以超过半小时的连续编程时间为一行。

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 4.19 | 14:00-22:00 | 阅读MIT页面内容，编写Graph接口下测试策略及各个测试用例。 | 按时完成，编写了ConcreteEdgeGraph实现的部分代码 |
| 4.20 | 10:00-17:00 | 编写完成在String数据类型下Graph接口的两种实现的代码 | 按时完成 |
| 4.20 | 19:00-23:00 | 完成泛型落实，撰写报告前半部分未实现泛型前的内容 | 未及时完成，泛型落实出现漏洞 |
| 4.21 | 08:00-20:00 | 完善泛型落实，编写相关测试用例并解决PoeticWalks问题 | 按时完成 |
| 4.22 | 18:00-21:00 | 完成P2 | 提前半小时完成 |
| 4.24 | 21:00-23:00 | 最后优化原有代码，撰写报告最后内容 | 报告滞后完成 |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| P1测试用例编写量过于庞大 | 核查等价类划分，发现错误的额外划分了“边权输入小于0”的等价类——这部分是spec规定之外的，已经由外部保证并不会出现负边权输入，故删除，降低了测试用例编写量。另外多次复用了先行编写的测试用例代码。 |
| PoeticWalks问题中从文件流读入后，不知道怎样安排才能又按小写读入单词，又能在最后输出时保留原输入中的大小写与标点字符。 | 在方法内预留一份原字符串分割后保留大小写与标点符号的拷贝，在最后输出时，用StringBuiler类append方法添加这部分内容，并在其中插入满足条件的小写字母单词。 |
| 忘记重写Override方法，导致在P2实现中对比Person类时出现错误 | 研究原有的Override的代码后在Person类中重写此方法，使name成为判断Person对象是否相等的标识。 |
| 对如何撰写抽象函数，表示无关性，checkRep感到困惑，无从下手 | 查看习题课提供的代码，回顾PPT内容，查询CSDN、博客园等平台上学长们做实验的心得体会，加深理解。 |

# 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（必答）

* 测试用例的优先级一定要高于各个方法实现代码的编写，这能够避免编写代码后编写测试用例时先入为主，保证编写测试用例的客观性。
* 尽管目前的实验项目结构还较为简单，模型图的功效也不可忽略。本次实验在ConcreteEdgesGraph和ConcreteVerticesGraph的实现中编写了简单的UML模型类图，为代码的实际编写提供了一定的指导作用。
* “多搜”——从博客、Github、官方Doc文档等多种途径获取可能需要的类、数据结构和方法，为自己的代码编写提供指导帮助。
* “别拖”——本次实验从4月19日开始才着手进行，如果没有截止日期的顺延，此实验内容的完成度恐怕很难保证。在之后的学习中仍需要我提前做好时间规划，合理安排学习时间。

## 针对以下方面的感受（必答）

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？

面向ADT编程与面向应用场景编程有相当大的差异。

面向ADT编程要求对编写的程序整体有充分的认识，要考虑接口、抽象类、类的整体设计，考虑哪些部分能够复用，哪些部分才需要具体实现。优秀的面向ADT编程设计可以节省大量时间，甚至能够为以后的项目复用，长期来看是非常有帮助的。

而面向应用场景编程则更“就事论事”、具体而微，针对具体的情景具体的数据结构编程思路显然更为简单，但是工作量大，复用性低也是其不可回避的缺陷。

1. 使用泛型和不使用泛型的编程，对你来说有何差异？

使用泛型的编程在编写时比较折磨，需要确保调用的方法与泛型的数据类型没有依赖关系，复杂度较不适用泛型的编程要搞，但在复用时也更加方便，同样是一个“渐入佳境”，先难后易的过程。

不使用泛型的编程复杂度较低，时间成本更低，但面向的实际应用场景也更为狭隘。

总的来说，个人认为在实际应用中，要尽可能多的使用泛型进行编程，以适配更多的应用场景。

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？

测试用例的优先级高于代码实现的编写，优势在于能够避免编写代码后编写测试用例时先入为主，排除已有代码的干扰，保证测试用例的客观性有效性。

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？

提高代码的使用效率，减少重复编写，节省程序员的精力。

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？

对自己和其他可能的代码使用者起到提醒的作用，为个人编写代码与其他人复用代码提供指导;能够提高代码的安全性，预防外部客户端可能的错误操作对内部代码的破坏；checkRep在某种程度上也能够定位漏洞，在实现代码中出现问题导致数据结构不符合RI时及时报错；等等。

个人目前并不习惯为ADT撰写specification, invariants, RI, AF，时刻注意ADT是否有rep exposure等操作，但会尝试在之后的OOP中坚持。

1. 关于本实验的工作量、难度、deadline。

个人认为实际工作量并不算特别大，在实验过程中我的时间更多花费在AF、RI等的撰写，测试用例编写，MIT页面题意的理解，Javadoc的查询上，这些实验者眼中的“难点”更多是由于个人长期未阅读英文页面以及OOP熟练度不足导致的。

Deadline…..24号上传的最后版本，不做评价 :-)

1. 《软件构造》课程进展到目前，你对该课程有何收获和建议？

好消息：课程干货很足

坏消息：太足了，要撑死了

明显感觉到课堂知识密度过大，感觉哪里都是重点，记笔记无从下手，甚至在如此大的课堂内容密度下PPT中还有需要课后自己学习的考试内容，恐怖如斯。个人希望此课程能在之后增加学时，或者利用实验课的空余时间，对课堂知识做出补充讲解。