

**2024年春季学期  
计算学部《软件构造》课程**

**Lab 2实验报告**

|  |  |
| --- | --- |
| 姓名 |  |
| 学号 |  |
| 班号 |  |
| 电子邮件 |  |
| 手机号码 |  |

**目录**

[1 实验目标概述 1](#_Toc97648181)

[2 实验环境配置 1](#_Toc97648182)

[3 实验过程 1](#_Toc97648183)

[3.1 Poetic Walks 2](#_Toc97648184)

[3.1.1 Get the code and prepare Git repository 2](#_Toc97648185)

[3.1.2 Problem 1: Test Graph <String> 2](#_Toc97648186)

[3.1.3 Problem 2: Implement Graph <String> 7](#_Toc97648187)

[3.1.3.1 Implement ConcreteEdgesGraph 7](#_Toc97648188)

[3.1.3.2 Implement ConcreteVerticesGraph 14](#_Toc97648189)

[3.1.4 Problem 3: Implement generic Graph<L> 19](#_Toc97648190)

[3.1.4.1 Make the implementations generic 19](#_Toc97648191)

[3.1.4.2 Implement Graph.empty() 20](#_Toc97648192)

[3.1.5 Problem 4: Poetic walks 21](#_Toc97648193)

[3.1.5.1 Test GraphPoet 21](#_Toc97648194)

[3.1.5.2 Implement GraphPoet 21](#_Toc97648195)

[3.1.5.3 Graph poetry slam 23](#_Toc97648196)

[3.1.6 使用Eclemma检查测试的代码覆盖度 24](#_Toc97648197)

[3.1.7 Before you’re done 25](#_Toc97648198)

[3.2 Re-implement the Social Network in Lab1 25](#_Toc97648199)

[3.2.1 FriendshipGraph类 26](#_Toc97648200)

[3.2.2 Person类 27](#_Toc97648201)

[3.2.3 客户端main() 28](#_Toc97648202)

[3.2.4 测试用例 28](#_Toc97648203)

[3.2.5 提交至Git仓库 29](#_Toc97648204)

[4 实验进度记录 29](#_Toc97648205)

[5 实验过程中遇到的困难与解决途径 30](#_Toc97648206)

[6 实验过程中收获的经验、教训、感想 30](#_Toc97648207)

[6.1 实验过程中收获的经验和教训（必答） 30](#_Toc97648208)

[6.2 针对以下方面的感受（必答） 30](#_Toc97648209)

# 实验目标概述

根据实验手册简要撰写。

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

本次实验由于使用 IntelliJ Idea，含有附带了 Code Coverage test for Java 插件，所以不需要特地配置插件，只需要和 Lab1 一致引入JUnit4 的 jar 包即可。

在这里给出你的GitHub Lab2仓库的URL地址（HIT-Lab2-学号）。

<https://github.com/ComputerScienceHIT/HIT-Lab2-2022112831>

# 实验过程

# 请仔细对照实验手册，针对三个问题中的每一项任务，在下面各节中记录你的实验过程、阐述你的设计思路和问题求解思路，可辅之以示意图或关键源代码加以说明（但千万不要把你的源代码全部粘贴过来！）。

## Poetic Walks

# 在这里简要概述你对该任务的理解。

Poetic Walks 这一实验束腰要求我们掌握详细的 ADT 设计和 OOP 的面相对象编程方法，使用 OOP 来完成对 ADT 的设计，并加以书写详细的 SPEC，包括 AF、RI、Rep Exposure，以及对每个方法详细的注解，并且在过程中增加对表示独立性、表示不变量、检查表示不变量等有了更加详细深刻的认识，也基本掌握了 ADT 设计的基本的抽象过程和设计流程。也大量运用了 Java 标准的 Collections 类。测试类的书写也应当根据 Rep 的设计分出不同的覆盖方法，总得来说SPEC的设计相当的费时费力，以及测试的书写和覆盖也不失为一件美事。

### Get the code and prepare Git repository

# 如何从GitHub获取该任务的代码、在本地创建Git仓库、使用Git管理本地开发。

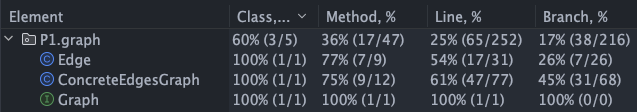
从<https://github.com/rainywang/Spring2022_HITCS_SC_Lab2/tree/master/P1>将仓库克隆到本地，并且将文件复制到本地的 Lab2 的目录下。

本地的 Lab2 目录是从远程克隆下的，所以已经有了远程的 remote origin master 分支。 只需要整成进行 git 的版本管理操作即可。常用 git add . ｜ git commit -m“备注” ｜ git push 这三个命令就可以完成。

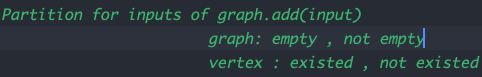
### Problem 1: Test Graph <String>

# 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。

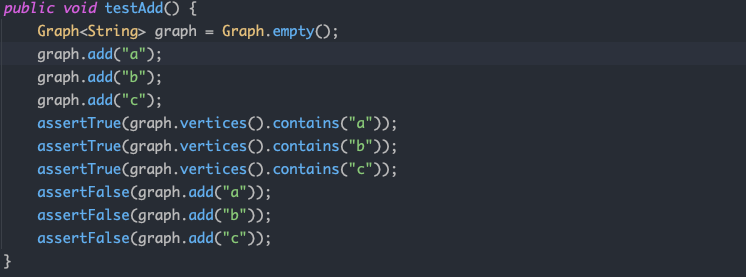
下面是代码的覆盖度测试结果：



#### addTest

设计思路：  
 

实现：



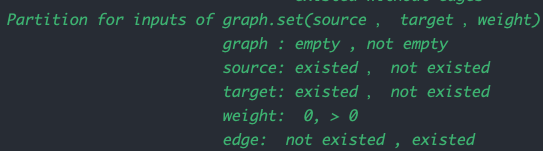
结果：

true 如果添加成功

false：如果添加失败，可能由于参数为空、格式不符或者已经添加过。

#### setTest

设计思路：



实现:



结果：

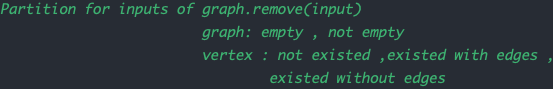
0 ：说明在原图中source 和 target 之间不存在边。若weight > 0 ，则在source 和 target 之间添加一条权重为weight的边；若weight = 0 ，

正数 ：说明在原图中source 和 target 之间存在边，返回值即为原来的边的权重。若weight > 0 ,将原来的边的权重修改为weight ；若weight = 0 ，删去这条边。

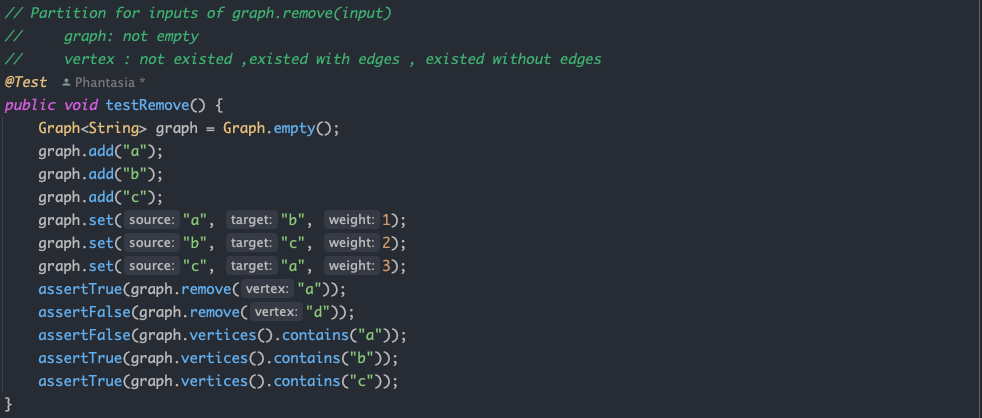
异常：可能由于 source 或者target 为 null 或者不符合泛型设置的格式，或者 weight 为负数。

#### removeTest

设计思路：



实现：



结果：

true ：删除点成功，如果这个点原来有边相邻接，所有与这个点邻接的边全部删去。否则只删去该顶点。

false：删除顶点失败， 说明原图中不存在该顶点。顶点集不变。边集不变。

#### verticesTest

设计思路：



实现：

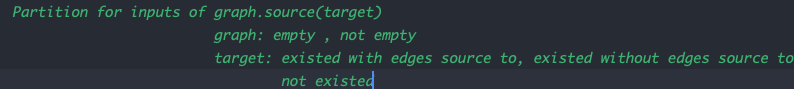


结果：

一个Set，包含所有的顶点；如果为空，则返回一个空的 Set。

#### sourcesTest

设计思路：



实现：

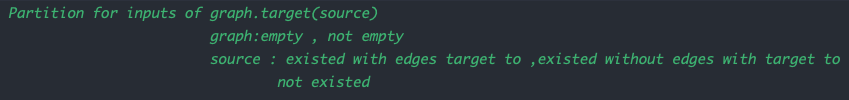


结果：

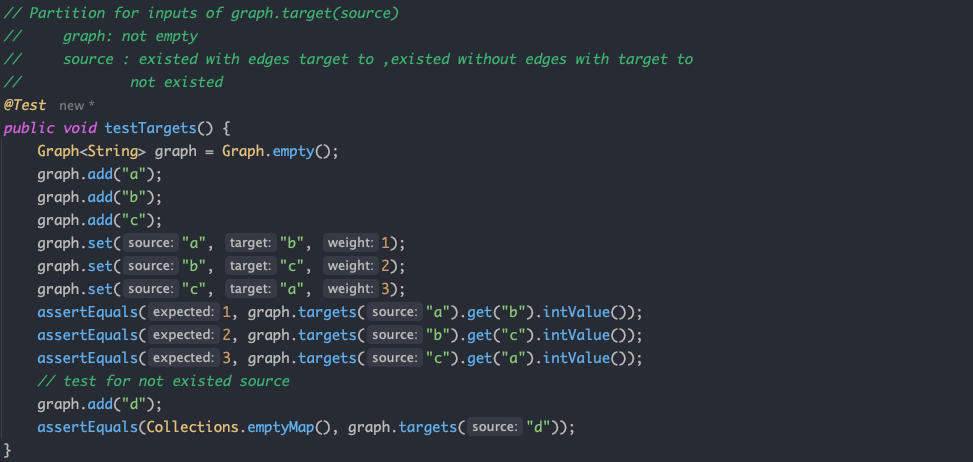
返回一个 Map 包含所有指向target 的顶点集以及边的权重；如果sources 为空，则返回空Map；

#### targetsTest

设计思路：



实现：



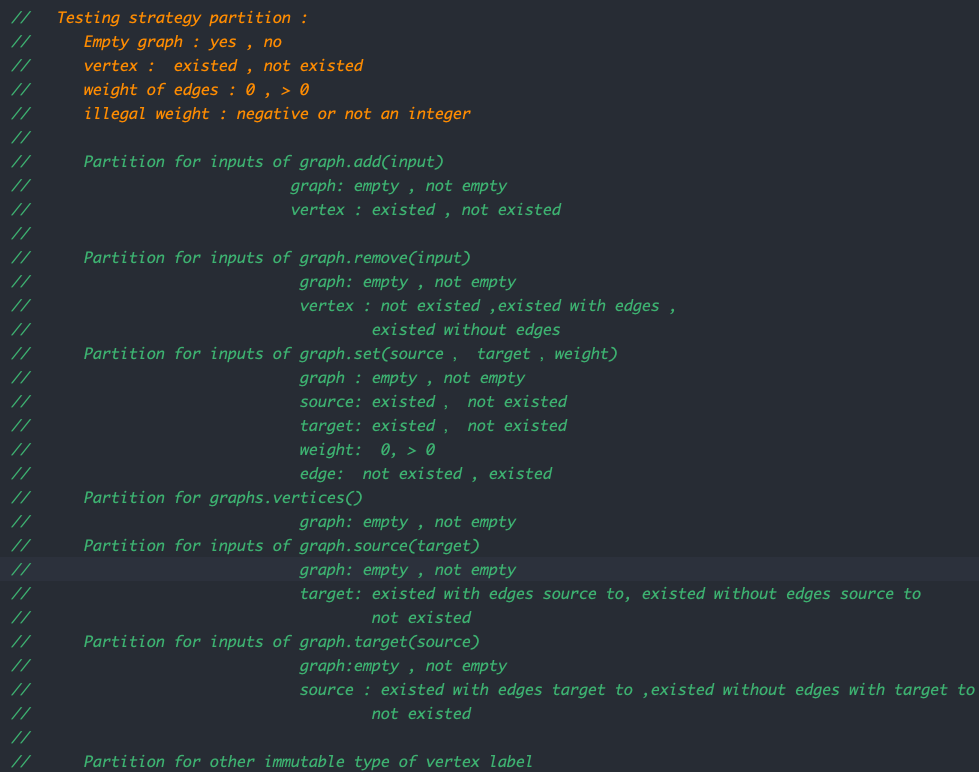
结果：

返回一个 Map 包含所有source 指向的顶点集以及边的权重；如果targets 为空，则返回空Map；

### Problem 2: Implement Graph <String>

# 由于 GraphInstanceTest 的测试在不完成 Graph 的条件下是无法执行的，所以在这里会提起 GraphInstanceTest 的设计，而不是 3.1.2 节。

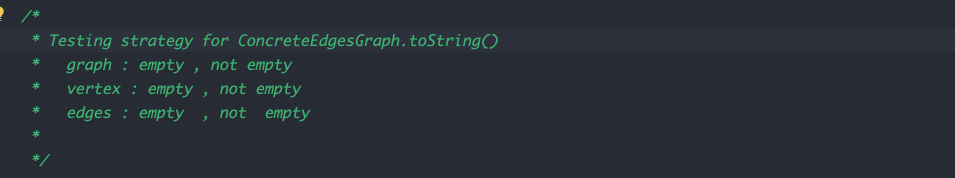
具体设计思路如下图所示，实现过程和 GraphStaticTest 类似，故不再赘述。



#### Implement ConcreteEdgesGraph

根据测试优先的设计原则，我们首先完成对该类极其内部 Edge 类的测试的设计。由于其他的测试可以通过调用 GraphStaticTest 和 GraphInstanceTest 中写好的测试样例进行测试，ConcreteEdgesGraph 类的测试只需要对 toString 方法进行测试即可。

**ConcreteEdgesGraph的toString()方法的具体测试策略如下：**



在实际测试过程中，也进行了对多种情况的覆盖。

覆盖 1：

graph: not empty

vertex: not empty

edges: not empty

实现：



结果：

在符合覆盖的情况下的输出是一种在 toString 方法中实现的固定格式的输出，为了保证输出的一致性，我对输出的结果按照字典序排序，避免因为打印顺序的问题造成的结果的不一致，而实际上包含的边集、点集完全一致。

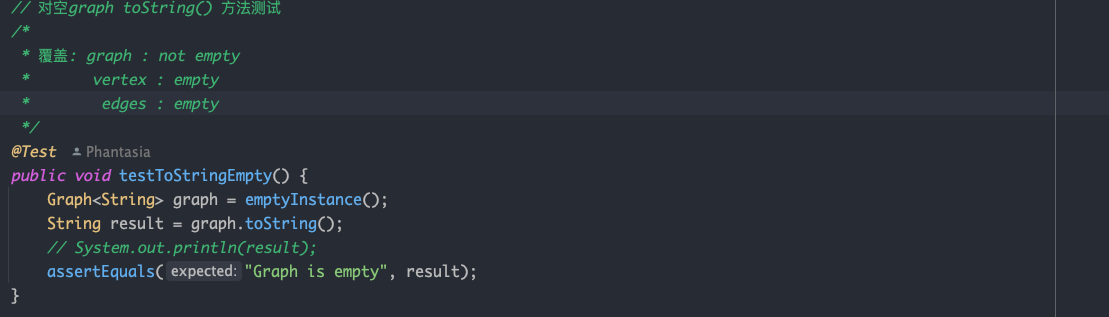
覆盖 2：

graph: not empty

vertex: empty

edges: empty

实现：



结果：

由于 graph 在初始化的时候就已经创建，所以 graph 本身在内存中不会为空，而 graph 中含有的 vertex 和 edges 集合，由于没有参数，导致实际上表现出的 graph 是一个空的结果。所以打印“Graph is empty”。

覆盖 3：

graph: not empty

vertex: not empty

edges: empty

实现：

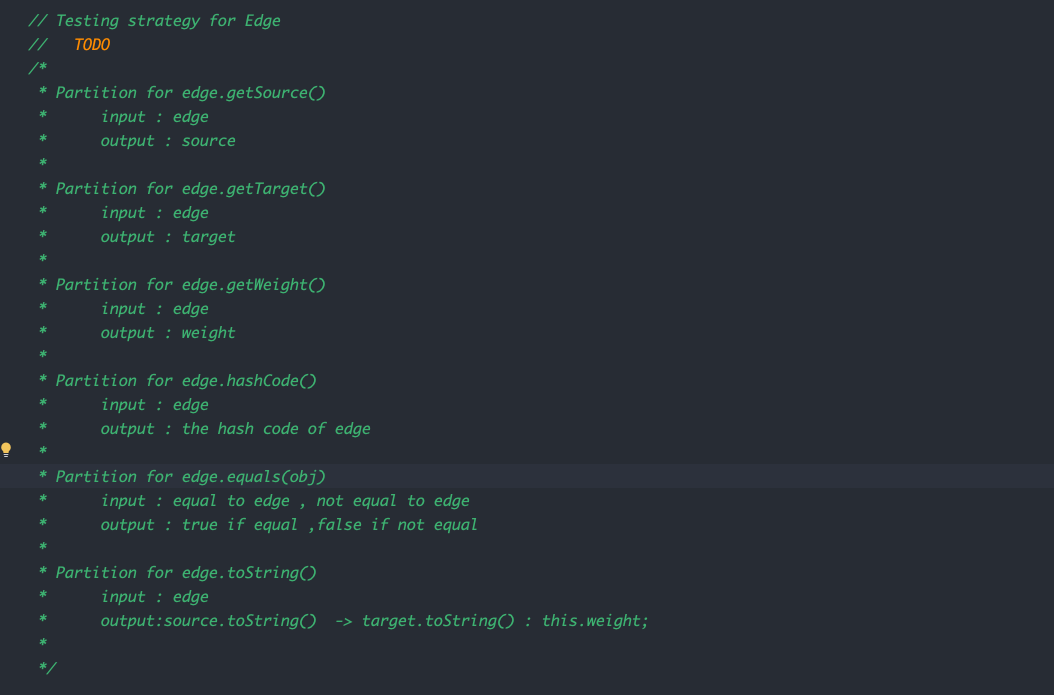


结果：

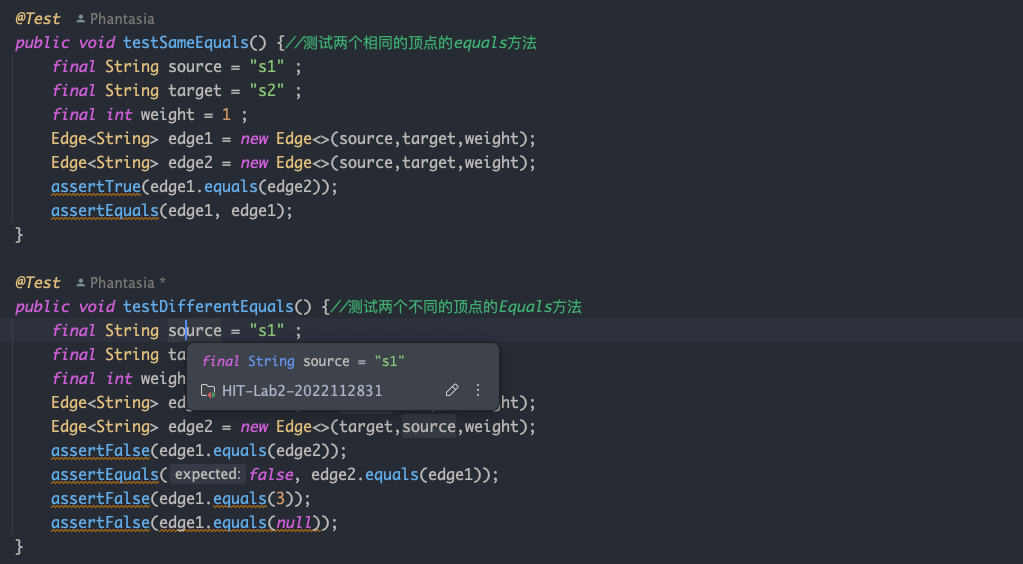
在 graph 和 vertex 不为空的情况下，由于图中没有边，所以仍然不会有实质性的 toString 结果返回，返回了一个提示信息“Graph is not empty，but no edges”。

**其次是进行内部类 Edge 的测试。**

具体的测试策略如图所示:



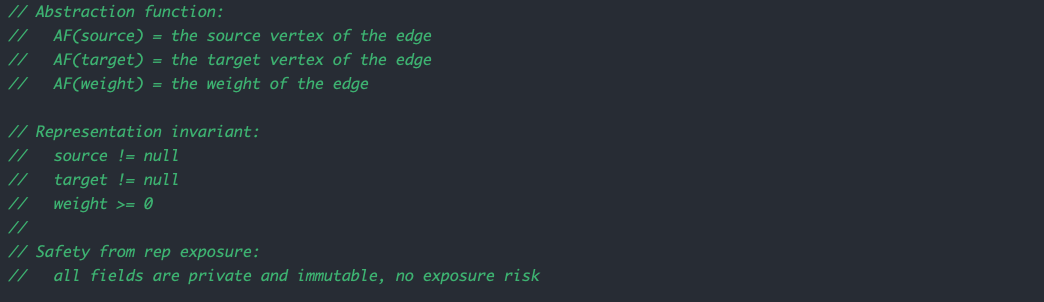
由于大多数方法并没有输入，所以并不需要对输入参数的 Partition，测试用例的设计也较为简单，唯一的 equals 方法含有一个 object 类型的参数，需要对其进行类型的 partition，比如 null、非 edge 类型、edge 类型，并且对于两个同样的source target weight 初始化出的不同地址的边或者是不同的 source target weight 参数的边的输入。其他方法的测试设计不再叙述，equals 测试实现如下：



**接下来是对应的具体的 ADT 实现，首先是对内部类 Edge 的实现**

1. 不变量设计

每个边都应该有的对应的起点和终点，以及对应的权重，Rep 应当是不可更改的和不可被外部访问的，所以均设置为private final类型。综上，所以可以根据边的不变量可以做出下面的 AF、RI、Safety from rep exposure 的设计。

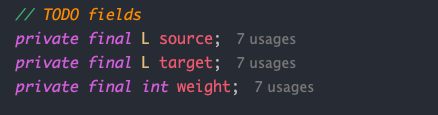


1. 检查不变量设计

根据 RI 中给出的条件可以写出 checkRep() 方法



1. Edge 类的具体实现

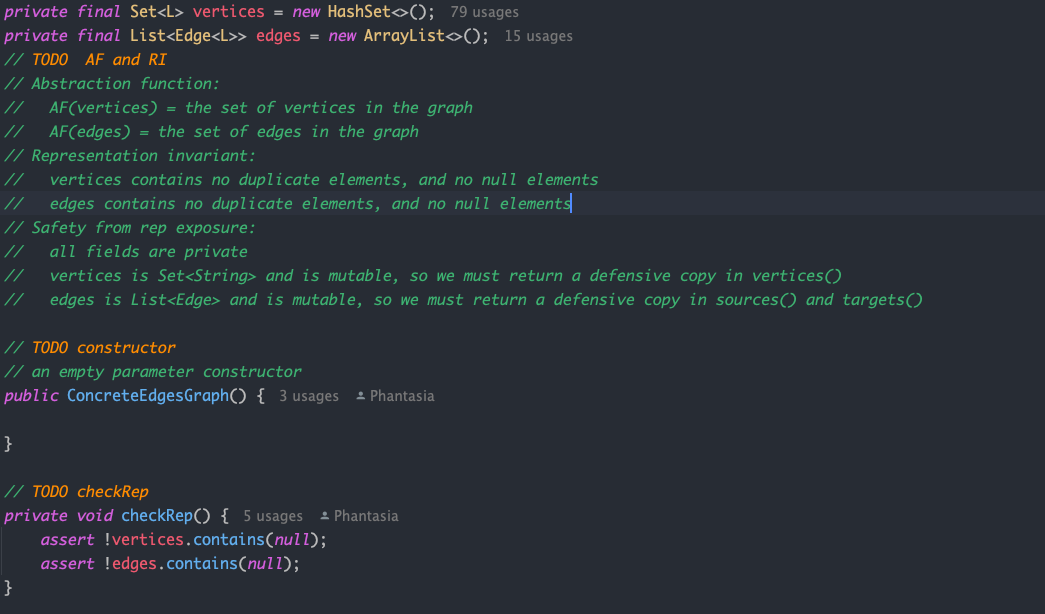
Rep 实现：  
 

其他Methods 实现：

|  |  |
| --- | --- |
| getSource() | 获得边的起点 |
| getTarget() | 获得边的终点 |
| getWeight() | 获得边的权重 |
| hashCode() | 重写 hashCode 方法 |
| equals() | 重写 equals 方法 |
| toString() | 重写 toString 方法 |

**接下来是对类 ConcreteEdgesGraph 的实现**

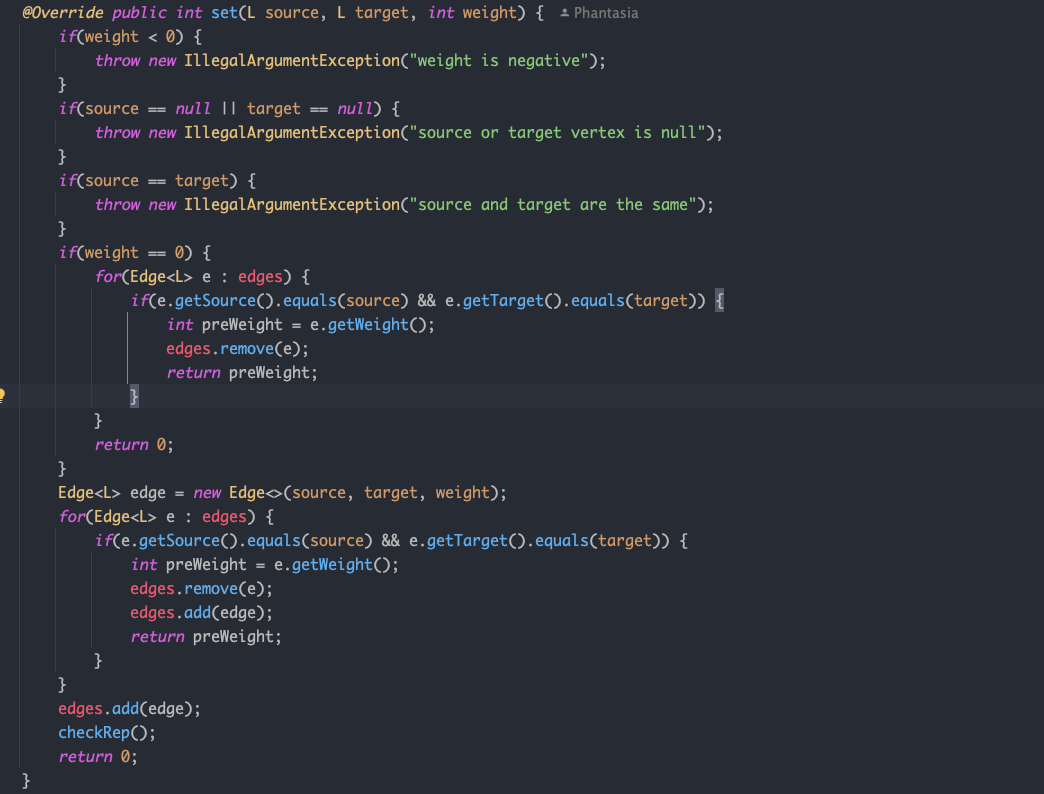
首先是 Rep 的设计，实验代码中规定了需要使用的 Rep，所以vertices 集合映射到的是 Graph 的点集，edges集合映射到的是 Graph 的边集，并且应当是私有且不可变的，对 mutable 类型的 vertices 和 edges 将会提供具体的 mutator 方法来对这两个集合进行更改。AF、RI、constructor、checkRep 的设计如下：



接下来是具体的成员方法的实现，部分方法实现较为简单，仅作实现逻辑的介绍，首先是 add() 方法，传入一个名为 vertex 的参数，用于添加点进入图中，如果 vertex 是不合法的值 或者 vertex 已经存在于点集中，则返回 false。vertices() 和 edges() 方法通过 defensive copy 返回一个原来集合的拷贝即可，实现简单。sources() 和 targets() 方法的实现需要对 edges 数组进行遍历，分别寻找到所有的以传入的 vertex 为 target 的边和 source 的边，并且设置一个Map，key 为 vertex 的 label，value 为对应的边的 weight。

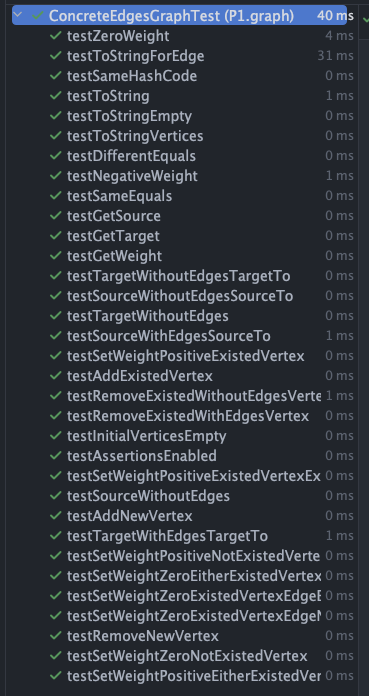
set() 和 remove() ,方法实现细节较多，此处详细介绍一下。

首先是 set 方法，用于向 edges 数组中添加对应或者修改的边，返回边原有的权值，如果不存在则返回 0。更改前并且要处理一定的不合法输入，比如权值为负数，source 或 target 为 null，亦或 target 和 source 一致，上面三个不合法输入都会抛出不合法参数的异常，而 weight 为 0 的情况则设置为删除当前边，但删除前需要确定当前边存在于 edges 中，否则 return 0 返回。在判断当前 edge 没有在 edges 中出现后，就可以添加 edge 到 edges 中，否则更新权值。实现如下：



而 remove 方法在删除成功后会返回 true，失败则会返回 false，首先需要处理不合法的情况，也就是传入的 vertex 的 label 为 null，会抛出不合法参数的异常，并且如果 vertex 不存在于 vertices 中，将会返回 false，更加要注意的是，删除点后还要删除对应的以此为 source 和 target 的边，可以使用 lamboda 表达式在 edges 中进行删除。实现如下：

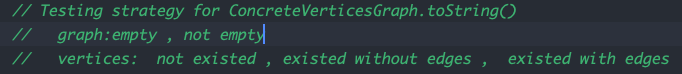
综上，ConcreteEdgesGraph 类就完成差不多了，后续再对代码进行测试，并且测试代码覆盖度（由于设计了一些原本方便操作的内部实现的 private 方法，后续弃用，导致 method 覆盖不完整）：





#### Implement ConcreteVerticesGraph

仍然应当首先进行测试用例的编写。和 ConcreteEdgesGraph 一样，由于一般情况下的测试在 GraphInstanceTest 中已经完成，所以只需要写 toString() 方法的测试。测试策略如下：



具体的实现过程和 ConcreteEdgeGraph 类十分相似，所以在这里不再赘述。

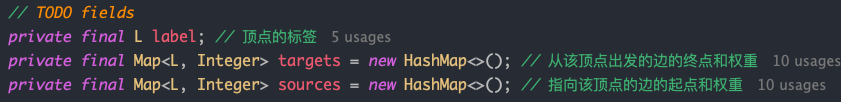
接下来是对内部类 vertex 的测试，下面是具体的测试策略：



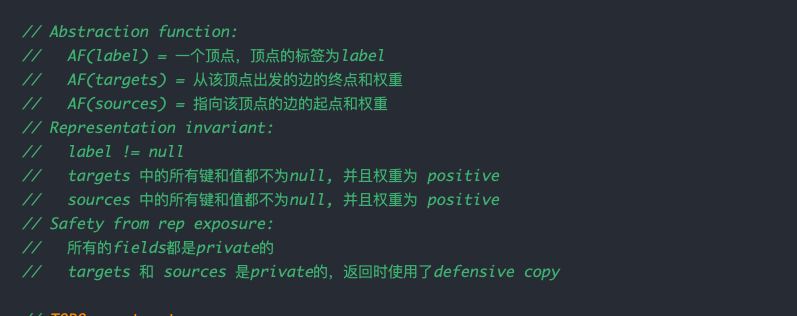
测试代码实现较为容易，不需要详细介绍，根据 Partition 的描述分别书写即可。

**下面是两个类的具体实现，首先是内部类 Vertex。**

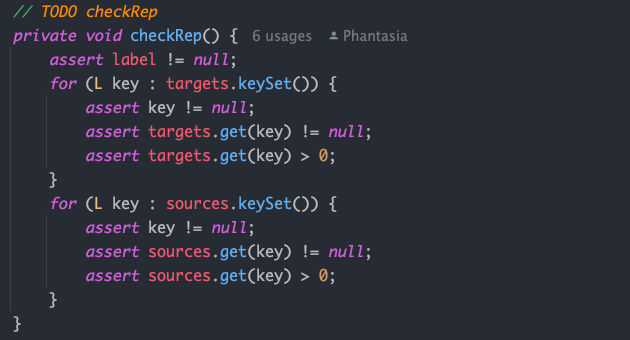
Vertex 类作为对顶点的描述，首先每个顶点应当有标志的 label，并且在有向图中有自己作为 source 指向的 targets，也有自己作为 targets被指向的 sources，并且这些有向的边应当有权重，于是可以使用 Map，以顶点的 label 作为 key，边的权重作为 value，设计两个 Map 分别是 sources 和 targets ，存储指向当前 vertex 和当前 vertex 指向的顶点集合。设计出 Rep 如下图所示：



根据上面的描述，AF 已经显而易见了，RI 的设计只需要 label 不为 null，并且 targets 和 sources 不含有 null 元素就可以，Safety from rep exposure 方面，需要在特定的 Accesser、observer方法和 Mutable 方法中访 问和修改内部的数据，所以变量都设置为 private 且 final。



根据 RI 的设计，我们都可以写出下面的 checkRep()

、

Methods 的设计

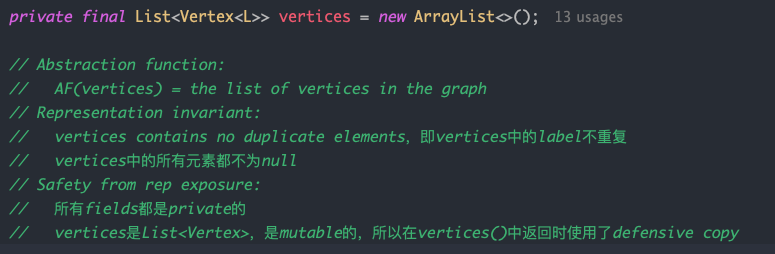
|  |  |
| --- | --- |
| getLabel() | 返回当前 vertex 的 label |
| getTargets() | 返回当前 vertex 指向的顶点的集合 |
| getSources() | 返回指向当前 vertex 的顶点的集合 |
| addTarget() | 为当前顶点添加一个指向的点 |
| removeTarget() | 删除当前顶点指向的点 |
| addSource() | 添加一个指向当前顶点的点 |
| removeSource() | 删除指向当前顶点的点 |
| getSourceWeight() | 根据 source 获得当前边的权值 |
| getTargetWeight() | 根据 target 获得当前边的权值 |
| toString() | 覆写 toString 方法 |

大多数方法的设计较为容易，getTargets 和 getSources 方法会返回进行的防御性拷贝的副本，其中需要特地注意 addSource 方法和 addTarget 方法，需要处理一些特殊的输入，以 addSource 方法为例，在处理 source 和 weight 时，需要针对 source 是否为 null，weight 是否为正数进行处理，如果 weight 为 0，则删除这条边，为负数则抛出 illegalArgumentException 的异常，剩下的情况才会正常工作，实现如下：



addTarget 方法实现类似，不多赘述。

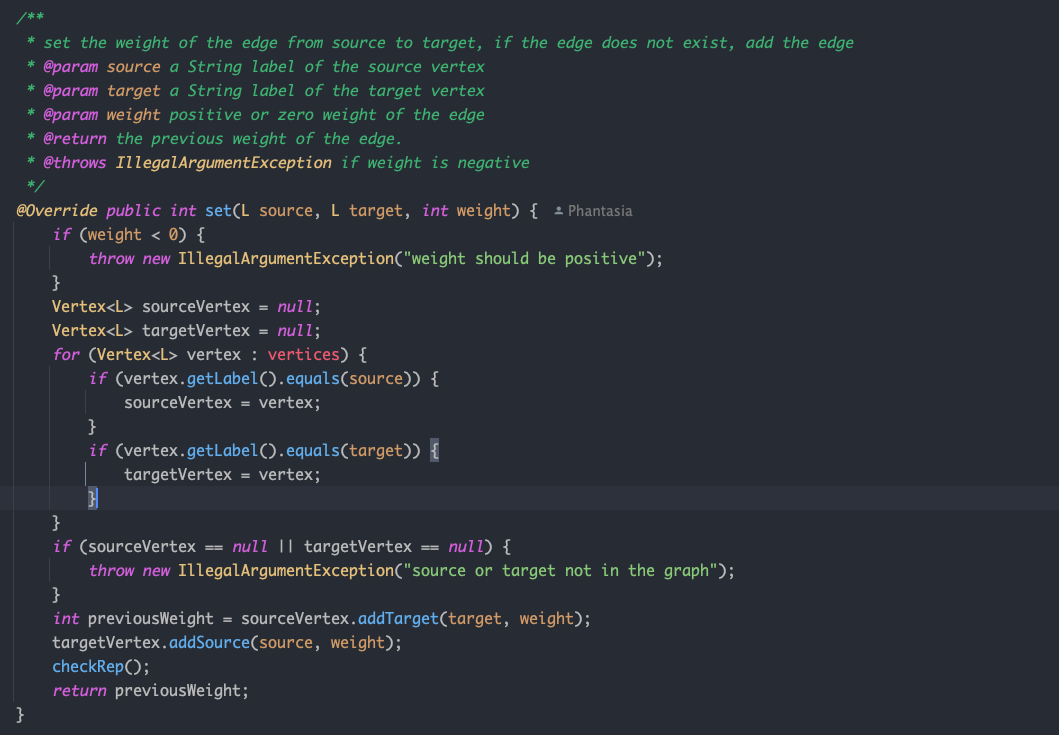
接下来就是 ConcreteVerticesGraph 的实现，由于代码已经设定好使用提供的 Rep，所以只需要给出对应的 AF、RI、和 Saferty from rep exposure，vertices 数组不应当含有重复的顶点，并且不能包含内容为 null 的顶点，而所有的 rep 应当是 private且 final 的，额外提供 mutator 方法以给供修改。实际设计如下：



根据 RI 的内容可以设计 checkRep 如下：



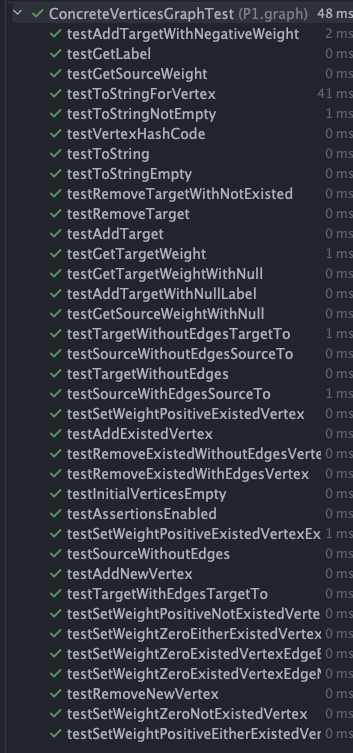
至于剩下的一些方法的设计，部分方法实现简单易懂，比如 add 方法，只需要做简单的类型检验和判断，并且 checkRep 即可，vertices 方法更是只需要返回 rep vertices 数组的 defensive copy 即可，set 方法、remove 方法和 ConcreteEdgesGraph 的设计如出一辙，只是由于使用 vertex 类实现，需要遍历每个 vertex 的数组来检查，并且删除的时候也需要 remove 与这个节点相关的所有节点，下面是 set 函数的代码示例：



targets 和 sources 方法，则需要遍历所有的以输入的参数的 vertex 姐弟那为 source 的节点，或者是 target 的节点，这需要遍历vertices 数组，下面是 sources 函数的代码实现：



综上，该类差不都就完成了。下面给出测试结果和代码覆盖度的测试:

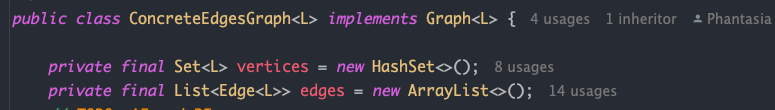


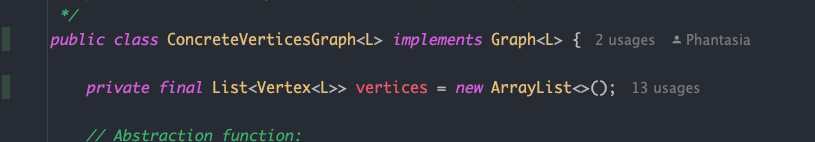


### Problem 3: Implement generic Graph<L>

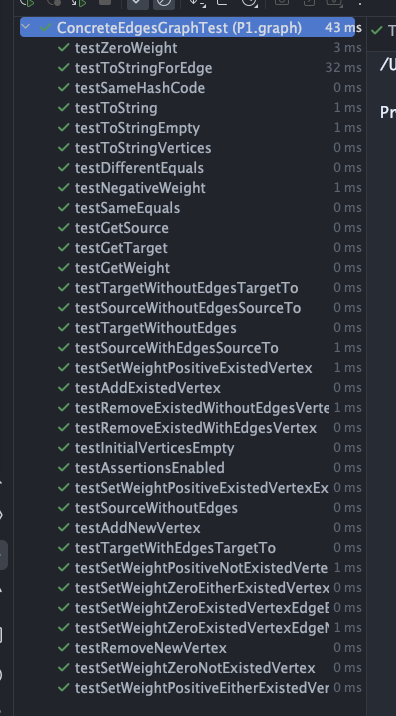
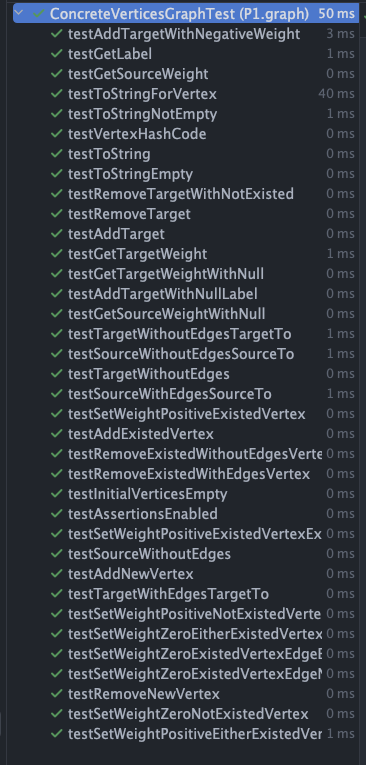
#### Make the implementations generic

关于泛型的具体实现，首先需要对原先指定 String 类型的变量将类型均改为泛型 L，并且使用迭代器循环的时候auto 形式的变量也应该指定类型，比如 Edge<L>等，并且修改类的声明：





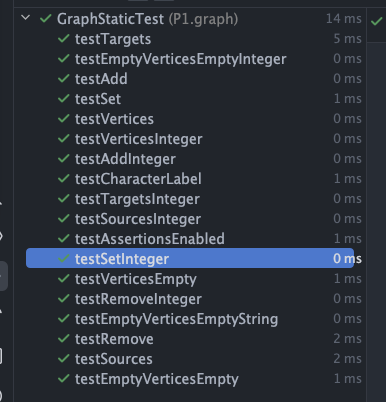
剩下的修改在代码内完成即可，较为简单。下面是修改完成后的测试结果：



#### Implement Graph.empty()

调用子类中的构造器即可：  
 

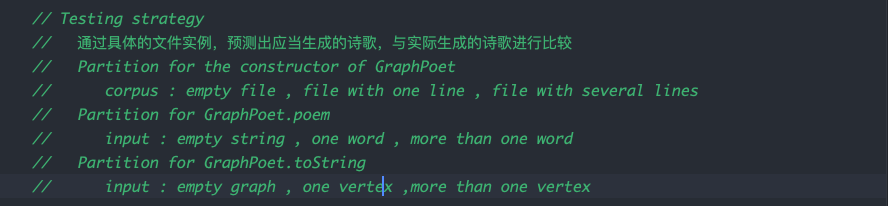
根据要求，添加了对其他 immutable 类型，添加了 Integer 类型和 char 类型的测试，测试均通过，结果如下：



### Problem 4: Poetic walks

#### Test GraphPoet

根据测试文件可能出现的情况，空文件、单行长文件、单个单词文件、正常文件，以及对 Poem 函数输入可能出现的情况：空串、单个单词、正常语句。包括对 toString 方法的测试，如空图、单个顶点、多个顶点、多个顶点不含边、正常图。测试策略如下：



并且搜索符合要求的预料库，准备作为测试样例。

#### Implement GraphPoet

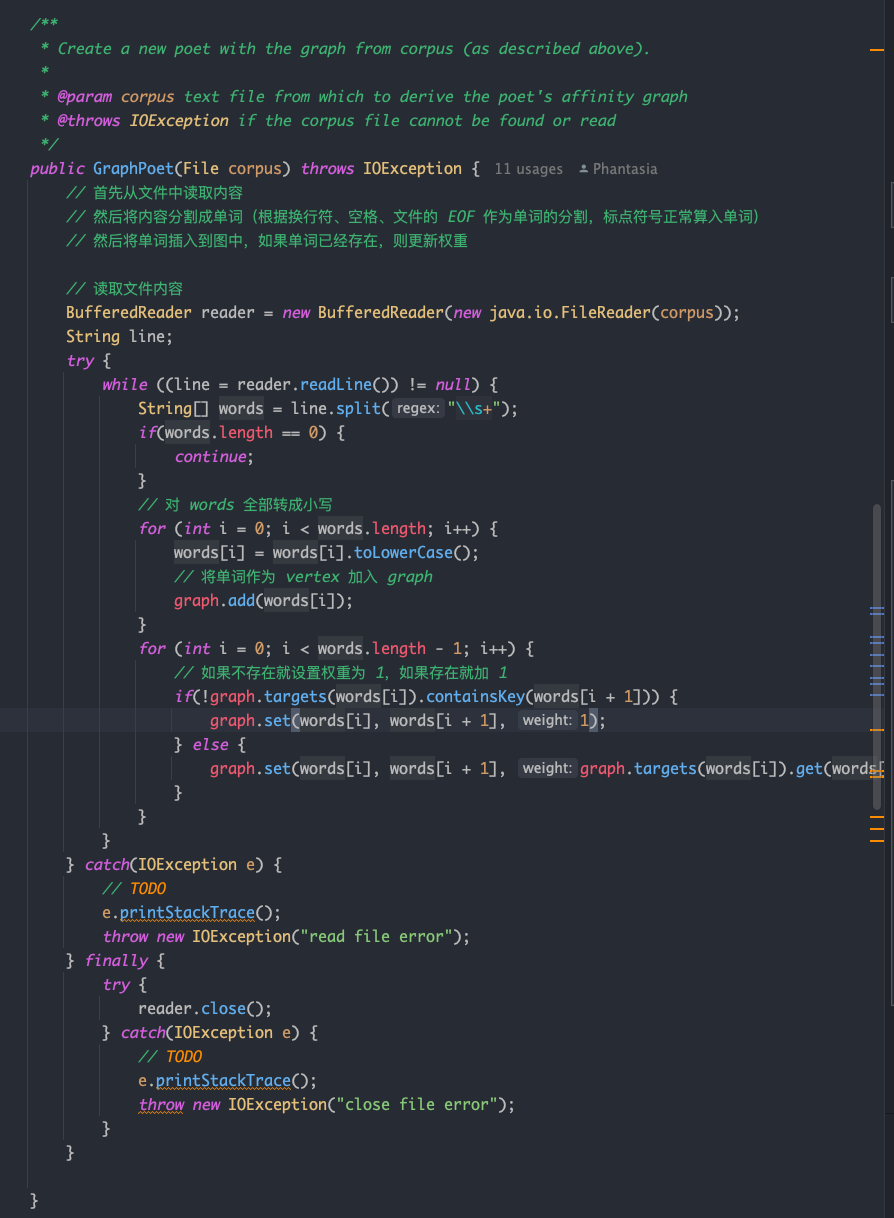
首先是 Rep 的设计，使用代码中规定好的empty 工厂方法获得一个 graph 实例，于是可以简单设计 AF、RI 和 Safety from rep exposure。



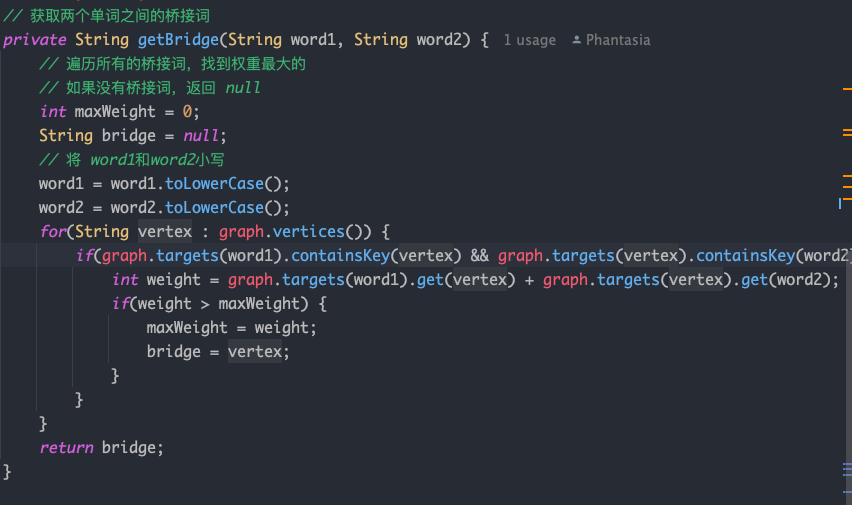
接下来是 checkRep 的内容，实际上 checkRep 的过程中和实现有关，因为检查不变量的过程中判断了顶点内容非空



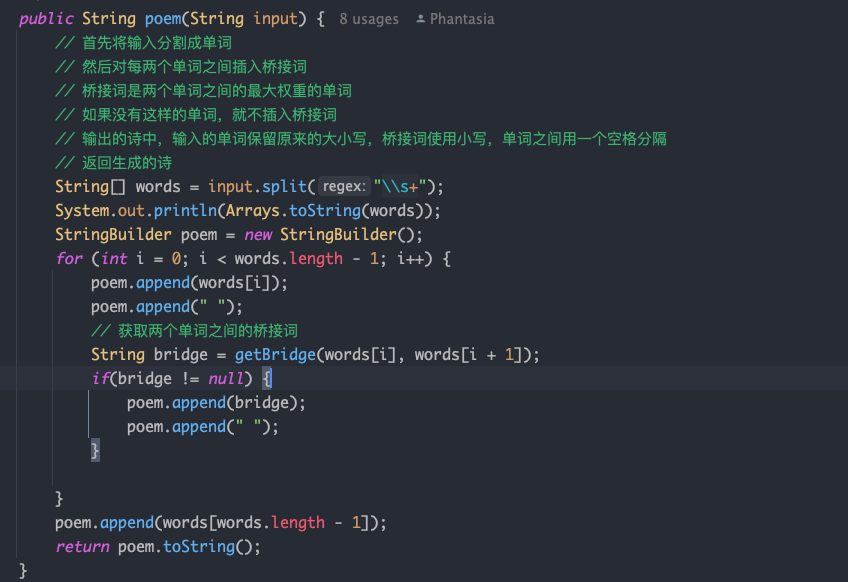
constructor需要生成一个完整的词汇亲和图，并且构造器的参数是一个文件类型，考虑使用 bufferReader 来读取文件的内容，可以使用 readLine，使用空白符，考虑使用正则表达式“\s+”来作为 split 的分割对象,由于 java 中反斜杠需要转义，于是变成了“\\s+”，并且注意词汇亲和图中的词汇都是小写的，所以需要对读入的单词进行 lowercase，当重复出现的时候进行权重的增加即可。实现如下图所示：



接下来是 poem 方法的实现，需要根据具体的词汇亲和图，将输入的句子进行改写，实际上是通过添加桥接词实现的，所以不妨先设计一个方法 getBridge，用来获得两个词汇之间的权重最大的亲和词，无非就是第一个词的 target 和第二个词的 source 一致，并且两条边的权重之和最大，所以实现想对容易，下面是具体的代码实现：

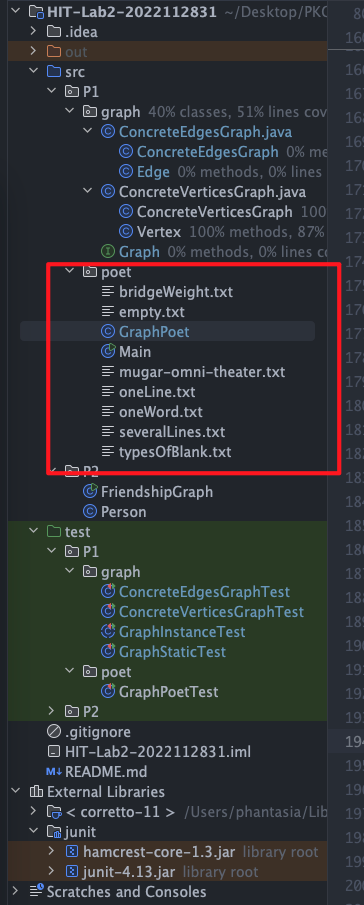


接下来是 peom 方法的具体实现，通过不断的拼接单词和桥接词就可以获得最终的语句。



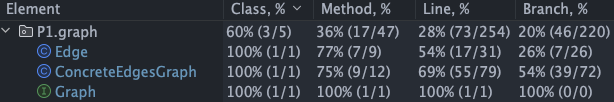
#### Graph poetry slam

从各种渠道包括网络或者小说等获得一定的预料库资源，存于对应的文本文件中。

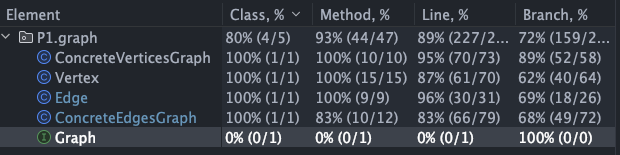


### 使用Eclemma/Code Coverage for Java检查测试的代码覆盖度

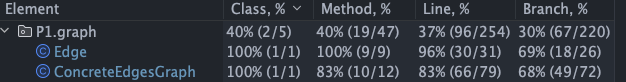
GraphStaticTest



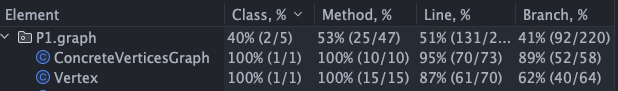
GraphInstanceTest



ConcreteEdgesGraphTest



ConcreteVerticesGraphTest

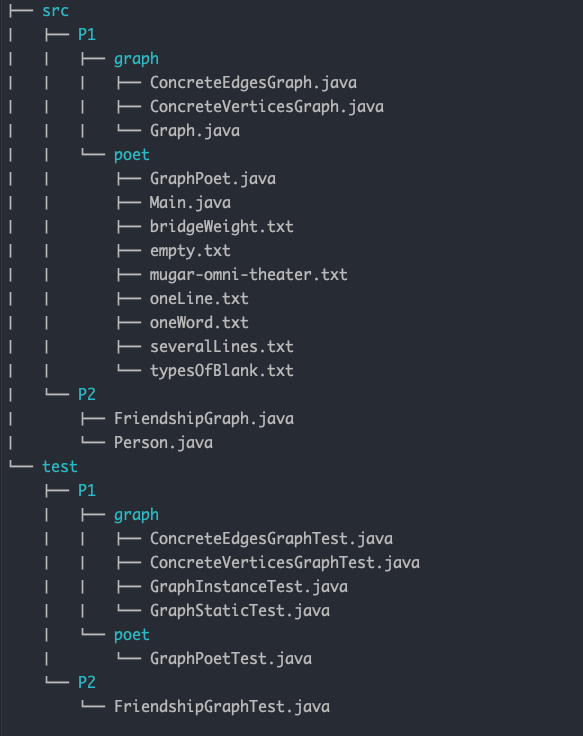


### Before you’re done

# 请按照[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，检查你的程序。

# 如何通过Git提交当前版本到GitHub上你的Lab2仓库。

# 在这里给出你的项目的目录结构树状示意图。



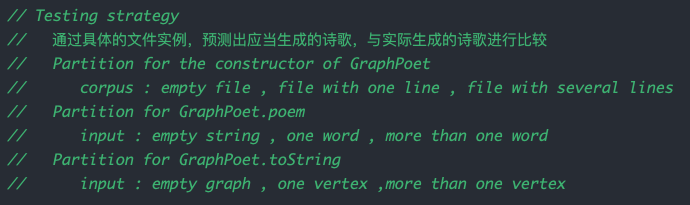
## Re-implement the Social Network in Lab1

根据第3.1节中“Poetic Walks”所定义的Graph<L>及其实现，重构实验1中第3.3节描述的FriendshipGraph类。确保新实现的FriendshipGraph类能与实验1中第3.3节的客户端代码（如main()函数中的代码）兼容，即不需要修改客户端代码也能正常运行。同时，重新运行实验1中编写的JUnit测试用例，以验证本实验中新实现的FriendshipGraph类的功能是否正常。

### FriendshipGraph类

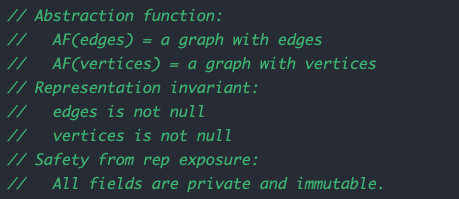
# 给出你的设计和实现思路/过程/结果。

首先完成测试的策略



根据测试的覆盖完成对应的测试代码即可，不过多赘述。

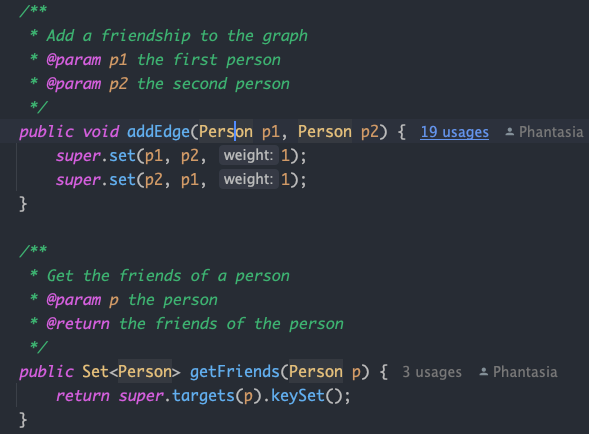
其次由于使用了一个工厂方法生成了一个 graph，内部调用了 concreteEdgesGraph 的构造器，实际上 AF、RI、Safety from rep exposure应当和ConcreteEdgesGraph 中实现比较类似，具体如下:



checkRep 如下：



其他方法的实现思路，其实大多数方法通过继承关系调用父类中的方法就可以完成，不如 addVertex，addEdge，getFriends 等，下面给出部分方法的实现：



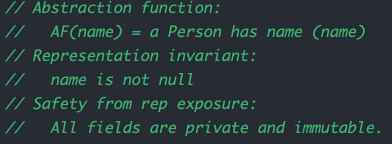
而 getDistance 方法的实现和 Lab1 中的实现思路也如出一辙，并没有不一致，只是图复用了 3.1 中实现的图，仅此而已，代码实现如下：



### Person类

# 给出你的设计和实现思路/过程/结果。

Person 类只需要记录当前 Person 的 name 即可，AF、RI 等如下图所示



检查不变量如下图所示：



实现了 accessor 方法 getName，覆写了 equals 方法和 hashcode 方法，不再赘述。

### 客户端main()

# 给出你的设计和实现思路/过程/结果。

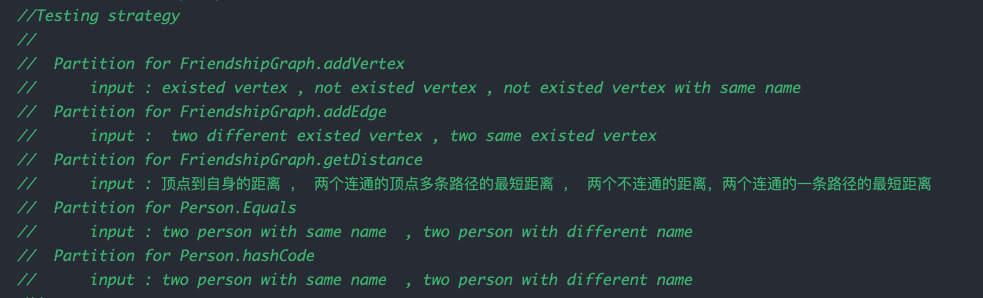
和 Lab1 中的代码实现一致，具体如下图所示



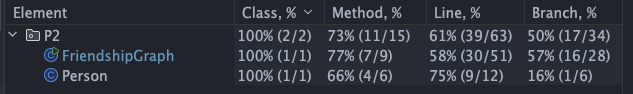
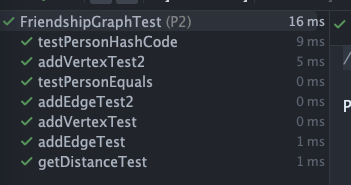
### 测试用例

# 给出你的设计和实现思路/过程/结果。

测试策略如下



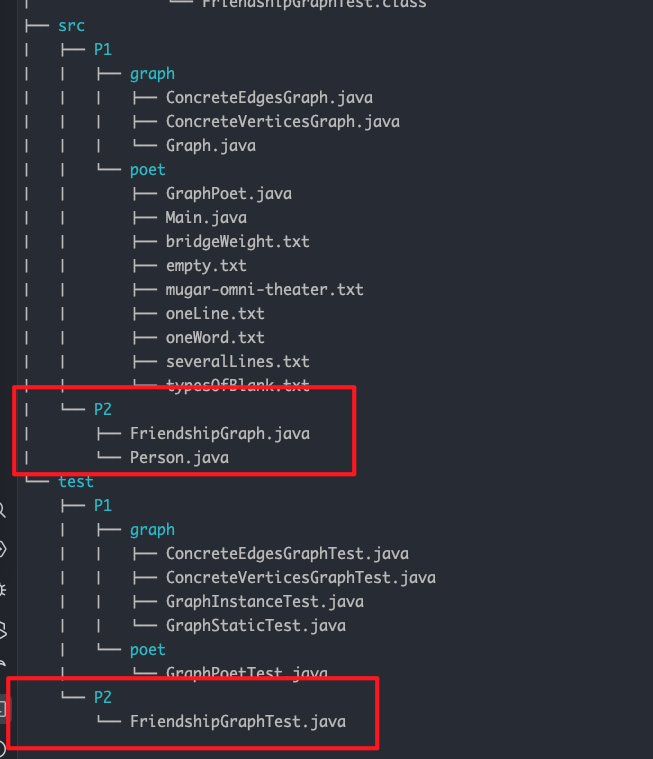
代码实现较为简单，和 Lab1 中的实现也十分类似，下面是测试结果和代码覆盖度：



### 提交至Git仓库

如何通过Git提交当前版本到GitHub上你的Lab3仓库。

在这里给出你的项目的目录结构树状示意图。



# 实验进度记录

请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。

每次结束编程时，请向该表格中增加一行。不要事后胡乱填写。

不要嫌烦，该表格可帮助你汇总你在每个任务上付出的时间和精力，发现自己不擅长的任务，后续有意识的弥补。

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 4.10 | 13:00-17:00 | finish graph instance and static test design | finished |
| 4.14 | 18:00-24:00 | finish graph edges and vertices graph implements and test design | finished |
| 4.15 | 15:00-17:00 | finish the poetic walk task | finished |
| 4.20 | 18:00-20:00 | finish the friendshipgraph task | finished |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| 理解整个实验二需要做的任务 | 通过翻译部分难以理解的语句，并且尝试进行代码的书写，在书写代码过程中逐渐掌握实验二的要求和主要任务 |
| 完成 graphInstanceTest 的设计 | 在设计这个测试类的时候意识到应当对测试类的覆盖进行划分，于是开始重新分析输入以及调用方法的 Rep 的状态，并且划分出对应的覆盖，并且过程中重写了 Static 测试 |

# 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）

做完这个实验后，我觉得自己学到了不少东西，但也确实碰到了不少问题。首先，实验让我更深刻地理解了抽象数据类型（ADT）和面向对象编程（OOP）的概念。一开始我对泛型和接口的使用还不是很熟悉，特别是在实现泛型图的时候，还得搞清楚怎么保证类型安全和灵活性，这部分花了我不少时间。

在编写代码的过程中，我发现写测试用例也很关键。一开始我就是随便写几个测试来看看代码能不能跑，后来发现这样做问题多多，改了前面的代码又得回头去修测试。后来我学会了先根据ADT的规格说明仔细设计测试用例，确保它们能全面覆盖各种情况，这样在实际编码时心里有数，出错的机会也少多了。

## 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的编程和直接面向应用场景编程在方法和思考过程上有很大的不同。从我的经验来看，面向ADT编程让我更多地关注于数据类型的设计和操作的抽象，而不是立刻跳到具体的应用实现上。这种方法有助于我先理清数据结构和功能方法的关系，使得编程更加模块化，也易于维护和扩展。

例如，在这个实验中，我们首先定义了图（Graph）的ADT，包括它应该有的方法和属性，然后再根据这个抽象定义去实现具体的功能。这样做的好处是，一旦ADT定义好了，实现细节即使变动，也不会影响到整个程序的其它部分，因为其他代码依赖的是ADT的接口，而不是具体实现。

而直接面向应用场景编程，通常是从一个具体的问题出发，直接着手解决问题。这种方式可能会更快地得到一个工作解决方案，但有时候会因为太注重立即的需求而忽视了代码的可维护性和扩展性。比如，如果一开始就钻研如何在社交网络中快速查找朋友关系，可能会写出很具体的代码，但这段代码在需求变化时可能就需要重写。

1. 使用泛型和不使用泛型的编程，对你来说有何差异？

使用泛型编程和不使用泛型编程在我看来差别挺大的，主要体现在代码的灵活性、安全性和可读性上。

使用泛型可以让代码更加灵活。在实验中，当我们实现图（Graph）这样的数据结构时，如果用泛型来定义，就可以让同一个图类用于不同类型的节点。比如说，我可以有一个存储字符串节点的图，也可以很方便地改为存储整数或者自定义对象的图。这种方式可以极大地复用代码，避免了为每种数据类型都编写一个新的类的重复。

然后，使用泛型还可以增强代码的类型安全性。通过泛型，编译器可以在编译时期就检查类型错误，减少运行时出错的可能性。比如，在一个泛型图类中，如果我尝试插入一个错误类型的节点，编译器会直接报错，而不是等到运行时才发现问题。

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？

在给出ADT（抽象数据类型）的规约后就开始编写测试用例，这种做法实际上是一种“测试先行”或者说“测试驱动开发”（TDD）的策略。这种方式带来的优势挺多的，对我来说，最大的几点好处是：确保覆盖所有功能、改善设计质量、促进模块化、及时发现错误，至于我是否能够适应这种测试方式，开始的时候确实有点不习惯，因为这种方式需要在还没有开始写实际的功能代码之前，就去思考如何验证这个功能。但是随着实践的增多，我发现这样做其实对整个开发过程有很大的帮助，特别是能够减少后期的debug时间，让开发过程更加迅速。

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？

显而易见的是降低了开发的成本，能够减少很多重复开发的时间，其次也提升了代码的稳定性和质量，只需要对 P1 中的 ADT 进行维护，并且泛型编程能够在编译时通过静态类型检查就能发现一些错误，能够减少代码在运行中出错的可能性。

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？

这些工作提升了代码的可读性和可维护性，让其他人能够方便的理解代码的实现逻辑，代码的健壮性也提升了，明确的 RI AF 等的定义，能够让方法和 Rep 在实现和定义的时候符合规范，增强代码的健壮性，同时由于表示独立性，能够避免代码的内部实现暴露。

1. 关于本实验的工作量、难度、deadline。

P1 的工作量相当巨大，P2 极少，难度适中，deadline 合适

1. 《软件构造》课程进展到目前，你对该课程有何收获和建议？

到目前为止，这门课程真的让我学到了很多关于如何更专业地编写软件的知识和过程。讲解了很多关于软件设计原则、测试驱动开发、重构技术等现代软件工程的实践方法，这些对我的编程思维有比较大的提升。

如果要给出建议，我希望课程能增加一些关于现代软件工具和环境的教学，比如更多关于Git、Docker、k8s 等或者持续集成工具的实际操作。这样不仅能帮助我们掌握编程技能，还能让我们熟悉和适应当前行业中常用的工具，为将来走向职场做好更充分的准备。