

**2021年春季学期**

**计算学部《软件构造》课程**

**Lab 2实验报告**

|  |  |
| --- | --- |
| 姓名 | 李昆泽 |
| 学号 | 1190201018 |
| 班号 | 1936603 |
| 电子邮件 | 3343033352@qq.com |
| 手机号码 | 15720828552 |

**目录**

2 实验环境配置

3 实验过程

3.1 Poetic Walks

3.1.1 Get the code and prepare Git repository

3.1.2 Problem 1: Test Graph <String>

3.1.3 Problem 2: Implement Graph <String>

3.1.3.1 Implement ConcreteEdgesGraph

3.1.3.2 Implement ConcreteVerticesGraph

3.1.4 Problem 3: Implement generic Graph<L>

3.1.4.1 Make the implementations generic

3.1.4.2 Implement Graph.empty()

3.1.5 Problem 4: Poetic walks

3.1.5.1 Test GraphPoet

3.1.5.2 Implement GraphPoet

3.1.5.3 Graph poetry slam

3.1.6 使用Eclemma检查测试的代码覆盖度

3.1.7 Before you’re done

3.2 Re-implement the Social Network in Lab1

3.2.1 FriendshipGraph类

3.2.2 Person类

3.2.3 客户端main()

3.2.4 测试用例

3.2.5 提交至Git仓库

4 实验进度记录

5 实验过程中遇到的困难与解决途径

6 实验过程中收获的经验、教训、感想

6.1 实验过程中收获的经验和教训

6.2 针对以下方面的感受

# **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**

实验环境基本上与Lab1类似，这里还需要在 Eclipse IDE 中安装配置 EclEmma（一个用于统计 JUnit 测试用例的代码覆盖度的 plugin），直接从Eclipse Market下载安装即可。

在这里给出你的GitHub Lab2仓库的URL地址（Lab2-学号）。

https://github.com/ComputerScienceHIT/HIT-Lab2-1190201018

# **实验过程**

## **Poetic Walks**

该任务是基于Graph<L>这个接口的，有了这个接口，这道题的大部分工作就基本上完成了，只要再加入一些简单的算法即可。

而关于Graph<L>这个接口，我们首先把它简化为Graph<String>，然后分为两个具体的类（分别是ConcreteEdgesGraph和ConcreteVerticesGraph）进行实现。在接口Graph<L>，实现定义了一些方法，而在具体的实现类里需要我们一一实现这些方法，并且保证不会有表示泄露的问题。

### **Get the code and prepare Git repository**

使用git命令：

git clone [git@github.com:rainywang/Spring2021\_HITCS\_SC\_Lab2.git](mailto:git@github.com:rainywang/Spring2021_HITCS_SC_Lab2.git)

即可获取到该任务的代码。

选择一个文件夹使用git init命令即可创建git仓库；

每完成一部分任务后都可以使用下列命令将仓库推送到GitHub上。

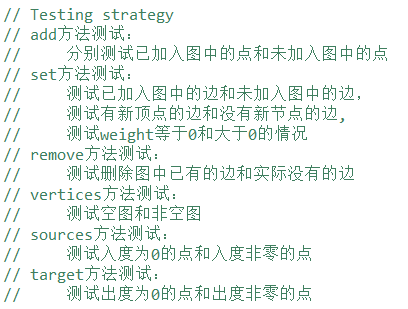
git add .

git commit -m “说明”

git push origin master

### **Problem 1: Test Graph <String>**

这部分主要是针对Graph设计测试策略，编写测试用例主要利用等价类划分的思想进行测试，测试策略如下：

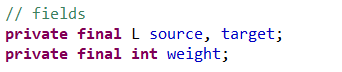


编写覆盖以上条件的测试用例即可。

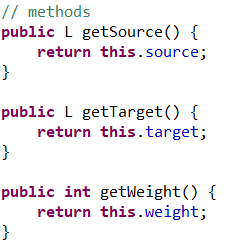
### **Problem 2: Implement Graph <String>**

#### **Implement ConcreteEdgesGraph**

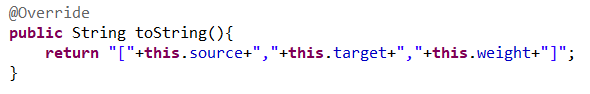
首先实现Edge类。Edge类的设计思路比较简单。Fields主要有三个rep，分别表示起始点、终止点和边的权重，如下图所示。



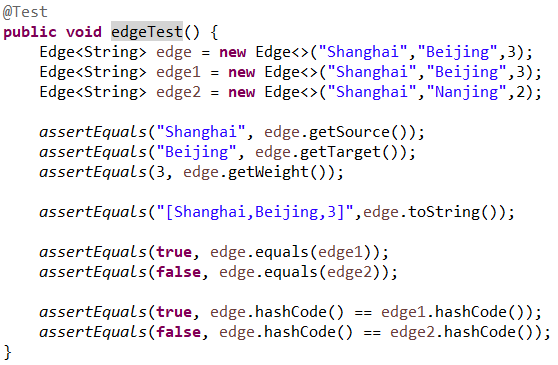
由于Edge类是immutable的，所以在Edge类中除了构造方法以外其他方法都不能修改rep的值，这也就导致Edge类中只有一些Observer的方法，如下图所示。



除此之外还有对toString方法的重写，如下图所示。

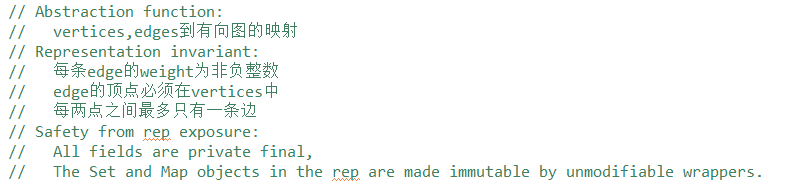


然后是针对Edge这个类的测试，截图如下。

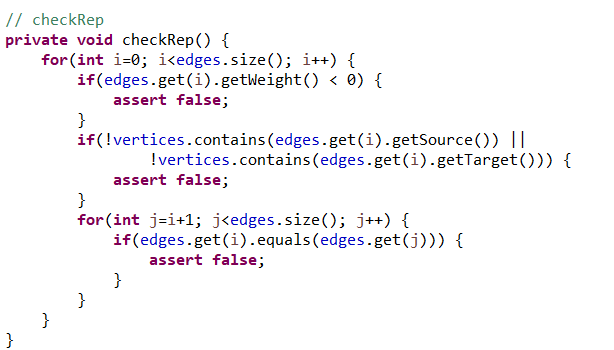


在这个测试样例里，对Edge里出现的各种方法均进行了测试，最后也通过了测试，总的测试通过截图会在之后一起给出。

下面是对ConcreteEdgesGraph类的实现。首先明确AF，RI和关于rep exposure的声明。



根据上述声明可以实现checkRep。



下面是一些实例方法的实现。

1. add方法

首先用Set的contains方法判断当前点是否在有向图中，如果在，则不进行操作，并返回false；如果不在，则调用集合的add方法加入当前点。

1. set方法

这里要考虑多种情况：

如果weight<0（非法），则输出提示信息，不对图做任何操作；

如果这条边存在，并且weight>0，则更新weight；

如果这条边存在，并且weight=0，删除之前的边；

如果这条边不存在，并且weight>0，则往图中加入这条边；

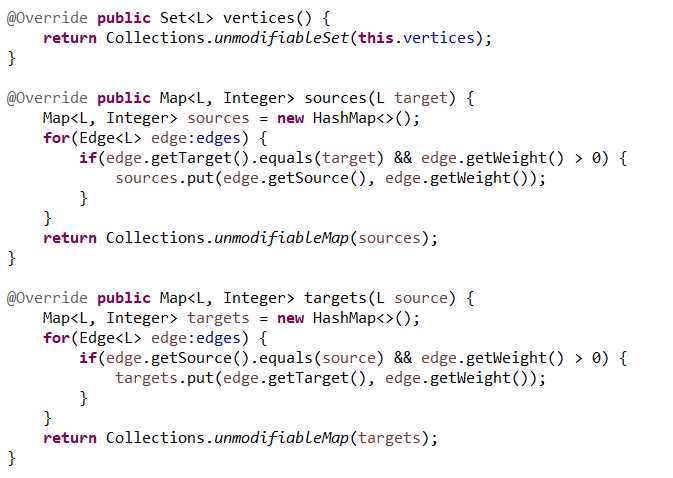
如果这条边不存在，并且weight=0，则不进行任何操作。

1. remove方法

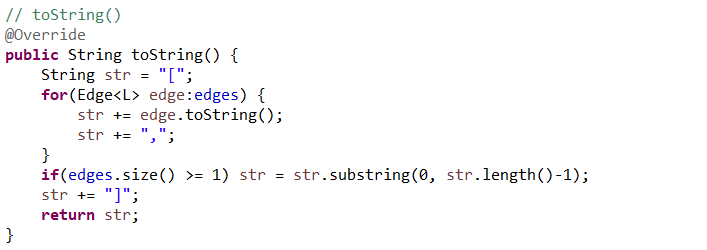
首先用Set的contains方法判断当前点是否在有向图中，如果不在，则不进行操作，并返回false；如果在，则调用边集edges的迭代器，删除所有与当前点相关联的边。

1. vertices、sources和targets方法

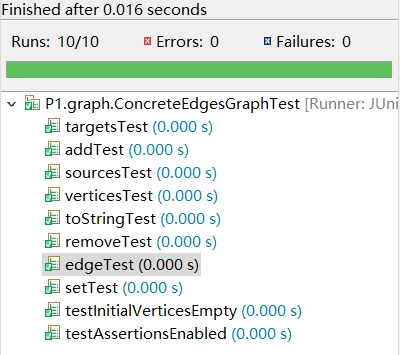
这三种方法比较好实现，这里就不再赘述具体的实现方法了，但是要注意它们返回值的类型是mutable的，可以使用unmodifiable wrappers将其变为immutable的。这三个方法的具体实现方法如下图所示。



最后是toString方法的重写，这个可以在Edge类的toString方法的基础上进行。



在测试部分，首先对toString这个Observer方法进行测试，然后在其他的instance方法中，则反复利用toString这个Observer方法进行测试，这样能充分测试整个对象的变化，而不是仅仅利用函数返回值进行测试。最后是所有测试均通过的截图。



#### **Implement ConcreteVerticesGraph**

这个类对有向图的实现类似于邻接表的处理。

List<Vertex>存储的是一些节点的信息，而Vertex里存储的是以当前节点为起始节点的所有点的信息和相应边的权重。

首先看Vertex类的实现。

（1）fields

可以设置一个name属性，记录当前节点的label。

然后设置一个Map<L, Integer>用来记录以当前节点为起始节点的所有点的信息和相应边的权重。

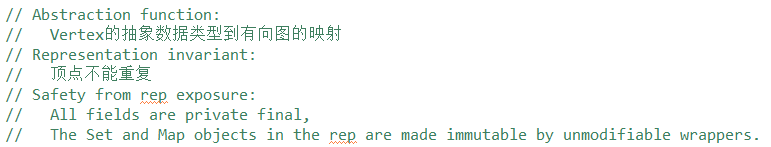
1. Methods

|  |  |
| --- | --- |
| 方法名 | 作用 |
| getName() | 返回当前节点的名称 |
| add(L vertex, int weight) | 添加vertex节点和对应权重到Map |
| remove(L target) | 从Map中移除名称为target的顶点（如果存在） |
| set(L target, int weight) | 设置Map中名称为target的节点（如果存在）对应边的权重。如果weight为负，则不进行任何操作；如果weight为0，则删除原来有的边；如果weight大于0，则更新边的权重。 |
| Vertices() | 返回Map |
| getWeight(L target) | 返回名称为target节点对应边的权重 |
| toString() | 以边的形式将Vertex转化成字符串 |

关于Vertex类的测试就要比Edge类复杂不少了。不过总的测试思想没有变，还是在测试instance方法时要尽量利用observer方法来检验，而在测试observer方法时却要用instance方法来对Vertex对象进行适当的改变。

下面是ConcreteVerticesGraph类的实现。

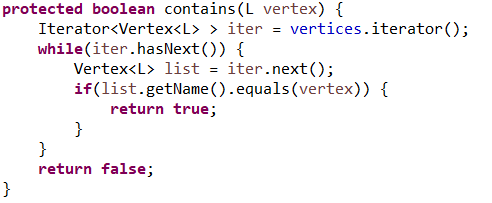
关于AF、RI和safety from rep exposure的声明如下。



下面是一些实例方法的实现。

（1）add方法

首先利用辅助的contains方法判断当前点是否在有向图中，如果在，则不进行操作，并返回false；如果不在，则调用List的add方法加入当前点对应的Vertex<L>。辅助方法的实现如下图所示。



（2）set方法

这里要考虑多种情况：

如果weight<0（非法），则输出提示信息，不对图做任何操作；

如果这条边存在，并且weight>0，则更新weight；

如果这条边存在，并且weight=0，删除之前的边；

如果这条边不存在，并且weight>0，则往图中加入这条边；

如果这条边不存在，并且weight=0，则不进行任何操作。

（3）remove方法

首先依然用之前提到的contains方法判断当前点是否在有向图中，如果不在，则不进行操作，并返回false；如果在，则调用List的迭代器，删除所有与当前点相关联的边。

（4）vertices、sources和targets方法

这三种方法比较好实现，这里就不再赘述具体的实现方法了，但是要注意它们返回值的类型是mutable的，可以使用unmodifiable wrappers将其变为immutable的。这三个方法的具体实现方法如下图所示。



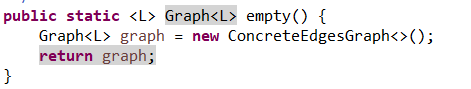
最后是toString方法的重写，这个可以在Vertex类的toString方法的基础上进行。

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

#### **Make the implementations generic**

这里一个比较容易的想法就是把实现过程中出现的String全部替换成L，但是会发现这样做可能会让程序报错，这是因为代码里可能用到了一些String特有的方法。我们将这些方法一一删去并使用一些通用的方法替代即可。

#### **Implement Graph.empty()**



Graph<L>是一个接口，需要用具体的类去实现。在new一个新的Graph<L>对象的时候也是如此，我们需要指定通过哪种具体的类进行实现，比如上图使用的是ConcreteEdgesGraph类。

### **Problem 4: Poetic walks**

#### **Test GraphPoet**

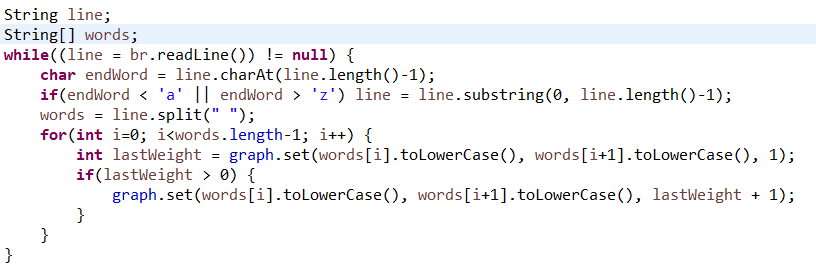
这里我们的测试策略是测试权重均为1的有向图和权重不都为1的有向图，如果权重均为1，那么对于可能新增单词的一个单词对，可能有多个单词可供选择，并且它们之间是没有优先级之分的；而如果权重不都为1，那么当选择新增的单词时，就可能会有有限级之分。另外，针对有向图为空或者input为空的情况，我也编写了相应特殊的测试样例进行测试。

据此测试策略，我编写了两个测试样例。

测试1根据“a b c d a.”构建有向图，很明显，图中所有边的权重都是1；测试2根据“a b c a d c d c.”构建有向图，图中“dc”这条边的权重是2。测试3的input为空，最后的结果也应该为空；测试4的有向图为空，最后的结果应该与input相同。

#### **Implement GraphPoet**

首先是根据文件中给出的字符串构建加权有向图。



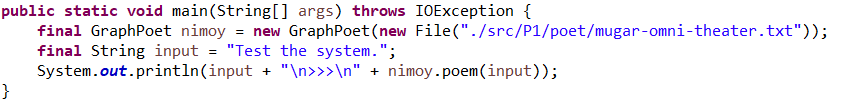
一开始，先过滤掉每行末尾可能的标点符号，然后调用split方法将它分解为一个字符串的数组。在往图里加入边时，需要调用Graph<L>自带的set方法，这个方法返回的是加入边之前这两个顶点间边的权重（如果边不存在则权重为0），我们利用这个特性构建GraphPeot中的有向图。我们往图中添加边时，默认权重是1,。用lastWeight这个变量保存之前边的权重，如果权重是0，则表示这两个点之前没有边相连，没有后续操作；如果返回的权重不是1，则说明这两个点之间本来是有边相连的，需要增加边的权重。

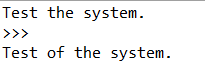
对于poem方法，对于每两个相邻的单词，我们只需要看前一个单词的targets和后一个单词的sources有没有交集即可，并且在这个交集中选取权重最大的那个（如果有多个则任选一个）作为新增的单词。这里需要注意无论是targets还是sources都是immutable的（使用了unmodifiable wrappers），所以我们需要新创建一个集合，先拷贝sources集合中的元素，再与targets求交集。如下图所示。



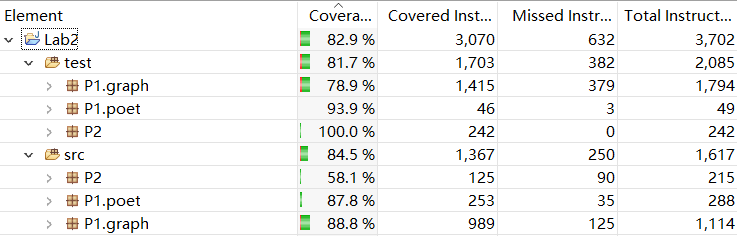
#### **Graph poetry slam**

原始数据是泰戈尔的生如夏花，输入输出如下：





### **使用Eclemma检查测试的代码覆盖度**



### **Before you’re done**

在本地git仓库中输入命令：

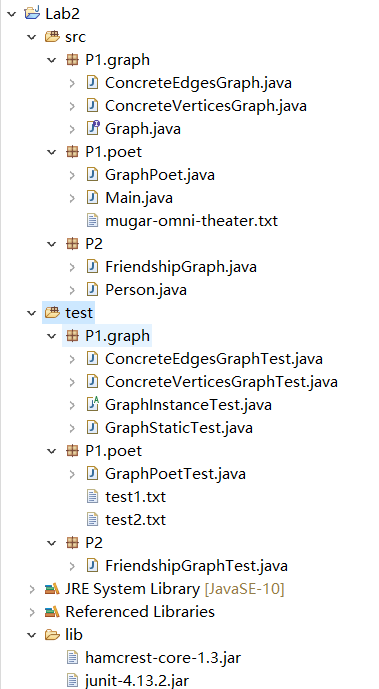
git add .

git commit -m “说明”

git push origin master

即可提交当前版本到GitHub上我的Lab2仓库。

项目的目录结构树状示意图如下。



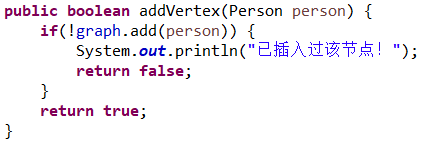
## **Re-implement the Social Network in Lab1**

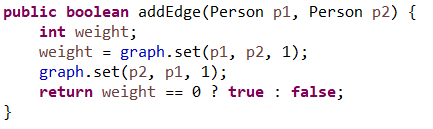
该任务主要是利用3.1节中实现的Graph<L>接口，重新实现 Lab1 中 3.3 节的 FriendshipGraph 类。

### **FriendshipGraph类**

1. addVertex()、 addEdge()

有了Graph<L>这个接口，这两个方法的实现就异常简单了。具体代码如下图所示。





这里需要注意的是，在这个类中，我们要构建的实际上是一个无向图，所以在加入边的时候每次要加入一对有向边。其实我们对这条边上的权重并不在意，只要原来没有这条边我们就加入一条新边，否则就不执行操作。

1. getDistance(Person p1, Person p2)

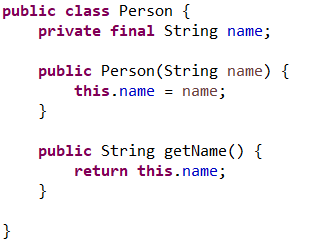
这里的实现思想实际上和Lab1里该方法的实现思路差不多，都是利用BFS计算两个节点的距离。具体代码如下图所示。



总的来看，这里的代码比Lab1中的实现代码要简单一些，有了Graph<L>这个接口，获取某个节点在图中的一些信息就会变得方便许多。例如BFS中想要获取与一个节点相连的所有节点，只需要调用图中的targets方法即可。

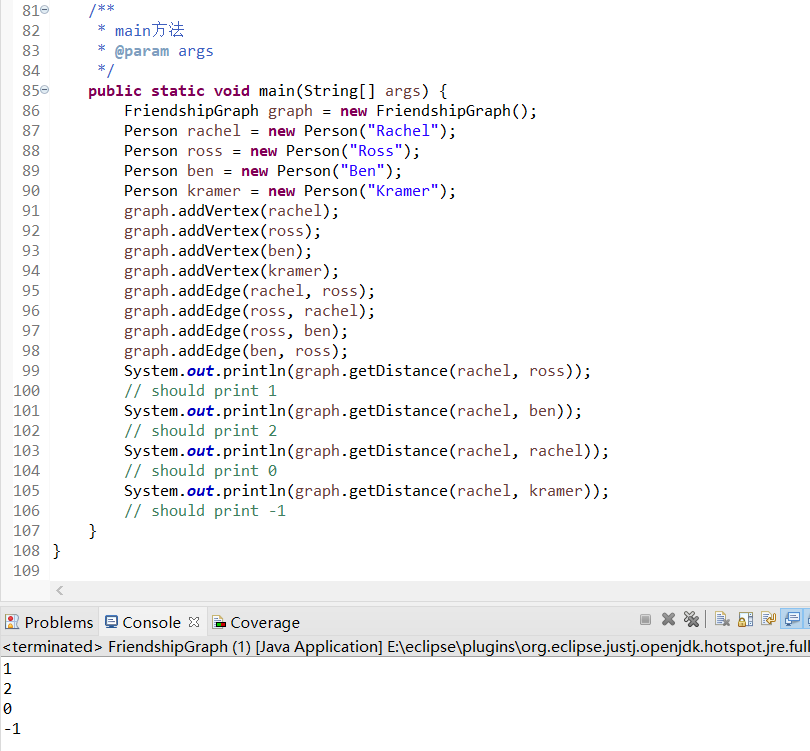
### **Person类**

Person类更加简单，因为有关图的操作全都在FriendshipGraph类中进行了实现，所以Person类中只需保存节点信息即可，具体实现如下图所示。



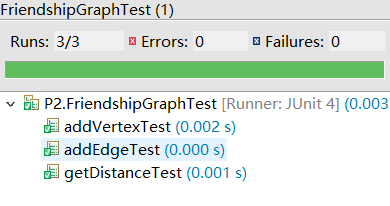
### **客户端main()**

Lab1中的客户端代码仍能执行出预期结果。



### **测试用例**

Lab1里所写的 JUnit 测试用例在现在的FriendshipGraph类上仍表现正常。



### **提交至Git仓库**

在本地git仓库中输入命令：

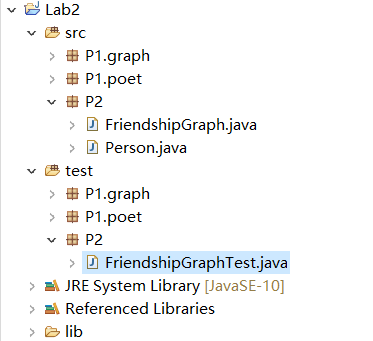
git add .

git commit -m “说明”

git push origin master

即可提交当前版本到GitHub上我的Lab2仓库。

项目的目录结构树状示意图如下。



# **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 |
| 2021-05-25 | 13:45-18:30 | 完成ConcreteEdgesGraph.java | ConcreteEdgesGraph.java基本完成，相关的JUnit未完成 |
| 2021-05-26 | 16:30-18:30 | 完成ConcreteEdgesGraph.java和测试样例的编写 | 完成计划任务，并且开始编写ConcreteVerticesGraph.java |
| 2021-05-28 | 9:00-11:20 | 完成ConcreteVerticesGraph.java和测试样例的编写 | 基本完成有向图的两种版本，并通过所有测试样例 |
| 2021-05-28 | 15:30-16:30 | 把String拓展到L | 按计划完成 |
| 2021-05-28 | 17:50-21:00 | 完成Peotic Walks | 解决了GraphStaticTest的报错问题，并且完成了Peotic Walks |
| 2021-05-29 | 18:30-20:00 | 完成所有功能的实现 | 实现了所有功能，完善了部分注释 |
| 2021-05-31 | 19:00-22:00 | 完成所有注释，完善代码、测试 | 按计划完成任务 |
| 2021-06-01 | 13:45-17:50 | 根据要求修改代码，写报告 | 按计划完成任务 |

# **实验过程中遇到的困难与解决途径**

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| 有时候对测试样例的编写有点没有头绪。 | 首先将测试样例划分成多个等价类，针对每个等价类寻找测试样例；另外是要充分利用Observer方法进行测试，而不能仅仅利用函数的返回值。 |
| 一开始不太清楚如何避免表示泄露。 | 表示泄露可以从几个方面避免：（1）fields中rep的类型设置成private final；（2）防御型拷贝；（3）如果返回值是可变类型的，可以使用unmodifiable wrappers将其转化为不可变的。 |
| 不知道checkRep应该如何去编写。 | 首先明确RI，想明白在整个类的各类方法的执行过程中，有哪些条件是必须满足的，然后在checkRep中主要使用assert语句对这些条件进行测试。 |
| 使用迭代器对一些可迭代的类型进行遍历时，在进行删除、修改其中的元素时，常常得不到想要的效果。 | 最好使用Iterator iter = type.iterator()的迭代方法，如果要删除该元素，最好使用iter.remove()方法，可直接删除该元素；而如果想修改某个元素，直接对这个可迭代的类型进行操作会比较好。 |

# **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**

这次实验最大的收获就是进一步改变了原来的编程习惯。

例如，在确定ADT的spec之后要先编写测试用例，保证自己设计的ADT是符合设计规约的，而不是一上来就编写代码。而在编写代码时，要时刻保证不会出现安全性（例如表示泄露）的问题，尽管这可能会带来代码执行效率的下降，但这却是作为程序员所必须要做的工作。

## **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？

面向ADT的编程一切只需要符合ADT的spec即可，具体有何应用场景，程序员无需知晓；而直接面向应用场景编程，程序员需要首先考虑如何设计ADT，这是面向ADT的编程所不需要的。

1. 使用泛型和不使用泛型的编程，对你来说有何差异？

不使用泛型的编程还是相对容易一些，不过在不使用泛型的基础上将其修改为使用泛型的程序也不太复杂。

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？

优势是确保自己设计的ADT能够符合规约。我在努力适应这种测试方式。

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？

使得代码更简洁，后续如果需要修改代码也更加方便。

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？

这些工作的意义是确保ADT的安全性、健壮性等等，以及是否符合spec。我愿意在以后编程中坚持这么做。

1. 关于本实验的工作量、难度、deadline。

本次实验的工作量主要集中在P1，总的难度其实还好，主要是适应这种新的编程模式还需要一定时间。

1. 《软件构造》课程进展到目前，你对该课程有何体会和建议？

这确实是一门需要编写大量代码的课程，不过如果想熟练掌握编程中的各种技巧以及新的编程风格，的确需要大量的代码训练。