哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2021 年春季学期

计算学部《软件构造》课程

Lab 2 实验报告

姓名	傅浩东
学号	1190202105
班号	1903002
电子邮件	1091288450@qq.com
手机号码	13881165621

目录

1 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。

2 实验环境配置

Java 环境实验一以及配置好，各种工具如 Git 也在实验一安装好。本实验要求安装配置的 EclEmma（用于统计 Junit 测试代码覆盖度的 plugin）在最新版的 Eclipse 中自带。

在这里给出你的 GitHub Lab2 仓库的 URL 地址（Lab2-学号）。
https://github.com/ComputerScienceHIT/HIT-Lab2-1190202105

3 实验过程

3.1 Poetic Walks

该任务主要是训练抽象数据类型（ADT）的设计、规约、测试，并使用面向对象编程（OOP）技术实现 ADT，最后 ADT 泛型化。

- 撰写测试用例：设计 Graph（黑箱检测）、编写异常操作。
- 两个类 ConcreteEdgesGraph，ConcreteVerticesGraph 实现 Graph 接口：add、set、remove、vertices、sources、targets，两个类中都有各自实现的类 Edge 和 Vertex，这些需要自己设计。
- 运用泛型的思想，将 String 拓展为泛型 L 类。
- 利用实现的 Graph，实现 GraphPoet 类，根据输入产生新的诗歌。

3.1.1 Get the code and prepare Git repository

在 Get the code 步骤中，从以下地址获取初始代码：
Git clone http://github.com/rainywang/Spring2021_HITCS_SC_Lab2.git
由于代码文件夹存在混乱，将 P1 置换到 src 和 test 目录下。
Prepare Git Repository 步骤中，git init。

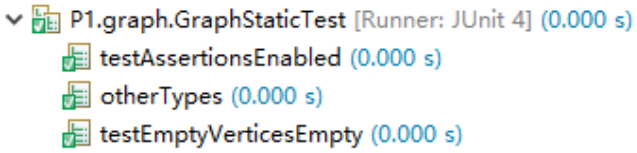
3.1.2 Problem 1: Test `Graph <String>`

目前只测试顶点标签为 String 的图形。之后将扩展到其他类型的标签。

1. 使用静态方法的测试策略在文件 GrapStaticTest.java 中进行测试，主要是测试函数 Grap.empty()。由于该方法是静态的，并且此时 L 类型只是 String，因此将只有一个实现，并且只需要运行这些测试一次。根据提供的这些测试，暂且不进行更改，后续过程将添加进 L 为不同类型的测试。
2. 在测试文件 GraphInstanceTest.java 中，根据编写的 testing strategy 来实现测试方法，测试包括：add、set、remove、vertices、sources、targets。对于每一个测试函数需要获得全新的空图，测试设计、实现思路和过程如下。

测试函数	测试设计、实现思路和过程
testAdd( )	往空图中添加新的点、已经存在的点，查看是否 add 成功。若是 add 重复的点，返回 false。
testSet( )	在图中添加进一些点，然后设置有向边，除了不同点之间的正权重边，考虑其他特殊情况： 边的 source 或 target 不存在，则 add 顶点； 若边权重为 0，删除该边； 边可以指向自己。
testRemove( )	删除一个顶点，同时要删除这个顶点作为 source 或 target 的那些边。因此除了检查该顶点是否还存在，还要检测相关边是否存在。
testVertices( )	略
testSourcesAndTargets( )	若有 A 作为 source 指向 B 的边，那么就有 B 作为 target 由 A 来的等权重的边；添加边后的某顶点的返回值与预期比较。

测试结果：

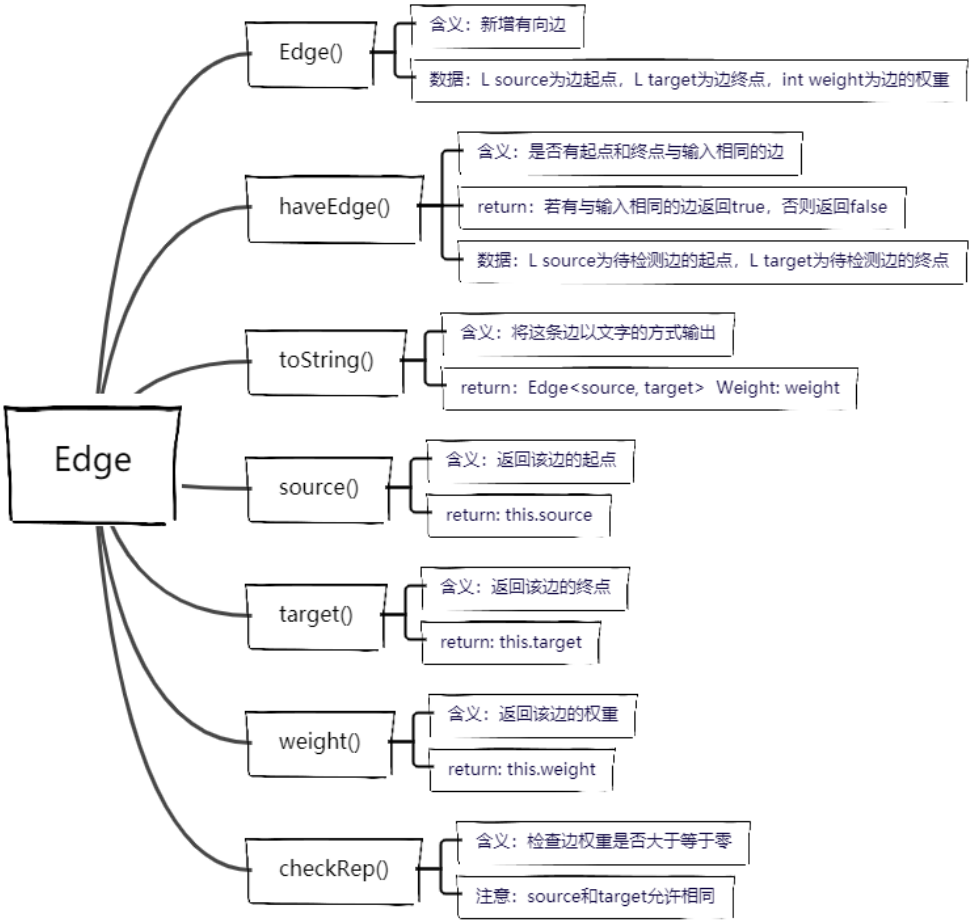


3.1.3 Problem 2: Implement `Graph <String>`

对于 Graph <String>的实现，主要是基于边和点两种方法来储存实现的。由此，在实现 ConcreteEdgesGraph 和 ConcreteVerticesGraph 之前还需要实现 Edge 和 Vertex 两种自定义类。除了实现上述问题中接口 Graph 的方法，每个方法都需要有 documented specification，还需要实现 AF(Abstraction Function)、RI(Representation Invariant)和防止表示暴露，书写 checkRep 方法以及重写 toString。

3.1.3.1 Implement `ConcreteEdgesGraph`

1. 首先设计 Edge 类：从 source 到 target 权重为 weight 的有向边。
   主要定义如下 4 中方法：haveEdge、source、target、weight，当然还重写了 toString，写了 private 方法 checkRep，它们的含义、参数、返回值以及注意事项等如下图所示：



首先对于 Edge 类里面的 fields

```
private final L source;
private final L target;
private final int weight;
```

对于Abstraction function，Edge类中的source表示有向边起点，target为该边的终点，weight为有向边的权重。

```
// Abstraction function:
// AF(source) = the source vertex of the edge
// AF(target) = the target vertex of the edge
// AF(weight) = the weight of the edge
```

对于Representation invariant，保证每一个Edge对象的权重都要大于等于零，若是权重 weight == 0，那么该边之后会被删去。

```
// Representation invariant:
// vertex can point to itself
// the weight have to be positive ( >= 0)
// if weight is 0, it will be removed later.
```

对于Safety from rep exposure，首先用private和final来表示fields，其次要使

数据不会泄露，外部无法修改内部实现。

```
    // Safety from rep exposure:
    // Check the rep invariant is true. That means setting
    // source, target and weight unchangeable / private / final
    // add return value MUST be immutable
```

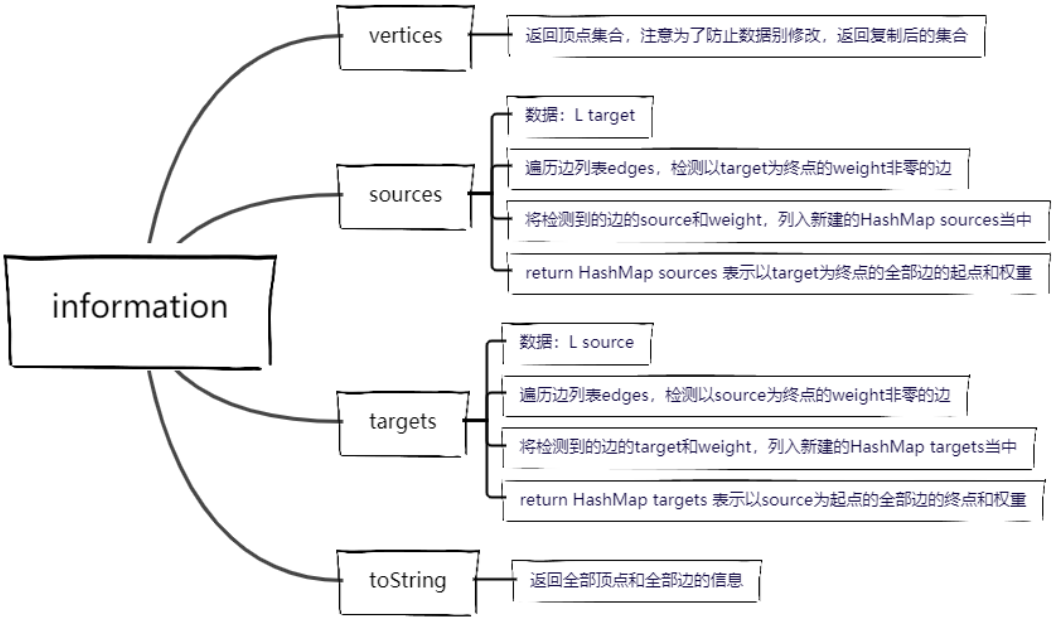
2. 实现 ConcreteEdgesGraph

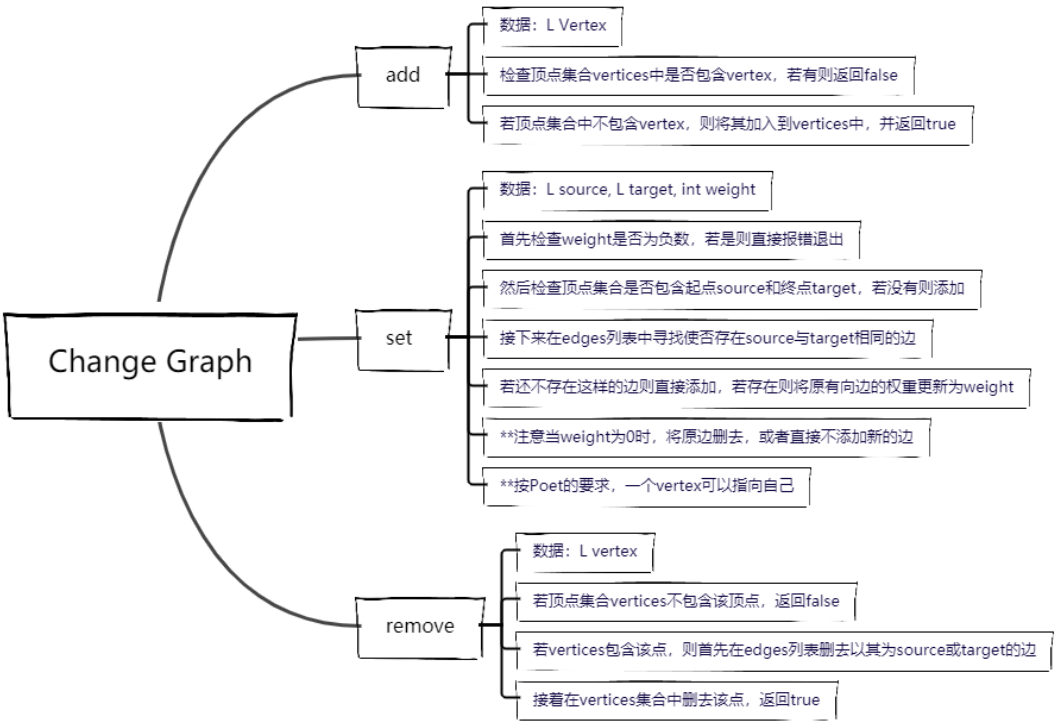
首先在 Graph.java 中我们以及清楚了重写 methods 的要求和含义，按要求有以下数据结构，那么基于此我们有 AF、RI 以及实现方法：

```
    private final Set<L> vertices = new HashSet<>();
    private final List<Edge<L>> edges = new ArrayList<>();
```

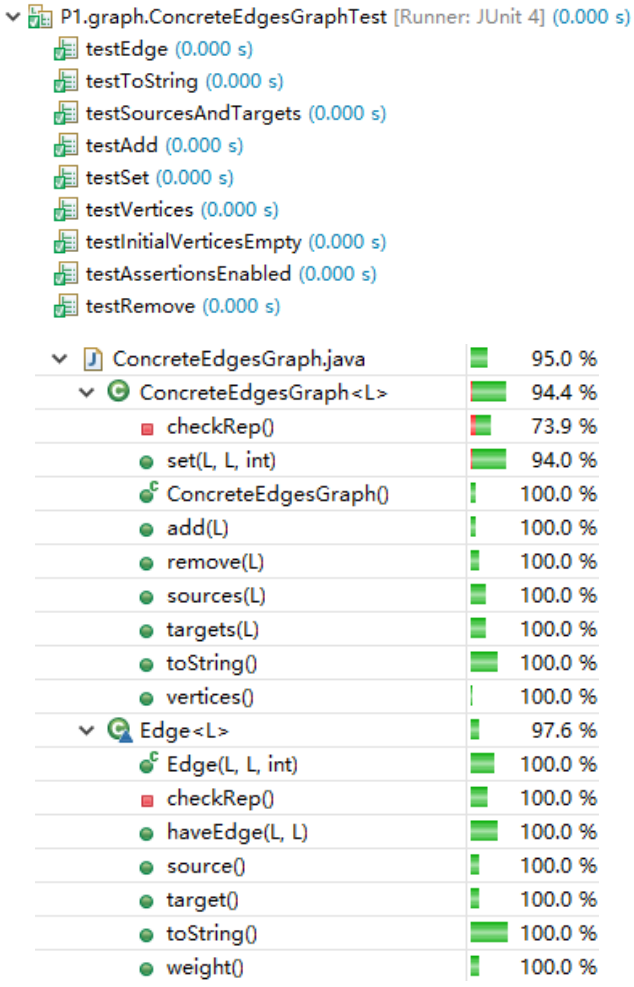
```
    // Abstraction function:
    // AF(vertices) = the vertices set of this graph.
    // AF(edges) = the edges list of this graph

    // Representation invariant:
    // one or no edge between the same source and target
    // the source and target of any edge in the edge list must be in the
vertices set

    // Safety from rep exposure:
    // Check the rep invariant is true, but list and set are mutable
    // All fields MUST be private
    // So make defensive copies instead of just return mutable data
```

该类以类 Edge 为基础重写 Graph<L>，如下两张图所示：





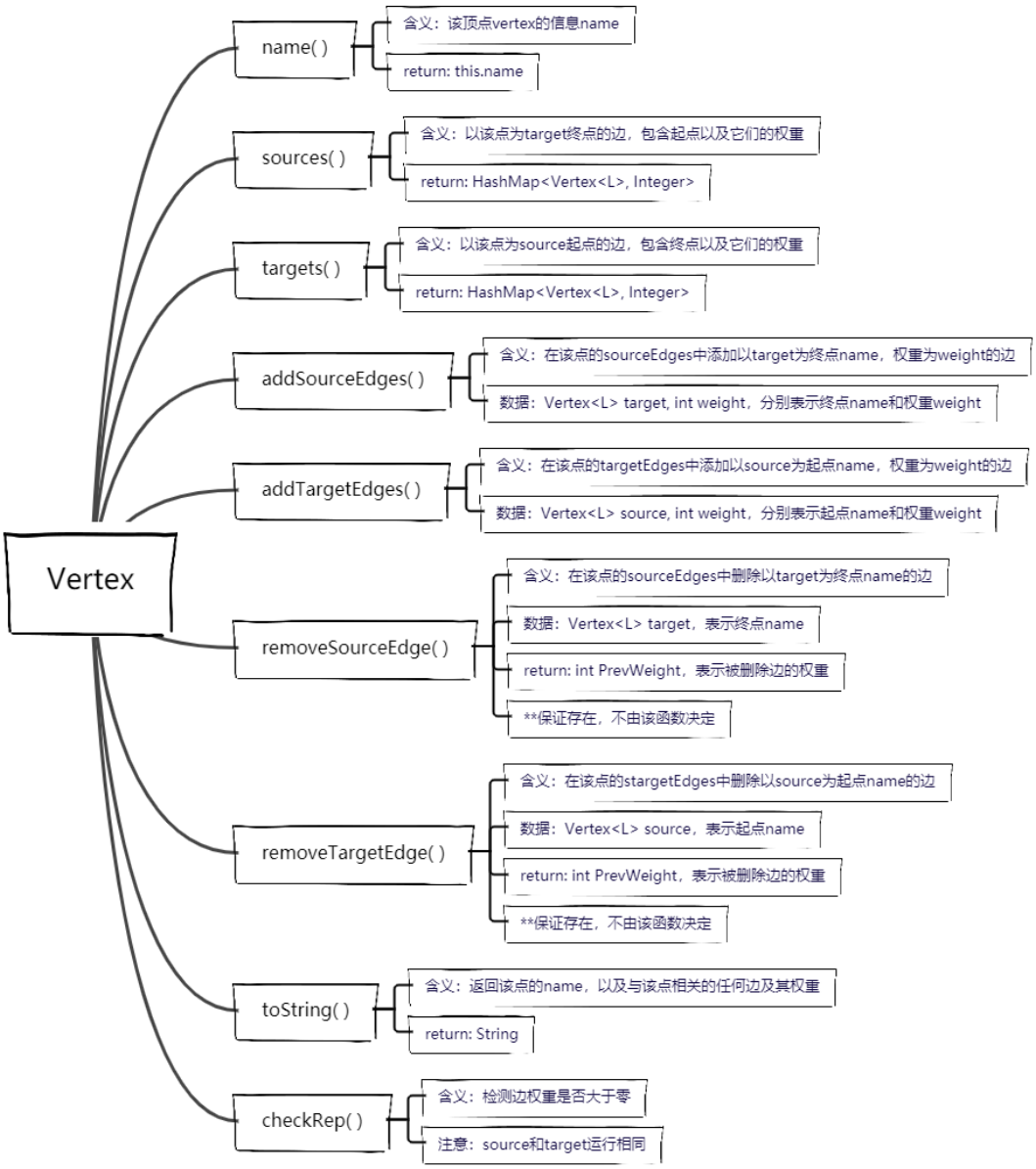
3. ConcreteEdgesGraphTest 测试结果以及覆盖率：



3.1.3.2 Implement ConcreteVerticesGraph

1. 首先设计 Vertex 类：每个 Vertex 中保存有 L name，以 Vertex 为起点的边 targets HashMap，以 Vertex 为终点的边 sources HashMap，储存有其他点和有向边的权重 weight。

主要定义如下几种方法：name、sources、targets、addSourceEdges、addTargetEdges、removeSourceEdges、removeTargetEdges，当然还重写了 toString，写了 private 方法 checkRep，它们的含义、参数、返回值以及注意事项等如下图所示：



首先对于 Vertex 类里面的 fields

```
    private final L name;
    // this vertex as source
    private final Map<Vertex<L>, Integer> sourceEdges = new HashMap<>();
```

```
    // this vertex as target
    private final Map<Vertex<L>, Integer> targetEdges = new HashMap<>();
```

对于Abstraction function，Vertex类中的sourceEdges表示以name为起点的边，存放有targets点以及有向边的权重weight；targetEdges表示以name为终点的边，存放有sources点以及有向边的权重weight；而那么表示该点的信息。

```
    // Abstraction function:
    // AF(sourceEdges) = vertecies to the target and thier weights
    //                   the edges to the vertex
    // AF(targetEdges) = vertecies from the source and thier weight
    //                   the edges from the vertex
```

对于Representation invariant，保证每一个Vertex对象的每一条相关有向边的权重都要大于零，并且A存有A到B的边，那么B中也必须要存有相同weight的边。

```
    // Representation invariant:
    // the weight have to be positive ( > 0)
    // if vertex A has a edge to vertex B, then B must
    // have an edge from A with the SAME weight
```

对于Safety from rep exposure，首先用private和final来表示fields，其次要使数据不会泄露，外部无法修改内部实现，对于返回的HashMap、Set等内容，一定要是返回复制后的数据。

```
    // Safety from rep exposure:
    // Check the rep invariant is true, all fields are private
    // return copied Maps
```

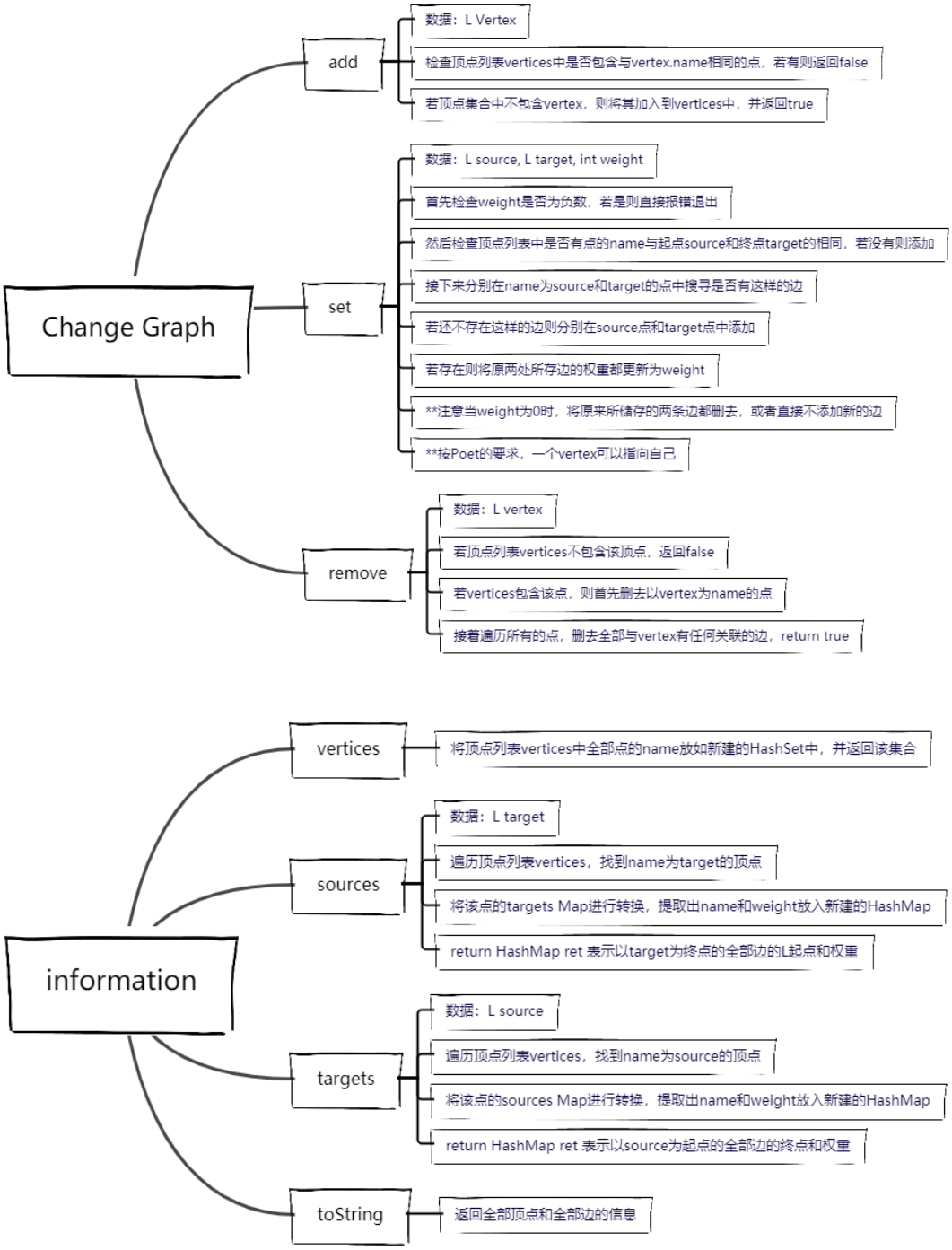
2. 实现 ConcreteVerticesGraph

首先在 Graph.java 中我们以及清楚了重写 methods 的要求和含义，按要求有以下数据结构，那么基于此我们有 AF、RI 以及实现方法：

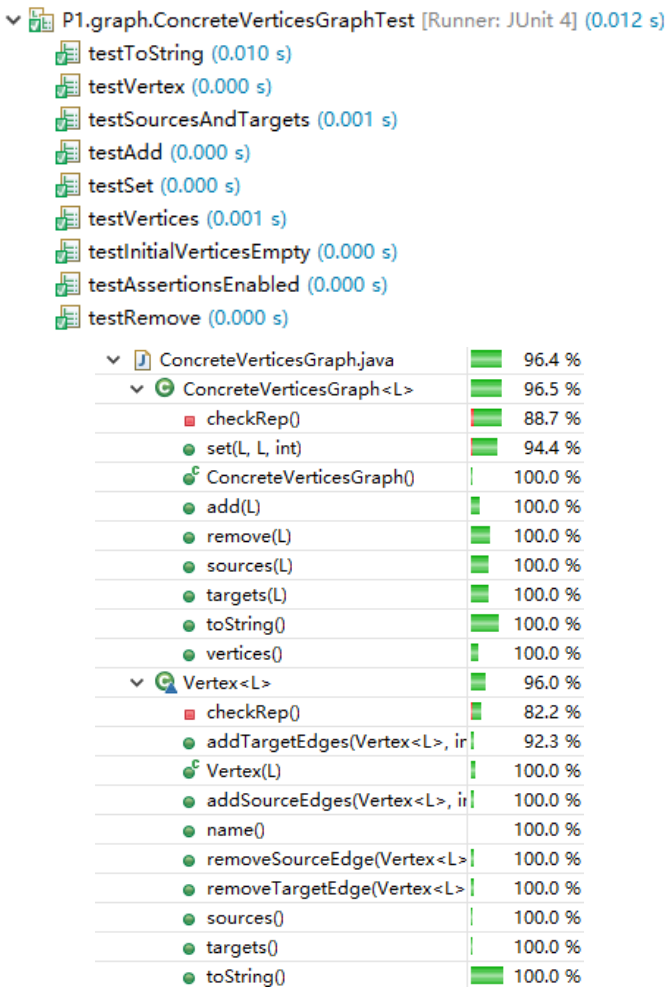
```
    private final List<Vertex<L>> vertices = new ArrayList<>();
```

```
    // Abstraction function:
    // AF(vertices) = the vertices of the graph

    // Representation invariant:
    // Each vertex's name(key) MUST be unique, and cant be null

    // Safety from rep exposure:
    // List is mutable, return copied data, and dont use dangerously
    // All fields MUST be private final
```

该类以类 Vertex 为基础重写 Graph<L>，如下两张图所示：



3. ConcreteVerticesGraphTest 测试结果以及覆盖率：

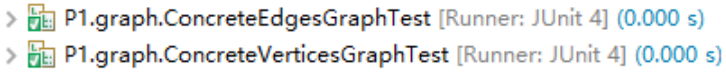


3.1.4 Problem 3: Implement generic `Graph<L>`

在以上的实现中，任何一项都依赖于标签是特定类型 String，即认定 L 为 String。接下来将已有的实现改为基于 L 泛型的即可。

3.1.4.1 Make the implementations generic

注意将所有的实现全部改为泛型实现即可，然后在更改结束后，重新测试 ConcreteVerticesGraphTest 和 ConcreteEdgesGraphTest 两个测试， Graph 在两个泛型实现下仍然可以通过，表示更改成功。



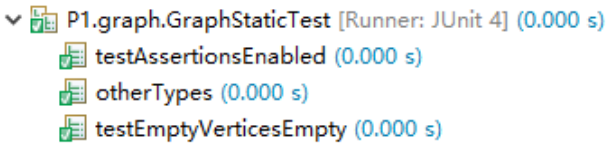
3.1.4.2 Implement `Graph.empty()`

empty 返回一个 Graph 接口的具体实现即可：

```
public static <L> Graph<L> empty() {
return new ConcreteEdgesGraph<L>();
}
```

更改 GraphStaticTest，使用不同泛式，例如基本数据类型（Immutable）中的 Double 和 Integer 类型进行检测，增加检测类型代码和结果如下：

```
// test other vertex label types in Problem 3.2
@Test
public void otherTypes() {
 assertEquals("expected empty() graph to have no vertices",
         Collections.emptySet(), Graph.<String>empty().vertices());
 assertEquals("expected empty() graph to have no vertices",
         Collections.emptySet(), Graph.<Integer>empty().vertices());
 assertEquals("expected empty() graph to have no vertices",
         Collections.emptySet(), Graph.<Double>empty().vertices());
}
```



3.1.5 Problem 4: Poetic walks

根据语库 corpus 生成基于 Words 的有向图，每条边上的权重表示该两个单词联系出现的次数。例如：A—2 —>B，表示 AB 这种形式出现过两次。值得注意的是，允许出现 AA 连续形式。根据生成的 Graph，若有 input，则对其自动补全，即检查两个单词之间是否在 Graph 中找得到另一个单词插入。即相邻单词 A 和 B，在 Graph 中有 A 到 C 和 C 到 B，那么就将 C 插入到 input 中等待输出，若存在多个 C 则选取两边 weight 之和最大的 C。插入到最后得到输出 output。

3.1.5.1 Test GraphPoet

在测试文件 GraphPoetTest.java 中，根据编写的 testing strategy 来实现测试方法，测试包括：corpus 类型（nothing、一行、多行、存在空行），input 类型（nothing、有无插入），output 类型（0/>=1 bridge word、choose heavier bridge word）。测试设计、实现思路和过程如下：

测试函数	测试思路设计
testExample()	deal with the example

testMyExample()	//corpus filename: seven-words.txt
	//corpus: free to add additional methods.
	//input: I want to add methods.

testEmptyInput()	//corpus filename: inputNothing.txt
	//corpus(I dont care):
	//input nothing

testEmptyCorpus()	//corpus filename: corpusNothing.txt
	//corpus nothing
	//input: I want to add methods.

testOneLineTreeOneWord()	//corpus filename: severalLines.txt
	//corpus:
	Hugo is one of the most popular open-source static site generators.
	With its amazing speed and flexibility, Hugo makes building websites fun again.
	//input: The popular hugo can makes websites with amazing speed.

testOneLineTreeSeveralWords()	//corpus filename: corpusToSelf.txt
	//corpus: Hello, hello, hello, Hello, my name is Fu Haodong.
	//input: I want to add methods.

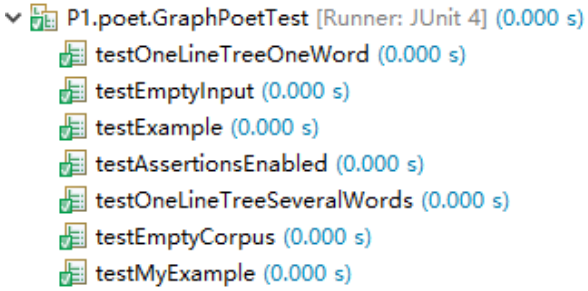
3.1.5.2 Implement GraphPoet

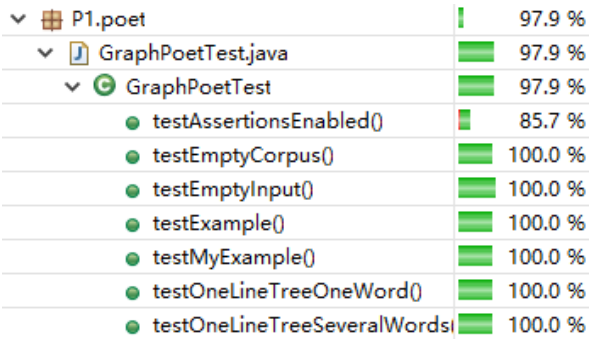
GrapPoet()方法：对于语库 corpus，先一行一行地读取，将内容全部利用 toLowerCase()方法转化为小写字母，再用 split(“ ”)方法分为一个个的单词，若单词后有标点符号，则标点符号也算入该单词，根据每一行的 String[] words 生成加权有向图。

Poem()方法：将 input 用 split(“ ”)方法分为一个个的单词，依次在 Graph 中查询是否在 inputWords[i]. toLowerCase()和 inputWords[i+1]. toLowerCase()之间的第三个单词，若有则插入，直到 inputWords 中的每一个单词对都被查询完。

Private void checkRep()：首先保证图不是空的、顶点表中的顶点不是空的且全是小写字母，最后保证每条边权重至少为 1.

测试结果及覆盖率：





3.1.5.3 Graph poetry slam

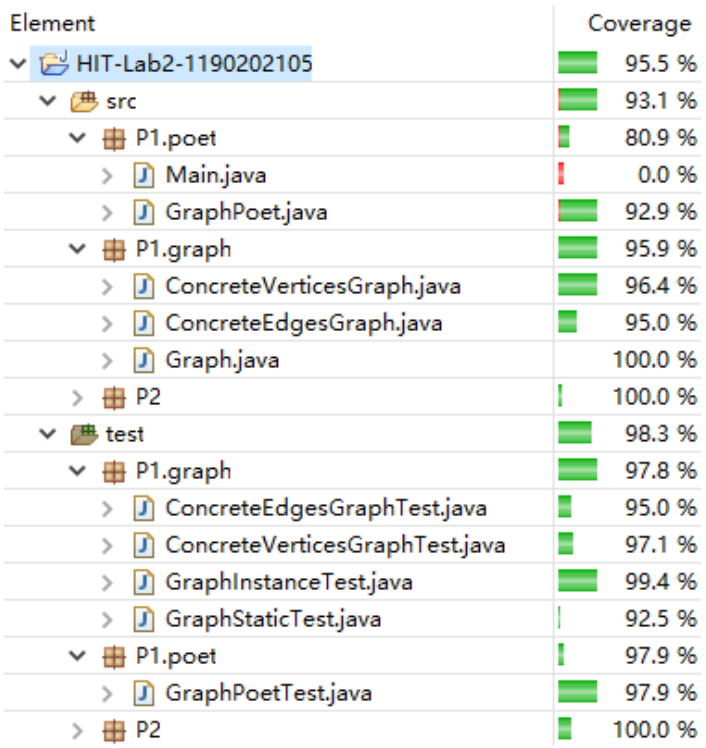
在 GraphPoet.java 的 main 函数中添加一个自己的测试，搜索一首英文诗。

```
final GraphPoet myNimoy = new GraphPoet(new File("src/P1/poet/jellicles-song.txt"));
final String myInput = "Jellicle are quit in some time.";
System.out.println(myInput + "\n>>>\n" + myNimoy.poem(myInput));
```

运行结果如下所示：

```
Jellicle are quit in some time.
>>>
Jellicle cats are quit in some time.
```

3.1.6 使用 Eclemma 检查测试的代码覆盖度



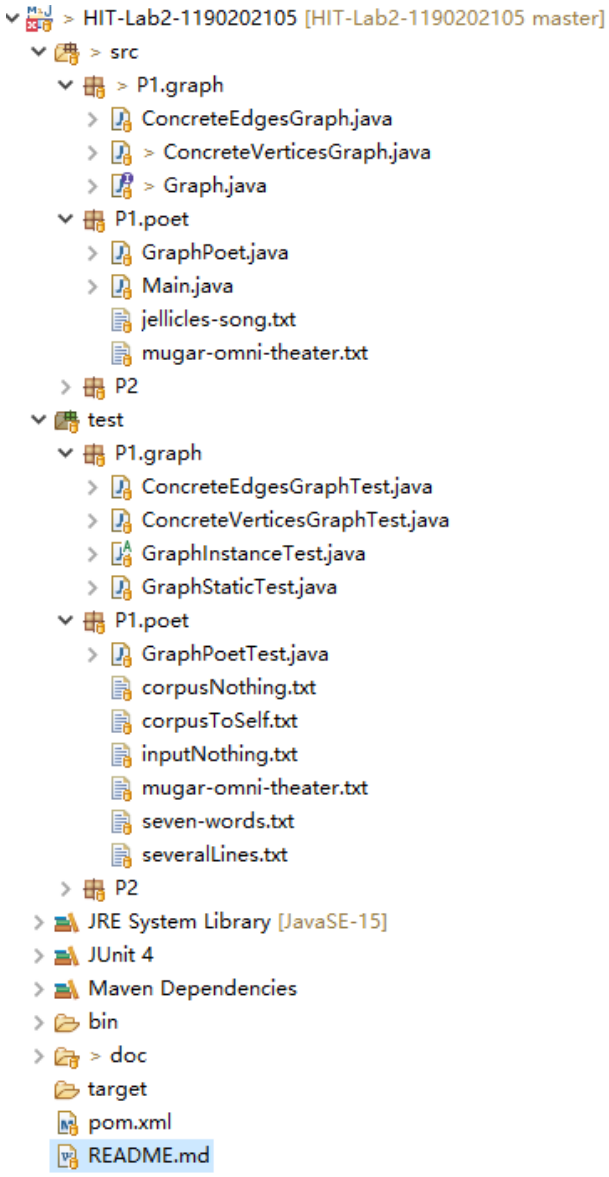
3.1.7 Before you're done

请按照 http://web.mit.edu/6.031/www/sp17/psets/ps2/#before_youre_done 的说明，检查你的程序。

通过 Git 提交当前版本到 GitHub 上你的 Lab2 仓库。

```
Git add .
Git commit -m "update"
Git push -u origin master
```

项目的目录结构树状示意图：

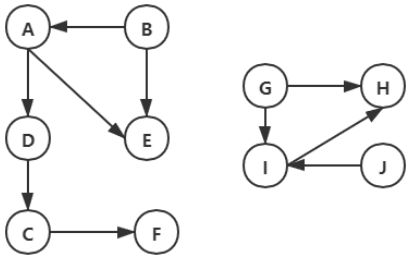


3.2 Re-implement the Social Network in Lab1

利用上述实现的 ADT，重新实现实验一中的 FriendshipGraph，令 L 泛型为实验一中的 Person，改变两个 java 类，通过本实验中的 ConcreteEdgeGraph 或者 ConcreteVertexGraph 来快速实现。

3.2.1 `FriendshipGraph` 类

首先创建一个新的 Graph，将 Person 添加到 vertices 中，添加朋友关系，形成有向边。接下来按照给出的例子，要对 Person、addVertex、addEdge、getDistance 进行分析。



3.2.1.1 Method addVertex()

创建节点的时候，直接使用 Graph 的 add 操作，与本操作十分吻合。

```
// add a new convex
public boolean addVertex (Person newPerson) {
    return graph.add(newPerson);
}
```

3.2.1.2 Method addEdge()

addEdge 的功能是为不同的顶点之间添加边，由于类要保留拓展到有向图的功能，所以每一次运行 addEdge 都只会添加一条单向边。利用 Graph 中的 set 操作，只不过要注意 set 是允许相同顶点的指向。而且此时不必使用太过复杂的权重，用 0 和 1 来表示有无边即可。

```
// add a new edge between two vertexes
public int addEdge (Person personA, Person personB) {
    if (personA.name().equals(personB.name())) return -1;
    int prevWeight;
    prevWeight = graph.set(personA, personB, 1);
    return prevWeight;
}
```

3.2.1.3 Method getDistance()

BFS 算法：从原始顶点开始一层一层地进行搜索，每经过一次顶点标记为 visited（这里设置一个 HashSet<Person> visited），下经过就直接跳过，并且把它的非终点全部入队；每搜索完一层则 distance++；若达到终点则返回层数。若最终队列里面不存在任何顶点则表示没有从原点到终点的路径。程序最开始还对两点的相同性进行检测，若是相同直接返回 0.

```
// get the distance between the two vertexes, BFS
public int getDistance(Person personA, Person personB) {
    // nobody is a friend of oneself
    HashSet<Person> visited = new HashSet<Person>();
    // the distance is 0, when they are the same person
    if (personA == personB)
        return 0;
    int temp, distance = 0;
    visited.add(personA);
    LinkedList<Person> queue = new LinkedList<Person>();
    queue.addAll(graph.targets(personA).keySet());
    int levelSize = queue.size();
    do {
        temp = 0;
        distance ++;
        for (int i = 0; i < levelSize; i++) {
            if (queue.getFirst() == personB)
                return distance;
            else if (visited.contains(queue.getFirst()))
                queue.removeFirst();
            else {
                // all the friends of the guy enter the queue
                queue.addAll(graph.targets(queue.getFirst()).keySet());
                temp += graph.targets(queue.getFirst()).size();
                // set the guy as visited
                visited.add(queue.getFirst());
                queue.removeFirst();
            }
        }
        // all the reachable are visited, but person 2 not found
        if (temp == 0) return -1;
        levelSize = temp;
    } while(true);
    }
```

3.2.2 Person 类

每个人对应到一个 Person 对象，且至少包含人的名字，定义方法返回名字 name()。

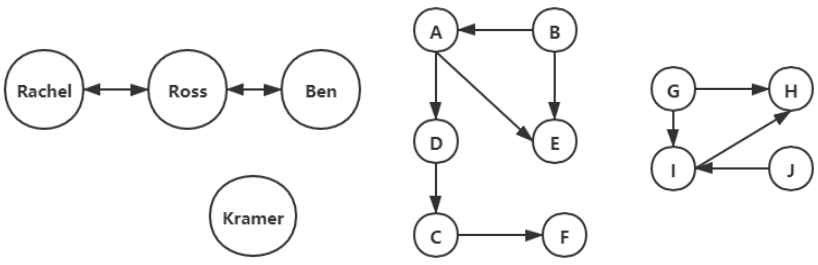
```
package P2;
public class Person {
    private final String name;
    // Abstraction function:
    // AF(name) = the name of the person

    // Safety from rep exposure:
    // all fields are final and private

    // constructor
    public Person(String str) {
        this.name = str;
    }

    // metheds
    public String name() {
        return name;
    }
}
```

3.2.3 客户端 main()

由于只是改变了 methods 的实现方法，直接用实验一中的 main 客户端。

3.2.4 测试用例

给出两个测试，首先按照下图进行创建关系：

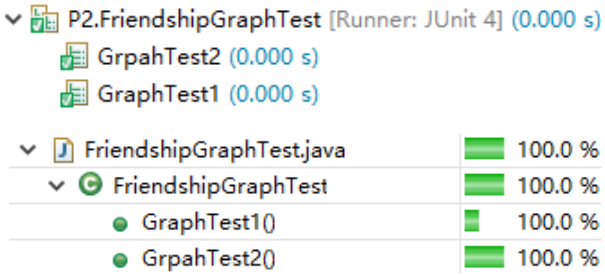


它们的测试用例分别如下：

```
assertEquals(1, graph.getDistance(rachel, ross));
assertEquals(2, graph.getDistance(rachel, ben));
assertEquals(0, graph.getDistance(rachel, rachel));
```

```
        assertEquals(-1, graph.getDistance(rachel, kramer));
        assertEquals(3, graph.getDistance(a, f));
        assertEquals(-1, graph.getDistance(a, b));
        assertEquals(-1, graph.getDistance(e, f));
        assertEquals(2, graph.getDistance(d, f));
        assertEquals(-1, graph.getDistance(a, j));
        assertEquals(0, graph.getDistance(i, i));
        assertEquals(-1, graph.getDistance(g, j));
        assertEquals(1, graph.getDistance(i, h));
```

检测结果及覆盖率：

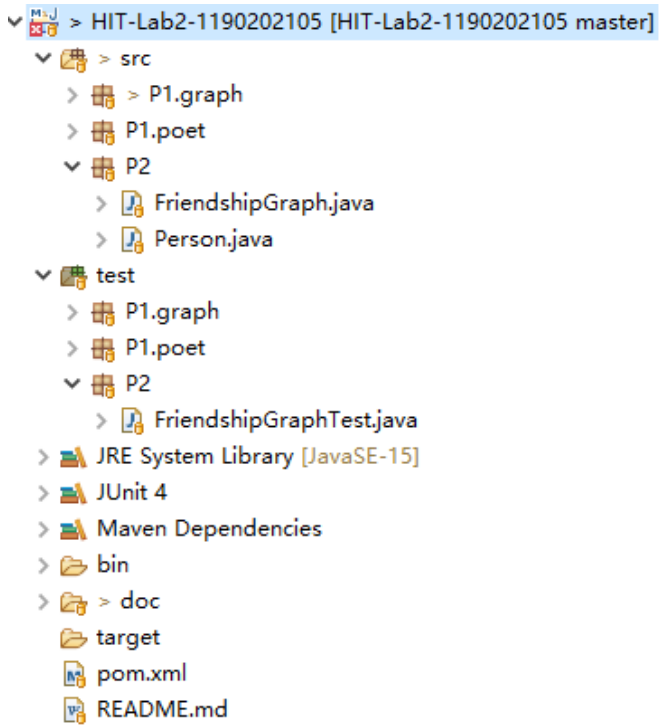


3.2.5 提交至 Git 仓库

通过 Git 提交当前版本到 GitHub 上 Lab2 仓库。

```
Git add .
Git commit -m “update”
Git push -u origin master
```

给出项目的目录结构树状示意图。



4 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。

日期	时间段	计划任务	实际完成情况
2021.05.30	晚	初始化项目、了解项目内容等	遇到困难，未完成
2021.05.31	晚	3.1 Problem 1: Test Graph<String>	遇到困难，未完成
2021.05.31	晚	3.1 Problem 2: ConcreteEdgesGraph	超时完成
2021.06.03	晚	3.1 Problem 2: ConcreteVerticesGraph	按时完成
2021.06.03	早	3.1 Problem 3: Generic Graph<L>	按时完成
2021.06.03	晚	3.1 Problem 4: Poet and Poet Test	按时完成
2021.06.04	早	3.1 Problem 1: Test Graph<String>	未完成
2021.06.04	晚	3.1 Problem 1: Test Graph<String>	几乎完成
2021.06.05	早	3.2 Re-implement Social Network	完成
2021.06.05	午	撰写报告	未完成
2021.06.05	晚	撰写报告	完成

5 实验过程中遇到的困难与解决途径

遇到的难点	解决途径
在实现 List 的选择删除时，不能使用增强 for 循环遍历	改用方法 removeIf

list，删除元素后会有异常抛出。	
对于泛型的不理解	查阅资料后有了一定的理解，但还是一知半解
对于基于 Vertex 实现的图，在进行 remove 操作时，难以实现在删除点之前就将相关边给删去。由此对 HashMap 的实现存疑。	没有解决，但是我先 remove 该点，之后进一步清理图中该删去的边。
对于 equals 和 HashCode 是否重写不清楚	了解后续课程的内容
不习惯于先写测试用例，在写的过程中对于不是自己定义的方法使用不习惯	在完成相关类的实现之后进一步修改测试用例

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

1. 掌握了一定从应用场景到 ADT 的抽象映射的能力；
2. Java 语言的有些类方法的使用不够熟练，甚至很多类方法都不知道，导致在实验过程中浪费了很多时间，有些代码写得过于冗长；

6.2 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？

在面向 ADT 编程的时候需要考虑代码的可复用性，需要使整个编程更加具有普适性；面向应用场景可以直接根据要求来编写程序，更易于操作，但是程序复用性很低，难以在其他场景进行复用。

(2) 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，不使用泛型的编程更加顺手和便于理解，但是使用泛型能够兼容所指定的类型，所以应用范围会更加广。

(3) 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？

优势是能够尽早找到程序中的错误，避免错误累积到后续难以修改；但是不太能适应这种测试方法，首先是方法规约于测试内容联系不上，再来就是测试用例有时不太能考虑到全部情况。

(4) 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？

在抽象编程的基础上，利用已有代码，避免重复工作，节省编程时间，减少开发成本。

(5) 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？

对于应用场景到 ADT 的抽象映射，其实难度不大，但是还是不熟悉 rep 方法等。

(6) 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？

意义在于避免数据外泄、指导程序编写的准确性和健壮性的提高，规避错误。愿意坚持注意是否有 rep exposure，由此来保护内容的私密性。

(7) 关于本实验的工作量、难度、deadline。

实验与课程进度不相符合，实验往往要落后课程一段时间，由此如果要按时完成实验在读懂要求上消耗的时间量是巨大的。由此导致工作量等等都很大，但是 deadline 也还能接受。

(8) 《软件构造》课程进展到目前，你对该课程有何体会和建议？

课程到目前为止，绝大多数内容都显得很抽象，在此之前对于 Java 的了解也没有那么多，所以短时间理解起来还是比较难的。

建议课程任务降低一点点，实验内容比例升高。