



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

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
手机号码	13383324846

目录

1 实验目标概述	1
2 实验环境配置	1
3 实验过程	4
3.1 Poetic Walks	4
3.1.1 Get the code and prepare Git repository	4
3.1.2 Problem 1: Test Graph <String>	5
3.1.3 Problem 2: Implement Graph <String>	7
3.1.3.1 Implement ConcreteEdgesGraph	7
3.1.3.2 Implement ConcreteVerticesGraph	10
3.1.4 Problem 3: Implement generic Graph<L>	13
3.1.4.1 Make the implementations generic	13
3.1.4.2 Implement Graph.empty()	14
3.1.5 Problem 4: Poetic walks	16
3.1.5.1 Test GraphPoet	17
3.1.5.2 Implement GraphPoet	17
3.1.5.3 Graph poetry slam	18
3.1.6 使用 Eclemma/Code Coverage for Java 检查测试的代码覆盖度	19
3.1.7 Before you're done	20
3.2 Re-implement the Social Network in Lab1	21
3.2.1 FriendshipGraph 类	21
3.2.2 Person 类	23
3.2.3 测试用例	23
3.2.4 提交至 Git 仓库	24
4 实验进度记录	24
5 实验过程中遇到的困难与解决途径	25
6 实验过程中收获的经验、教训、感想	25
6.1 实验过程中收获的经验教训（必答）	26
6.2 针对以下方面的感受（必答）	26

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

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。

```
haoran@luo MINGW64 /f/Git_Tools/gitRep/HIT-Lab2-2022113415 (master)
$ git pull
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 1018 bytes | 63.00 KiB/s, done.
From https://github.com/ComputerScienceHIT/HIT-Lab2-2022113415
* [new branch]      master      -> origin/master
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details.

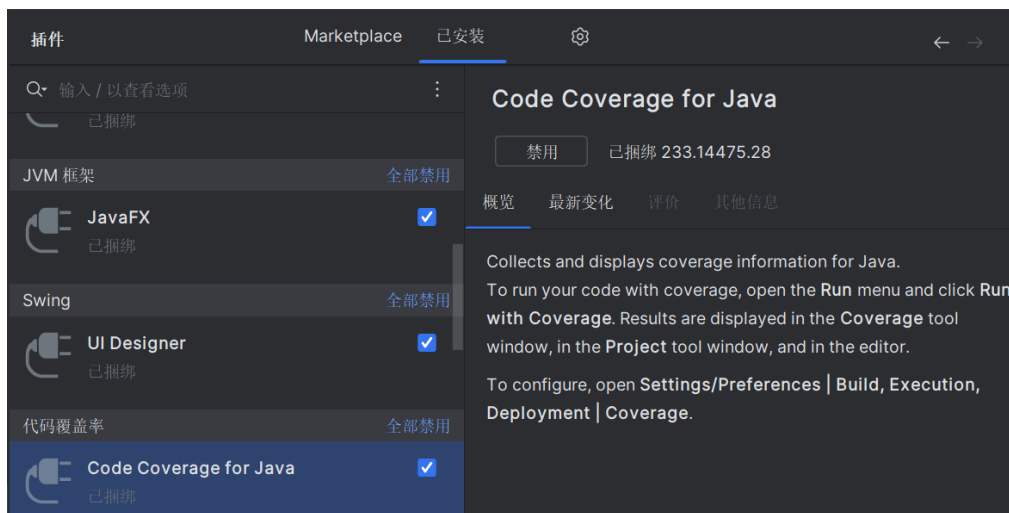
git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

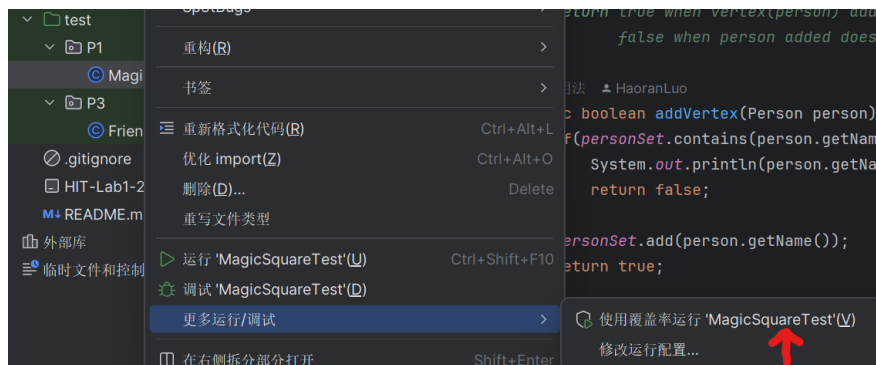
git branch --set-upstream-to=origin/<branch> master

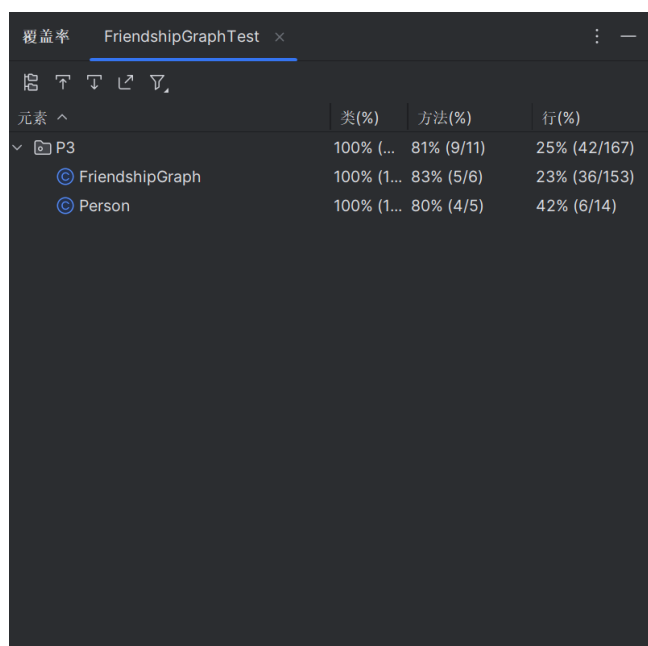
haoran@luo MINGW64 /f/Git_Tools/gitRep/HIT-Lab2-2022113415 (master)
$ git pull --rebase origin master
From https://github.com/ComputerScienceHIT/HIT-Lab2-2022113415
* branch            master      -> FETCH_HEAD
```

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。



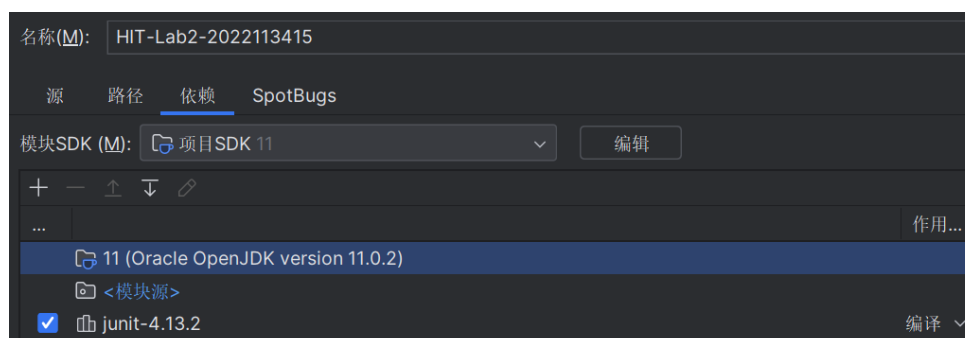
如图, 这里在已经完成的 Lab1 项目中进行测试。对于已经实现的可以运行的类文件, 通过右击要运行的类文件, 在“更多运行/调试”中找到“使用覆盖率运行”即可。在类、方法与代码行三个维度的覆盖度测试结果将在 IDEA 界面的右侧显示。





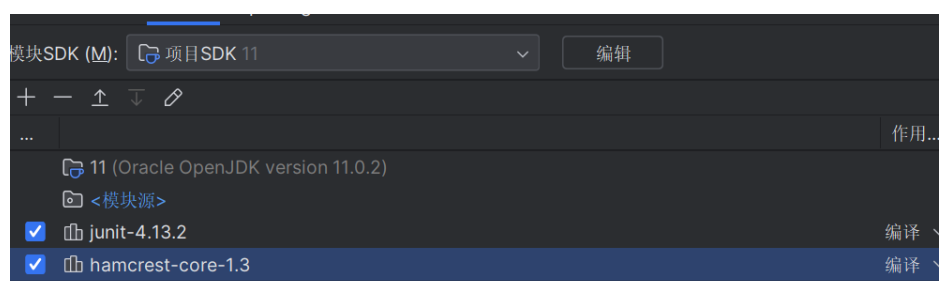
元素	类(%)	方法(%)	行(%)
P3	100% (...)	81% (9/11)	25% (42/167)
FriendshipGraph	100% (1...)	83% (5/6)	23% (36/153)
Person	100% (1...)	80% (4/5)	42% (6/14)

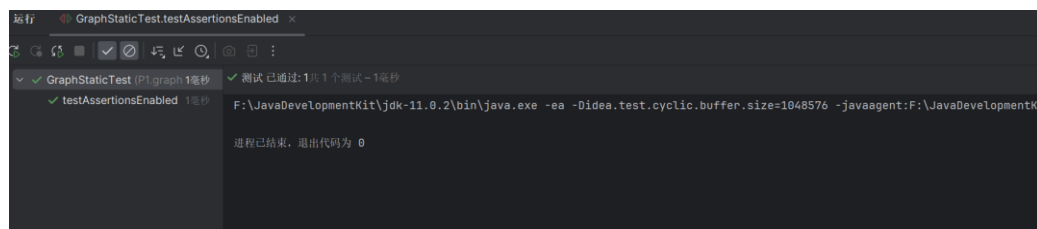
对新建的 Lab2 项目配置项目结构, 导入 Junit 库用于测试, 步骤同 Lab1。运行 Junit 测试类是出现如下报错, 查阅资料发现系手动导入 Junit 库时缺少 hamcrest-core 包导致, 联网下载后导入后, Junit 库正常运行。



```
java.lang.NoClassDefFoundError: org/hamcrest/SelfDescribing

    at java.base/java.lang.ClassLoader.defineClass1(Native Method)
    at java.base/java.lang.ClassLoader.defineClass(ClassLoader.java:1016)
    at java.base/java.security.SecureClassLoader.defineClass(SecureClassLoader.java:174) <5 个内部行>
    at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:521)
    at org.junit.internal.builders.JUnit4Builder.runnerForClass(JUnit4Builder.java:10) <1 个内部行>
```





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>

3 实验过程

3.1 Poetic Walks

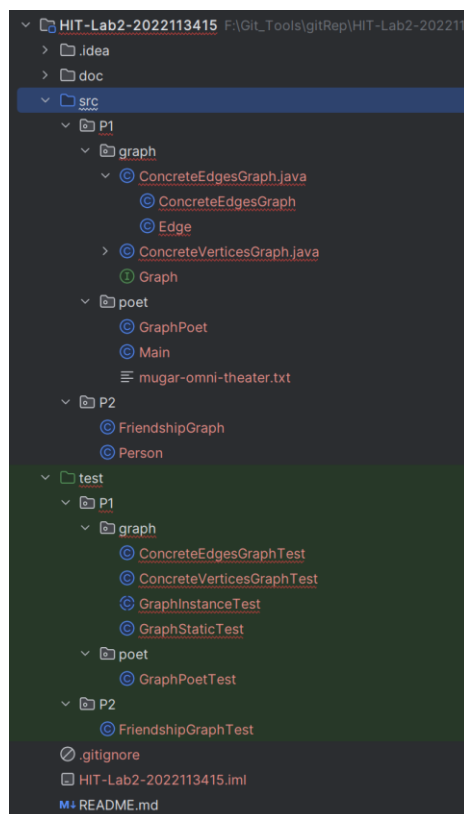
任务要求我们实现 Graph 接口下的两个实现类 ConcreteEdgesGraph 和 ConcreteVerticesGraph 中的各个方法, 编写有关测试用例, 撰写 AF、RI、预防表示泄露措施等, 之后编写实现代码, 并将这两个实现类扩展到泛型<L>, 之后应用编写的抽象数据类型解决 PoeticWalks 问题。

3.1.1 Get the code and prepare Git repository

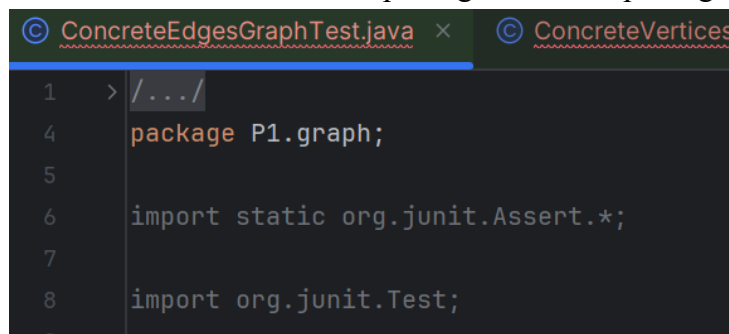
启动 git Bash, cd 进入当前项目目录, 使用 git clone 命令从指定的链接获取初始代码, 后按照要求的项目结构调整项目目录。

```
haoran@luo MINGW64 /f/Git_Tools/gitRep/HIT-Lab2-2022113415 (master)
$ git clone https://github.com/rainywang/Spring2022_HITCS_SC_Lab2.git
Cloning into 'Spring2022_HITCS_SC_Lab2'...
remote: Enumerating objects: 57, done.
remote: Total 57 (delta 0), reused 0 (delta 0), pack-reused 57
Receiving objects: 100% (57/57), 14.81 KiB | 219.00 KiB/s, done.
Resolving deltas: 100% (11/11), done.

haoran@luo MINGW64 /f/Git_Tools/gitRep/HIT-Lab2-2022113415 (master)
$
```



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。

3.1.2 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：

```
-testAdd(): 测试Graph接口内add方法
考虑两种等价类：添加的点未加入图/已加入图
在同一方法中完成测试

-testSet(): 测试Graph接口内set方法
**方法应当在外部确保weight输入不为负值**
考虑5个维度：顶点source在图中/不在图中 顶点target在图中/不在图中
source与target相同/不同
输入值weight为非0正值/0
当前图中从source顶点指向target顶点有边/无边
要使得source与target相同 必须有source target都在图中/都不在图中
而source target都不在图中时 必定没有在图中的从source顶点指向target顶点的边
故source与target相同情况下有 $2 \times 2 + 2 = 6$ 个等价类
考虑source与target不相同的情况，也即一般情况
--source target不满足都在图中时 必定没有在图中的从source顶点指向target顶点的边
此时有 $3 \times 2 \times 1 = 6$ 个等价类
--source target都在图中时 考虑当前图中从source顶点指向target顶点有边/无边
此时有 $2 \times 2 \times 1 \times 1 = 4$ 个等价类
整理得到总计有 $6 + 6 + 4 = 16$ 个等价类
设计testSet()方法1~16

-testRemove(): 测试Graph接口内remove方法
考虑两个维度：要移除的顶点在图中/不在图中 要移除的顶点在图中有边/无边
除去矛盾情况，整理得到总计有3个等价类
设计testRemove()方法1~3

-testVertices(): 测试Graph接口内的vertices方法
考虑两种等价类：图中有顶点/无顶点
对有顶点的情况进一步展开测试：测试图中有小规模顶点/较大规模顶点的情况(以20个为界限)
总计3个等价类
在同一方法中完成测试

-testSources(): 测试Graph接口内的sources方法
由于无向图可以看做特殊有向图 故对有向图的一般情况测试即可
考虑顶点target有/无入边和顶点target在图中/不在图中
整理得到3个等价类，在同一方法中完成测试

-testTargets(): 测试Graph接口内的targets方法
由于无向图可以看做特殊有向图 故对有向图的一般情况测试即可
考虑顶点source有/无出边和顶点source在图中/不在图中
整理得到3个等价类，在同一方法中完成测试
```

完成后提交至 Git。

3.1.3 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。

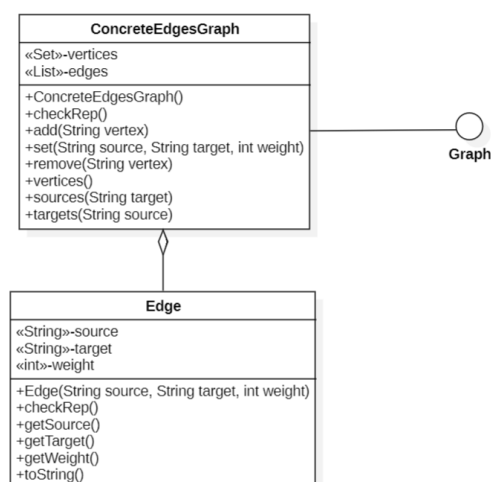
在两种实现的代码中需要:

- 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 `toString` 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。

3.1.3.1 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):

```
// Abstraction function:
//   AF(vertices) = Vertices in actual graph
//   AF(edges) = Edges in actual graph
//
// Representation invariant:
//   每个顶点的关键字（名称）唯一确定
//   边允许自达 但从某一顶点指向一个顶点最多允许存在一条有向边
//   边的权值为非0正值 所有权值为0的边视为不存在
//
// Safety from rep exposure:
//   对返回的内部变量采用防御式拷贝，预防类内部泄露
//   ConcreteEdgesGraph类内全部属性采用private final修饰
```

```
// Abstraction function:
//   AF(source) = 当前边出发的点
//   AF(target) = 当前边指向的点
//   AF(weight) = 当前有向边的权值
// Representation invariant:
//   weight > 0
//   顶点 source & target 必定在实际图中
//   source 可以等于 target
// Safety from rep exposure:
//   'private final' 修饰
//   返回的数据结构均为immutable 不能通过外部修改影响内部
```

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。

```
// checkRep
5 个用法  👤 Luo Haoran
public void checkRep(){
    for(Edge edge: edges){
        assert edge.getWeight()>0;
        assert vertices.contains(edge.getSource());
        assert vertices.contains(edge.getTarget());
    }
}
```

```
// checkRep
4 个用法  👤 Luo Haoran
public void checkRep(){
    assert this.source != null;
    assert this.target != null;
    assert this.weight > 0;
}
```

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：

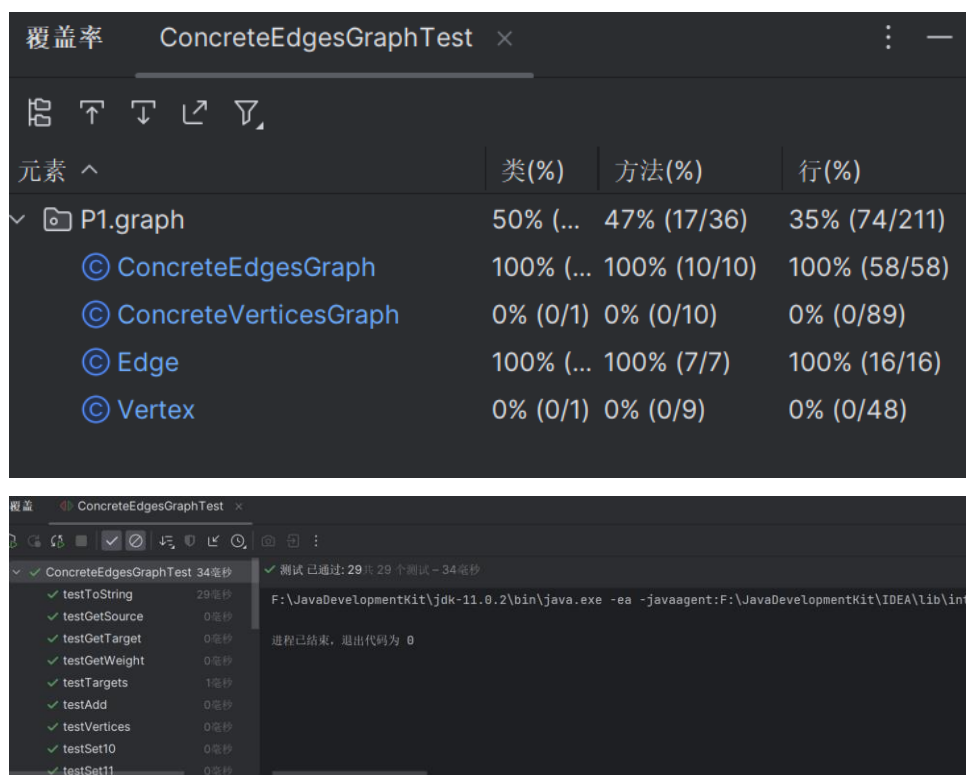
```
// Testing strategy for ConcreteEdgesGraph.toString()
// 考虑顶点被边连接和顶点不被边连接的情况 在同一个方法中实现两个等价类
```

```
// Testing strategy for Edge
// 直接测试Edge类中的三个方法：获取起点 终点与此边权重

// tests for operations of Edge
// 测试"获取起点"方法返回值正确
新 *
@Test
public void testGetSource(){
    Edge edge = new Edge( source: "start", target: "end", weight: 6);
    assertEquals( expected: "start", edge.getSource());
}
// 测试"获取终点"方法返回值正确
新 *
@Test
public void testGetTarget(){
    Edge edge = new Edge( source: "start", target: "end", weight: 6);
    assertEquals( expected: "end", edge.getTarget());
}
// 测试"获取权重"方法返回值正确
新 *
@Test
public void testGetWeight(){
    Edge edge = new Edge( source: "start", target: "end", weight: 6);
    assertEquals( expected: 6, edge.getWeight());
}
```

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）。

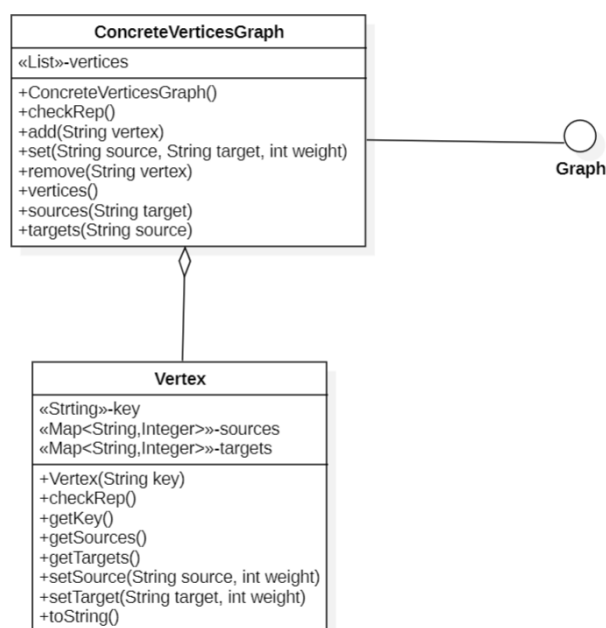
使用覆盖率运行 ConcreteEdgesGraphTest 类，由于它是 GraphInstanceTest 类的 extend，故运行时一并运行 GraphInstanceTest 测试类中的测试方法，得到测试结果以及覆盖率状况如下：



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

3.1.3.2 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):

```
// Abstraction function:
//   AF(vertices) = 图中的各个顶点及他们之间相互的父节点子节点关系
// Representation invariant:
//   vertices中每个顶点的key唯一确定
// Safety from rep exposure:
//   "private final" + 返回Map、Set类型数据时的防御式拷贝

// Abstraction function:
//   AF(key) = 结点的关键字 (名字)
//   AF(sources) = 此节点全部二元组<当前结点的某父节点, 从该父节点指向当前结点有向边的权重>的集合
//   AF(targets) = 此节点全部二元组<当前结点的某子节点, 从当前结点指向该子节点有向边的权重>的集合
// Representation invariant:
//   所有边权值均>0
//   任意Vertex的key关键字唯一确定(在ConcreteVerticesGraph类中保证)
//   允许自达边存在 但两点之间同一方向的边最多只许存在一条(数据结构保证)
//
// Safety from rep exposure:
//   对返回的Map类型数据sources与targets采用防御性拷贝
//   Vertex类内属性均采用private final修饰
```

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:

```
// Testing strategy for ConcreteVerticesGraph.toString()
//   考虑顶点被边连接和顶点不被边连接的情况 在同一个方法中实现两个等价类

// Testing strategy for Vertex
/*
  对Vertex类中其他方法进行测试
  -getKey() 测试中直接验证对比key即可
  -setSource()
    考虑两个维度: 有/没有source指向当前顶点的边 weight为0/非0正值
    ConcreteVerticesGraph已确保source在图中 checkRep确保weight不小于0
    整理得到4个等价类 设计testSetSource() 1~4
  -setTarget()
    考虑两个维度: 有/没有当前顶点指向target的边 weight为0/非0正值
    ConcreteVerticesGraph已确保source在图中 checkRep确保weight不小于0
    整理得到4个等价类 设计testSetTarget() 1~4
  -getSources()
    考虑一个维度: 当前结点有/无父结点 共2个等价类
    在同一方法中测试
  -getTargets()
    考虑一个维度: 当前结点有/无子结点 共2个等价类
    在同一方法中测试
*/
```

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。

```
// checkRep
8 个用法  👤 Luo Haoran
public void checkRep(){
    Set<Vertex> verticesSet = new HashSet<>(vertices);
    assert verticesSet.size() == vertices.size();
}
```

```
// checkRep
2 个用法  👤 Luo Haoran
public void checkRep(){
    Set<String> cur_sources = sources.keySet();
    Set<String> cur_targets = targets.keySet();
    if(!cur_sources.isEmpty()){
        for(String source: cur_sources)
            assert sources.get(source) > 0;
    }
    if(!cur_targets.isEmpty()){
        for(String target: cur_targets)
            assert targets.get(target) > 0;
    }
}
```

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:

```
▼ graph 40% 类, 64% 行已覆盖
  ▼ ConcreteEdgesGraph.java
    ◎ ConcreteEdgesGraph 0% 方法, 0% 行已覆盖
    ◎ Edge 0% 方法, 0% 行已覆盖
  ▼ ConcreteVerticesGraph.java
    ◎ ConcreteVerticesGraph 100% 方法, 100% 行已覆盖
    ◎ Vertex 100% 方法, 100% 行已覆盖
  ⓘ Graph 0% 方法, 0% 行已覆盖
```



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

3.1.4 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。

3.1.4.1 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。

```
13 1 个用法  Luo Haoran *  
14 public class ConcreteVerticesGraph<L> implements Graph<L> {  
15     13 个用法  
    private final List<Vertex<L>> vertices = new ArrayList<>();  
  
    // checkRep  
    2 个用法  Luo Haoran *  
    public void checkRep(){  
        Set<L> cur_sources = sources.keySet();  
        Set<L> cur_targets = targets.keySet();  
        if(!cur_sources.isEmpty()){  
            for(L source: cur_sources)  
                assert sources.get(source) > 0;  
        }  
        if(!cur_targets.isEmpty()){  
            for(L target: cur_targets)  
                assert targets.get(target) > 0;  
        }  
    }  
}
```

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：

```
// tests for operations of Edge
// 测试"获取起点"方法返回值正确
@Luo Haoran *
@Test
public void testGetSource(){
    Edge<String> edge = new Edge<>( source: "start", target: "end", weight: 6);
    assertEquals( expected: "start",edge.getSource());
}
// 测试"获取终点"方法返回值正确
@Luo Haoran *
@Test
public void testGetTarget(){
    Edge<String> edge = new Edge<>( source: "start", target: "end", weight: 6);
    assertEquals( expected: "end",edge.getTarget());
}
// 测试"获取权重"方法返回值正确
@Luo Haoran *
@Test
public void testGetWeight(){
    Edge<String> edge = new Edge<>( source: "start", target: "end", weight: 6);
    assertEquals( expected: 6,edge.getWeight());
}
```

3.1.4.2 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。

```
8 个用法 @Luo Haoran *
public static <L> Graph<L> empty() {
    // 由于外部并不知道内部是如何实现的 我们这里返回任意一个Graph实现的空图即可
    return new ConcreteVerticesGraph<>();
    // throw new RuntimeException("not implemented");
}
```

Empty 方法实现


```

Luo Haoran
@Test
public void testEmptyVerticesEmpty() {
    assertEquals( message: "expected empty() graph to have no vertices",
        Collections.emptySet(), Graph.empty().vertices());
}

// test other vertex label types in Problem 3.2
/*
    考虑1个维度: 是String/Boolean/Integer/Float/Character...类型
*/
@Test
public void testOtherLabelTypes(){
    // 同学那里获取的想法...直接测试空图就行了
    assertEquals( message: "expected empty() graph to have no vertices",
        Collections.emptySet(), Graph.<String>empty().vertices());
    assertEquals( message: "expected empty() graph to have no vertices",
        Collections.emptySet(), Graph.<Boolean>empty().vertices());
    assertEquals( message: "expected empty() graph to have no vertices",
        Collections.emptySet(), Graph.<Character>empty().vertices());
    assertEquals( message: "expected empty() graph to have no vertices",
        Collections.emptySet(), Graph.<Float>empty().vertices());
    assertEquals( message: "expected empty() graph to have no vertices",
        Collections.emptySet(), Graph.<Integer>empty().vertices());
    assertEquals( message: "expected empty() graph to have no vertices",
        Collections.emptySet(), Graph.<Collections>empty().vertices());
}

```

GraphStaticTest 内部分测试方法

以覆盖率运行测试类，结果如下：



覆盖率 GraphStaticTest			
元素 ^	类(%)	方法(%)	行(%)
✓ P1.graph	40% (...)	13% (5/37)	4% (10/210)
ConcreteEdgesGraph	0% (0/1)	0% (0/10)	0% (0/58)
ConcreteVerticesGraph	100% (...)	40% (4/10)	10% (9/87)
Edge	0% (0/1)	0% (0/7)	0% (0/16)
Graph	100% (...)	100% (1/1)	100% (1/1)
Vertex	0% (0/1)	0% (0/9)	0% (0/48)

至此完成泛型<L>实现。

3.1.5 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。

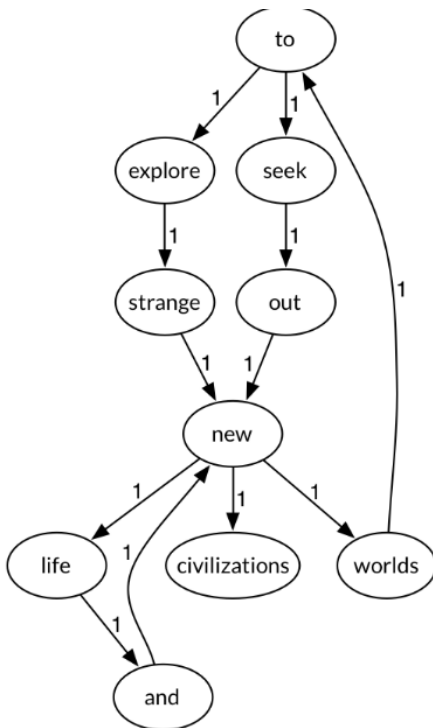
“单词亲和图”中各个顶点为大小写不敏感的单词，它的有向边则代表前一个单词与后一个单词的连接关系，此有向边的边权为前一个单词后面直接邻接后一个单词的情况出现的总次数。

以 MIT 页面的语料库——两行短诗为例：

For example, here's a small but inspirational corpus:

To explore strange new worlds
To seek out new life and new civilizations

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 表示在语料中出现过 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：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。

3.1.5.1 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：

```
// Testing strategy
/*
    testPoem():
    保留原本类内spec中提供的两个测试用例（转为Hello and goodbye.txt和mugar-omni-thater.txt）
    确保代码能够正常运行 并在可通往多个单词顶点的情况下优先选择权值较高的节点
    之后任意选取部分歌词和诗歌 代入用户的身份编写测试用例

    testToString():
    选择任一txt文件作为文件流读入 直接测试
*/
// tests
```

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）。

3.1.5.2 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：

```
// Abstraction function:
//   AF(graph) = 由语料库中全部单词按规则生成的“单词亲和图”
// Representation invariant:
//   graph中边权全部为正
//   完成语料库读入后graph非空
//   不存在内容单词重复的顶点 (由Graph类保证)
// Safety from rep exposure:
//   'private + final'
//   返回值均为Immutable的数据类型
```

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”。

```
70  /**
71   * Create a new poet with the graph from corpus (as described above).
72   * |
73   * @param corpus text file from which to derive the poet's affinity graph
74   * @throws IOException if the corpus file cannot be found or read
75   */
76   9 个用法  Luo Haoran
    public GraphPoet(File corpus) throws IOException {
```

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。

```
119  /**
120   * Generate a poem.
121   *
122   * @param input string from which to create the poem
123   * @return poem (as described above)
124   */
125   8 个用法  Luo Haoran
    public String poem(String input) {
```

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。

3.1.5.3 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：

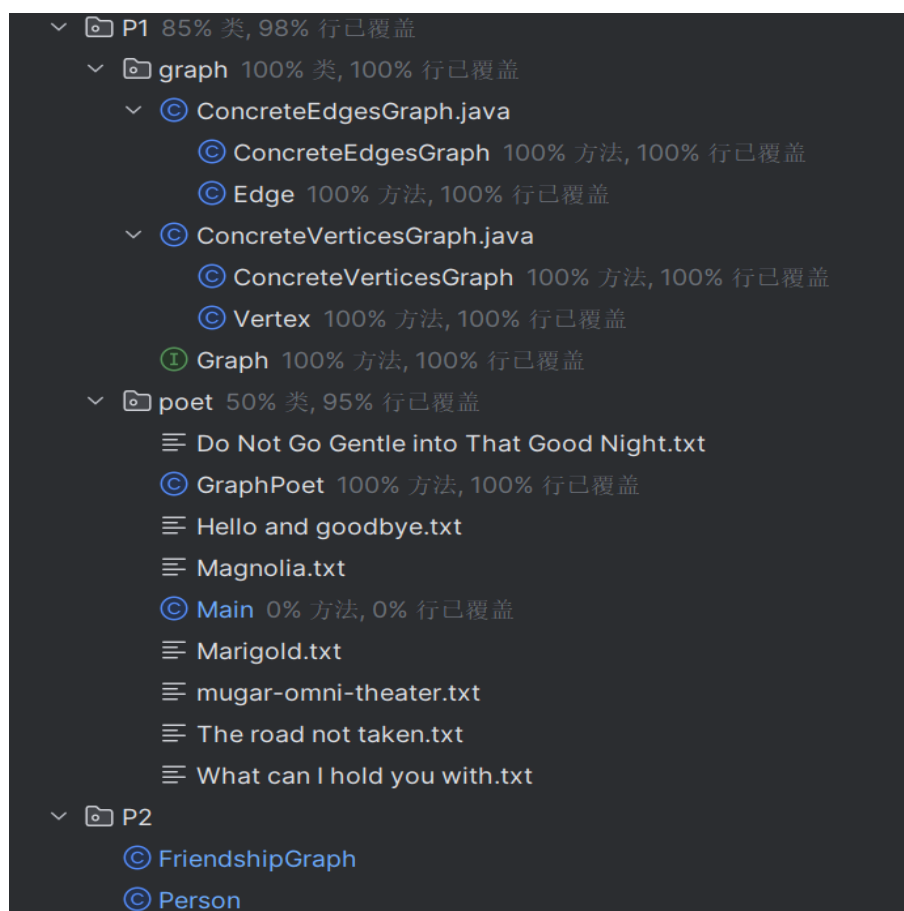
```
Lu Haoan *
public static void main(String[] args) throws IOException {
    final GraphPoet GentleGoodNight = new GraphPoet(new File("src/P1/poet/Do Not Go Gentle into The
    final String input = "Go into a bay, rage with sight.";
    System.out.println(input + "\n>>>\n" + GentleGoodNight.poem(input));
}

F:\JavaDevelopmentKit\jdk-11.0.2\bin\java.exe -javaagent:F:\JavaDevelopmentKit
Go into a bay, rage with sight.
>>>
Go gentle into a green bay, rage rage with blinding sight.
|
进程已结束，退出代码为 0
```

如果能找到一些庞大的语料库，或许这个例子会“更酷”？

3.1.6 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。

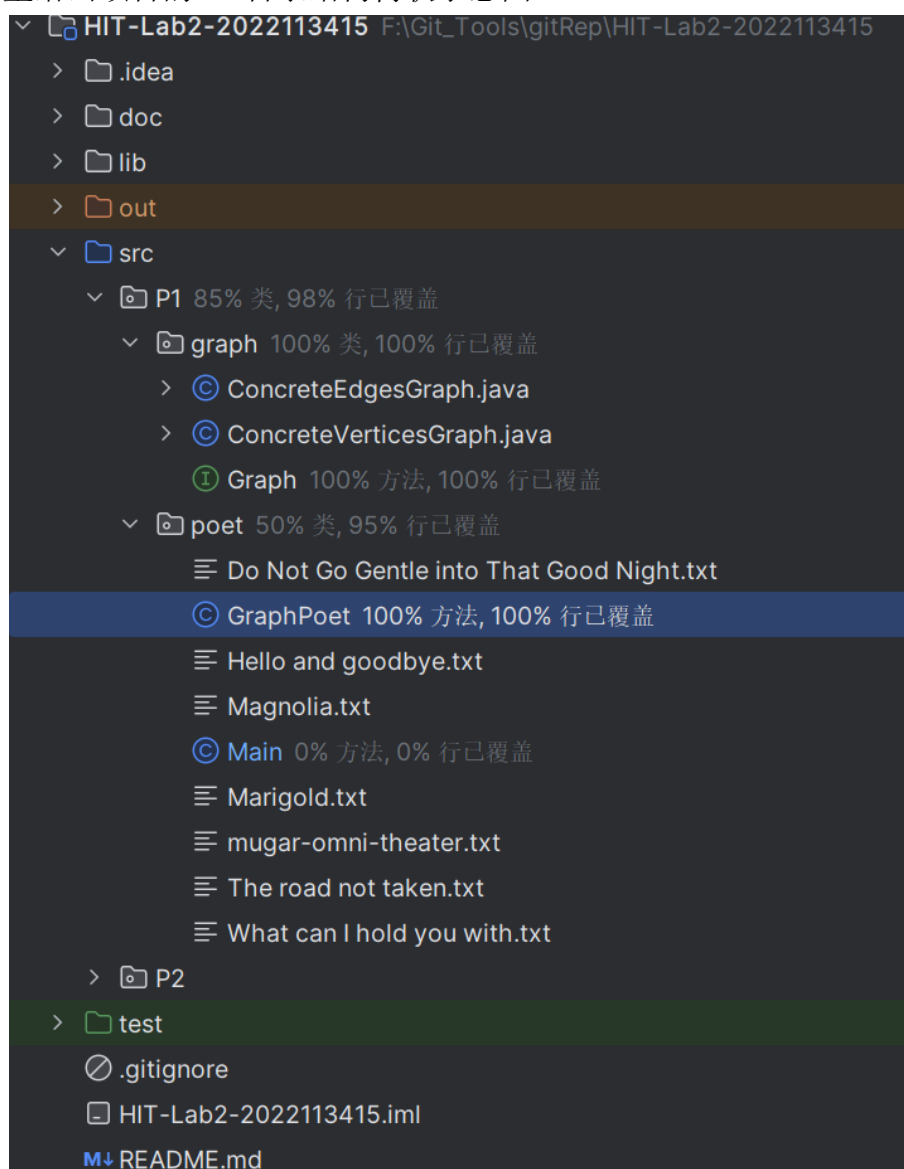
3.1.7 Before you're done

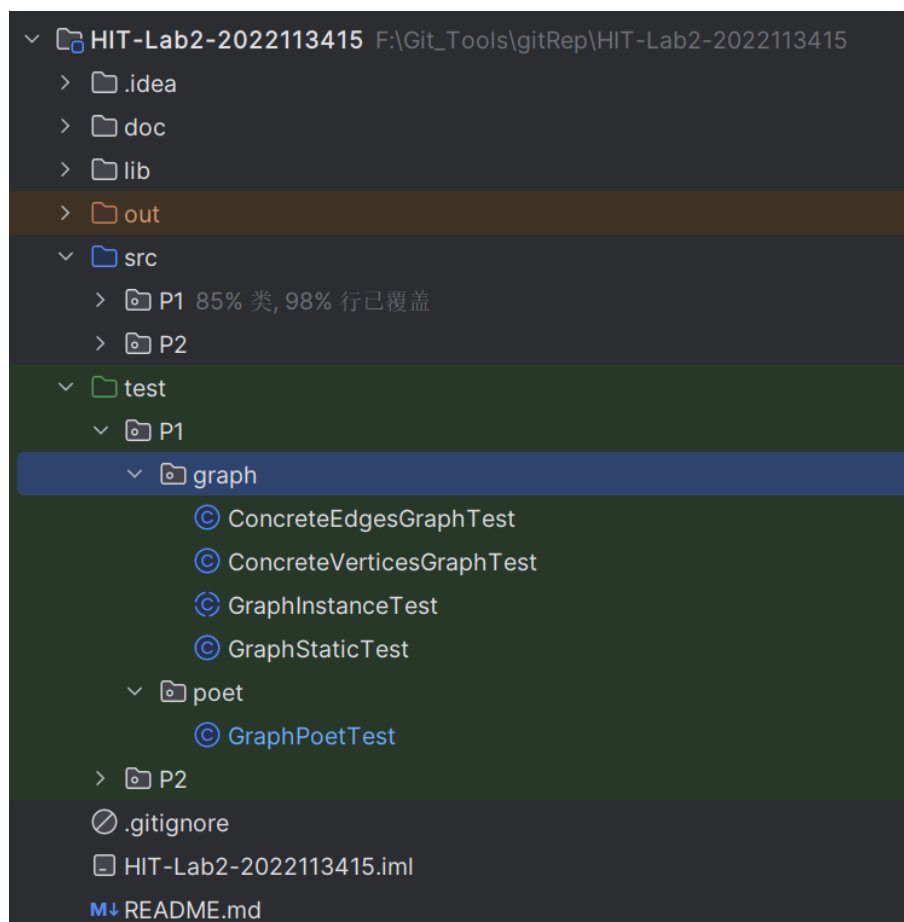
按照 http://web.mit.edu/6.031/www/sp17/psets/ps2/#before_youre_done 的说明，检查程序，在 IDE 界面并未提示项目文件中有任何由代码本身导致的 Warning 报错。

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：





3.2 Re-implement the Social Network in Lab1

Problem2 要求我们基于在前面步骤中定义的 $\text{Graph}\langle L \rangle$ 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。

3.2.1 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：

```

8  ▶ public class FriendshipGraph {
    17 个用法
9      private final Graph<Person> graph = new ConcreteEdgesGraph<>();
10
11  // Abstraction function:
12  //   AF(graph) = 全体人员及人员间关系组成的关系图
13  //
14  // Representation invariant:
15  //   每个顶点Person的关键字（姓名）唯一确定
16  //   边不允许自达
17  //   所有边的权值均为1
18  //
19  // Safety from rep exposure:
20  //   返回的内部变量均为Immutable的数据类型
21  //   类内全部属性采用private final修饰
22

```

根据如上内容，为 FriendshipGraph 类编写 checkRep 方法：

```

26  // checkRep
    2 个用法 新 *
27  public void checkRep(){
28      Set<String> person_names = new HashSet<>();
29      for(Person p: graph.vertices()){
30          person_names.add(p.getName());
31          for(Person p2: graph.targets(p).keySet()){
32              assert graph.targets(p).get(p2) == 1;
33              assert !p2.getName().equals(p.getName());
34          }
35      }
36      assert person_names.size() == graph.vertices().size();
37  }

```

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。

3.2.2 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。

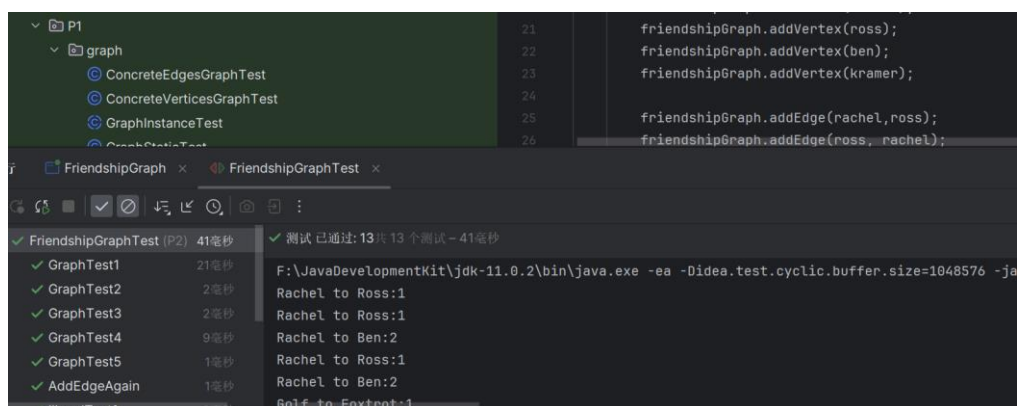
`Person` 类较 Lab1 不同: Lab1 中实验者将 `Person` 在关系图中的邻接信息存储于 `Person` 类内, 这里则放在了 `FriendshipGraph` 类的 `graph` 对象中, 因而需要对 `Person` 类进行修改, 删去原有的 `ChildList` 属性, `generateChildList` 方法等。现在 `Person` 类内仅包含一个存储 `Person` 对象名称的字符串 `name`, `Person` 类的构造方法以及获取 `Person` 对象 `name` 的方法 `getName`。

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:

```
13 ④ public boolean equals(Object object){
14      if(this==object) return true;
15      if(object==null || this.getClass()!=object.getClass()) return false;
16      Person person = (Person) object;
17      return this.name.equals(person.name);
18  }
```

3.2.3 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。

3.2.4 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。

```
haoran@luo MINGW64 /f/Git_Tools/gitRep/HIT-Lab2-2022113415 (master)
$ git add test

haoran@luo MINGW64 /f/Git_Tools/gitRep/HIT-Lab2-2022113415 (master)
$ git commit -m "Test and src Finished, Doing doc"
[master f6ab8cd] Test and src Finished, Doing doc
5 files changed, 38 insertions(+), 8 deletions(-)

haoran@luo MINGW64 /f/Git_Tools/gitRep/HIT-Lab2-2022113415 (master)
$ git push -u origin master
Enumerating objects: 29, done.
Counting objects: 100% (29/29), done.
Delta compression using up to 20 threads
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 927.33 KiB | 9.46 MiB/s, done.
Total 15 (delta 7), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (7/7), completed with 7 local objects.
To https://github.com/ComputerScienceHIT/HIT-Lab2-2022113415.git
785b982..f6ab8cd master -> master
branch 'master' set up to track 'origin/master'.

haoran@luo MINGW64 /f/Git_Tools/gitRep/HIT-Lab2-2022113415 (master)
```

4 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

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
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 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。

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。

6.2 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。

(2) 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。

(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？

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。

(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?

提高代码的使用效率, 减少重复编写, 节省程序员的精力。

(5) 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。

(6) 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 :-)

(7) 《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。