2024 年春季学期软件构造

Lab1 实验报告

班号	2211104	学号	2022211917	姓名	周雨凡
开始	2024.5.25	截止	2024.6.7 第十四周周五 17:00		

目录

1. 实验目标概述

本次实验旨在通过抽象数据类型（Abstract Data Type, ADT）的设计和面向对象编程（Object-Oriented Programming, OOP）的运用，深入理解并实践软件工程中的核心概念。实验要求参与者：

1. 分析给定的应用场景，识别并定义所需的 ADT，即明确数据结构及其操作的逻辑视图。
2. 设计 ADT 的规约，包括前置条件（pre-condition）和后置条件（post-condition），确保函数调用的正确性和行为的一致性。
3. 对 ADT 进行泛型化处理，使其能够处理多种数据类型，增强其适用性和复用性。
4. 根据规约设计 ADT 的不同实现方式，包括确定数据结构的具体表示（representation）、表示不变性（representation invariant）和抽象函数（abstraction function），以确保 ADT 的内部状态在任何情况下都保持正确且一致。
5. 使用 OOP 原则实现 ADT，同时检查实现中是否违背了表示不变性以及是否存在表示泄露的问题，以维护封装性和数据安全性。
6. 利用 ADT 及其具体实现，开发解决特定应用问题的程序。

实验过程中，学生将学习如何在复杂环境中设置和配置实验环境，遵循 MIT 的指导文档完成编程任务，获取和使用初始代码，以及进行必要的测试以验证程序的正确性。此外，实验还要求学生撰写实验报告，详细记录实验过程中的挑战、解决方案以及所完成的开发任务，确保报告的结构清晰、内容精炼、格式规范。最终，学生需在规定时间内提交电子版实验报告，遵循特定的文件命名和存放规则。

2. 实验环境配置

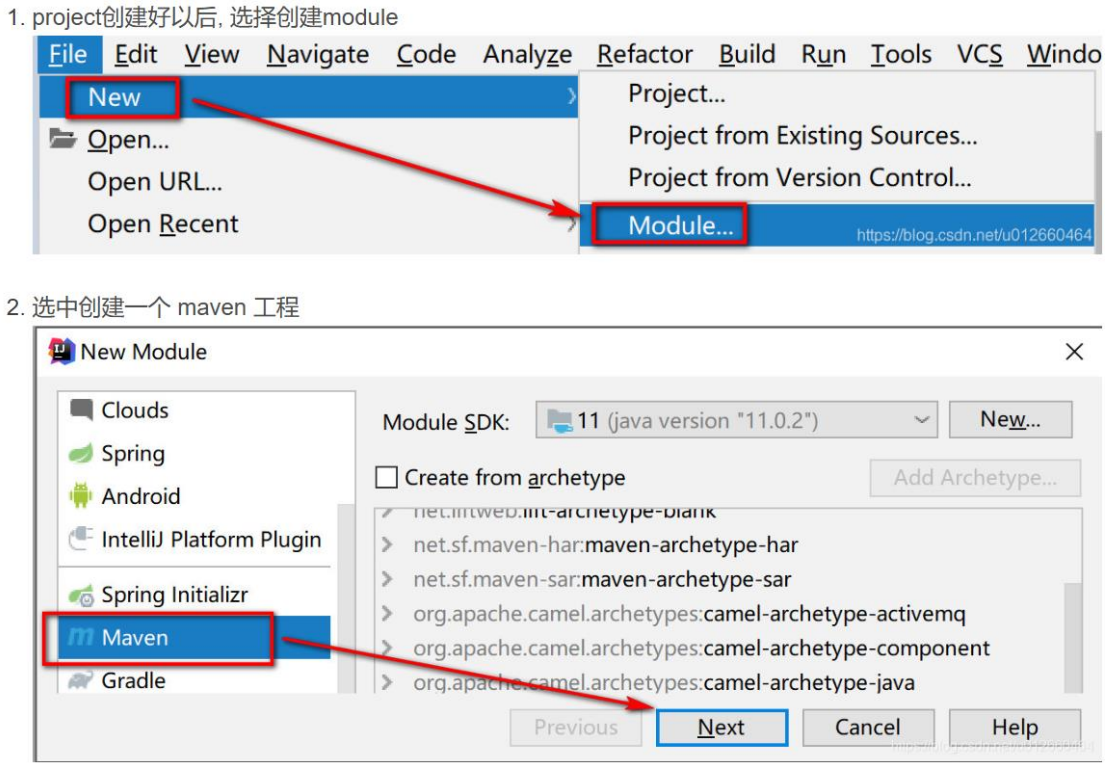

图 2-1 使用 IDEA 创建 Maven 示意

在创建项目的时候，本打算直接通过配置编写 pom.xml 文件的方式导入 Maven 依赖从而将项目转为 Maven 工程，但是在转化的过程中网络下载过慢导致超时，而且源文件是使用 Eclipse 创建的，里面存在诸如.classpath 文件等，不与 IDEA 兼容，所以不得不重新创建一个新项目，然后选择 Maven 来进行创建，将原来的文件导入。

同时使用 git init 命令初始化 git 仓库，通过配置.gitignore 文件来管理提交的文件。

更改后的目录树型结构如下（使用 tree 命令打印）：

```
├─src
│  ├─main
│  │  ├─java
│  │  │  ├─graph
│  │  │  └─poet
│  │  └─resources
│  └─test
│      └─java
│          ├─graph
│          └─poet
```

3. 实验过程（Poetic Walks）

3.1 Problem 1: Test Graph <String>

任务首先对抽象类 GraphInstanceTest 设计测试函数。

下面部分是针对 Graph<String>设计测试策略，编写测试用例利用等价类划分的思想进行测试，测试策略以及每个测试函数的设计思路如下：

1. 添加顶点 add()方法

testAddVertex()：测试成功添加单个顶点到图中。
testAddDuplicateVertex()：测试尝试添加重复顶点是否会失败。
testAddMultipleVertices()：测试成功添加多个顶点到图中。
testAddNullVertex()：测试尝试添加空顶点是否会抛出异常。
testAddEmptyStringVertex()：测试是否可以成功添加空字符串作为顶点。

2. 添加边 set()方法

testAddNewEdge()：测试成功添加新边到图中。
testUpdateEdgeWeight()：测试成功更新现有边的权重。
testRemoveEdge()：测试成功移除现有边。
testSelfLoopEdge()：测试成功添加自环边。
testAddMultipleEdges()：测试成功添加多条边到图中。

3. 移除顶点 remove()方法

testRemoveVertexExists()：测试成功移除存在的顶点。
testRemoveVertexNotExists()：测试尝试移除不存在的顶点是否会失败。
testRemoveVertexWithEdges()：测试成功移除带有边的顶点。
testRemoveVertexWithSelfLoop()：测试成功移除带有自环的顶点。
testRemoveVertexFromEmptyGraph()：测试从空图中移除顶点是否会失败。

4. 获取顶点 vertices()方法

testVerticesAfterAddVertex()：测试添加单个顶点后的顶点获取。
testVerticesAfterAddMultipleVertices()：测试添加多个顶点后的顶点获取。
testVerticesAfterRemoveVertex()：测试移除顶点后的顶点获取。
testVerticesAfterAddAndRemoveVertices()：测试添加和移除顶点后的顶点获取。
testVerticesFromEmptyGraph()：测试从空图获取顶点。

5. 获取源顶点 sources()方法

testSourcesEmptyGraph()：测试从空图获取源顶点。
testSourcesNoIncomingEdges()：测试无入边顶点的源顶点获取。
testSourcesOneIncomingEdge()：测试有一条入边顶点的源顶点获取。
testSourcesMultipleIncomingEdges()：测试有多条入边顶点的源顶点获取。
testSourcesAfterRemovingEdge()：测试移除边后的源顶点获取。
testSourcesSelfLoop()：测试有自环边顶点的源顶点获取。

6. 获取目标顶点 targets()方法

testTargetsEmptyGraph()：测试从空图获取目标顶点。

testTargetsNoOutgoingEdges()：测试无出边顶点的目标顶点获取。

testTargetsOneOutgoingEdge()：测试有一条出边顶点的目标顶点获取。

testTargetsMultipleOutgoingEdges()：测试有多条出边顶点的目标顶点获取。

testTargetsAfterRemovingEdge()：测试移除边后的目标顶点获取。

testTargetsSelfLoop()：测试有自环边顶点的目标顶点获取。

3. 2 Problem 2: Implement Graph <String>

3.2.1 Implement ConcreteEdgesGraph

下面部分是对 ConcreteEdgesGraph 类的实现。

定义抽象与具体表示：

1. 抽象函数（AF）：描述了顶点集合 vertices 和边列表 edges 如何映射到一个有向图。顶点集合表示图中的所有顶点，边列表表示图中的所有边，每条边都有一个源顶点、目标顶点和权重。
2. 表示不变性（RI）：定义了数据结构在任何时候必须满足的约束条件，确保数据结构的一致性和正确性。边必须有起始节点和目标节点，权重必须是大于 0 的实数，每个顶点必须存在于 vertices 集合中，每两点之间最多只能有一条边。
3. 防止表示暴露：所有字段声明为私有和最终，确保它们不能被外部代码直接访问或修改。返回顶点和边的不可修改视图，防止调用者修改这些集合。在有必要时，使用防御性拷贝技术，保护内部数据结构不被外部修改。
4. 类的设计：ConcreteEdgesGraph 类实现了 Graph<String> 接口，表示一个有向图。Edge 类表示有向图中的边，包含源顶点、目标顶点和权重。

实现思路：顶点使用 HashSet 存储，边使用 ArrayList 存储。提供添加顶点、设置边、移除顶点、获取顶点集合、获取源顶点和目标顶点的映射等方法。使用 checkRep 方法检查表示不变性，确保数据结构的正确性。

实现过程：

顶点和边的存储：vertices 使用 HashSet 存储顶点，确保顶点的唯一性。edges 使用 ArrayList 存储边，允许快速添加和遍历边。

关键代码：

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

添加顶点：add 方法检查顶点是否为 null，如果是，抛出 NullPointerException。如果顶点已经存在于集合中，返回 false。否则，添加顶点到集合中，并调用 checkRep 方法检查表示不变性。

关键代码：

```
if (vertex == null) throw new NullPointerException("顶点不能为 null");
if (vertices.contains(vertex)) {
    return false; // 顶点已存在
```

```
}
vertices.add(vertex);
```

设置边：set 方法检查源顶点和目标顶点是否为 null，如果是，抛出 NullPointerException。如果源顶点或目标顶点不存在于顶点集合中，添加它们。查找现有的边，如果找到，移除它并根据新权重决定是否添加新边。如果没有找到现有的边，且新权重大于 0，添加新边。

关键代码：

```
for (Edge e : edges) {
    if (e.getSource().equals(source) && e.getTarget().equals(target)) {
        int oldWeight = e.getWeight();
        edges.remove(e); // 移除现有的边
        if (weight > 0) {
            edges.add(new Edge(source, target, weight)); // 添加新边
        }
        checkRep();
        return oldWeight; // 返回旧权重
    }
}
```

移除顶点：remove 方法检查顶点是否存在于集合中，如果不存在，返回 false。移除与顶点相关的所有边，并从顶点集合中移除顶点。调用 checkRep 方法检查表示不变性。

关键代码：

```
// 移除与顶点相关的边
edges.removeIf(edge -> edge.getSource().equals(vertex) ||
edge.getTarget().equals(vertex));
// 移除顶点本身
vertices.remove(vertex);
```

获取顶点和边：vertices 方法返回顶点集合的不可修改视图。sources 方法返回所有指向指定目标顶点的边的源顶点及其权重的映射。targets 方法返回所有从指定源顶点出发的边的目标顶点及其权重的映射。

关键代码：

```
return Collections.unmodifiableSet(vertices);
```

检查表示不变性：checkRep 方法检查每条边的源顶点和目标顶点是否在顶点集合中，边的权重是否大于 0，每两点之间是否有多于一条边。

关键代码：

```
// 检查源和目标顶点是否在顶点集合中
assert vertices.contains(source) : "源顶点不在顶点集合中";
assert vertices.contains(target) : "目标顶点不在顶点集合中";
// 检查边的权重是否大于 0
assert weight > 0 : "边的权重必须是大于 0 的实数";
```

完整实现：ConcreteEdgesGraph 类实现了有向图的基本功能，支持添加、移除顶点，设置边及获取顶点和边的信息。Edge 类表示图中的边，确保源顶点、目标顶点不为 null，边的权重为非负数。

ConcreteEdgesGraph 的测试策略：

testToStringEmptyGraph:创建一个空的 ConcreteEdgesGraph 实例。调用 toString() 方法并验证返回的字符串表示是否符合预期（应表示为空图）。

testToStringOnlyVertices:创建一个 ConcreteEdgesGraph 实例。添加多个顶点但不添加任何边。调用 toString() 方法并验证返回的字符串表示是否只包含顶点信息且没有边信息。

testToStringWithEdges:创建一个 ConcreteEdgesGraph 实例。添加多个顶点并添加一些边。调用 toString() 方法并验证返回的字符串表示是否包含正确的顶点和边信息。

testToStringMultipleEdgesBetweenVertices:创建一个 ConcreteEdgesGraph 实例。添加多个顶点和多条边（包括相同顶点之间的多条边）。调用 toString() 方法并验证返回的字符串表示是否正确反映多个顶点之间的多条边。

Edge 的测试策略：

构造函数（有效边）:调用 Edge 的构造函数创建一个具有非空源顶点、非空目标顶点和正权重的边实例。验证边实例的属性是否正确初始化且无异常抛出。

构造函数（源顶点为 null）:调用 Edge 的构造函数，传入 null 作为源顶点。捕获异常并验证抛出的是 AssertionError。

构造函数（目标顶点为 null）:调用 Edge 的构造函数，传入 null 作为目标顶点。捕获异常并验证抛出的是 AssertionError。

构造函数（权重为负数）:调用 Edge 的构造函数，传入负数作为边的权重。捕获异常并验证抛出的是 AssertionError。

测试结果：

图 3.2.1-1 测试结果

详细的测试导出文件在 Test Results - ConcreteEdgesGraphTest.html 文件中。

测试覆盖率：

表 3.2.1-1 测试覆盖率

Class	Class(%)	Method(%)	Branch(%)	Line(%)
ConcreteEdgesGraph	100% (1/1)	100% (9/9)	79.3% (46/58)	100% (65/65)
Edge	100% (1/1)	85.7% (6/7)	75% (9/12)	92.3% (12/13)

详细的测试覆盖率文档保存在 htmlReport/index.html 中。

3.2.2 Implement ConcreteVerticesGraph

定义抽象与具体表示：

1. 抽象函数（AF）：对于 ConcreteVerticesGraph 类，AF 将顶点列表(vertices)映射到一个有向图，其中每个顶点包含它的出边。具体来说，顶点列表中的每个 Vertex 对象表示图中的一个顶点，Vertex 对象中的'name'字段表示该顶点的名称，'edges'字段是一个映射，将目标顶点的名称映射到边的权重。对于 Vertex 类，AF 将 Vertex 对象表示为一个顶点及其指向的有向边的集合。Vertex 对象中的'name'字段表示顶点的名称，'edges'字段是一个映射，将目标顶点的名称映射到边的权重。
2. 表示不变性（RI）：ConcreteVerticesGraph 类的 RI 包括以下几点：顶点列表中的每个顶点名称应该是唯一的。顶点的名称不能为 null。每个顶点的边的目标顶点应该存在于顶点列表中。边的权重应该是大于 0 的实数。Vertex 类的 RI 包括以下几点：顶点名称不能为 null。边的权重必须是非负数。
3. 防止表示暴露：ConcreteVerticesGraph 类和 Vertex 类都使用了私有字段和不可变类型，并通过返回不可修改的集合来防止表示暴露。此外，在需要时使用了防御性拷贝，确保外部无法直接修改内部数据结构，从而保证了数据的安全性。
4. 类的设计：ConcreteVerticesGraph 类是实现 Graph 接口的具体类，用于表示有向图。它使用一个列表来存储顶点，并提供了操作来添加、设置、删除顶点以及获取顶点的源和目标。Vertex 类是 ConcreteVerticesGraph 类的内部类，表示图中的顶点。它包含顶点的名称和指向其他顶点的有向边的集合，提供了操作来设置、移除边以及获取顶点的名称和边集合。

实现思路：ConcreteVerticesGraph 类使用一个列表来存储顶点，对外提供了添加、设置、删除顶点以及获取顶点的源和目标的操作。它通过维护表示不变性来确保内部数据结构的一致性。Vertex 类表示图中的顶点，它包含顶点的名称和指向其他顶点的有向边的集合，通过维护表示不变性来确保顶点对象的一致性。

实现过程：

ConcreteVerticesGraph 类：

checkRep()：检查表示不变性，确保图中的顶点名称不为空且唯一，并且每条边的权重为正数。通过遍历顶点列表进行检查。

关键代码：

```
Map<String, Integer> edges = vertex.getEdges();
for (Map.Entry<String, Integer> entry : edges.entrySet()) {
    assert entry.getValue() > 0 : "边的权重必须是正数";
}
```

add(L vertex)：向图中添加一个新顶点。如果顶点已存在，返回 false；否则，添加新顶点并返回 true。在添加新顶点后，调用 checkRep() 以确保表示不变性。

关键代码：

```
for (Vertex v : vertices) {
    if (v.getName().equals(vertex)) {
        return false; // 顶点已存在
    }
```

```
}
vertices.add(new Vertex(vertex));
```

set(L source, L target, int weight)：设置或更新源顶点到目标顶点的边的权重。如果权重为 0，则移除边。如果边存在则更新其权重，否则添加新边。返回旧的权重值，并检查表示不变性。

关键代码：

```
if (sourceVertex != null) {
    if (weight == 0) {
        oldWeight = sourceVertex.removeEdge(target);
    } else {
        oldWeight = sourceVertex.setEdge(target, weight);
    }
} else {
    if (weight != 0) {
        sourceVertex = new Vertex(source);
        sourceVertex.setEdge(target, weight);
        vertices.add(sourceVertex);
    }
}
```

findVertex(L name)：在顶点列表中查找指定名称的顶点。如果找到，返回该顶点，否则返回 null。该方法用于辅助查找顶点。

关键代码：

```
for (Vertex vertex : vertices) {
    if (vertex.getName().equals(name)) {
        return vertex;
    }
}
```

remove(L vertex)：从图中移除指定顶点以及所有连接该顶点的边。如果顶点不存在，返回 false；否则移除顶点并返回 true。在移除顶点后，调用 checkRep() 以确保表示不变性。

关键代码：

```
vertices.remove(vertexToRemove);
for (Vertex v : vertices) {
    v.removeEdge(vertex);
}
```

vertices()：返回图中所有顶点名称的集合。该集合是不可修改的，以确保外部无法改变图的内部状态。

关键代码：

```
Set<String> vertexNames = new HashSet<>();
for (Vertex v : vertices) {
    vertexNames.add(v.getName());
}
return Collections.unmodifiableSet(vertexNames);
```

sources(L target)：返回一个映射，表示所有指向目标顶点的源顶点及其边的权重。处理自环边的情况，并返回一个不可修改的映射。

关键代码：

```
if (v.getName().equals(target) && v.getEdges().containsKey(target)) {
    // 将目标顶点及其权重添加到源顶点映射中
    sourceMap.put(target, v.getEdges().get(target));
    // 标记存在自环边
    hasSelfLoop = true;
}
```

targets(L source)：返回一个映射，表示源顶点指向的所有目标顶点及其边的权重。该映射是不可修改的。

关键代码：

```
for (Vertex v : vertices) {
    if (v.getName().equals(source)) {
        return Collections.unmodifiableMap(v.getEdges());
    }
}
return Collections.unmodifiableMap(new HashMap<>());
```

Vertex 类：

checkRep()：检查表示不变性，确保顶点名称不为空，边的权重为非负数。通过遍历边列表进行检查。

关键代码：

```
assert name != null : "顶点名称不能为 null";
for (int weight : edges.values()) {
    assert weight >= 0 : "边的权重必须是非负数";
}
```

getEdges()：返回顶点的边集合（不可变视图）。确保外部无法修改顶点的边集合。

关键代码：

```
return Collections.unmodifiableMap(edges);
```

setEdge(L target, int weight)：设置或更新指向目标顶点的边的权重。如果边存在则更新其权重，否则添加新边。返回旧的权重值。

关键代码：

```
Integer oldWeight = edges.put(target, weight);
return oldWeight != null ? oldWeight : 0;
```

removeEdge(L target)：移除与目标顶点的边。返回旧的权重值，如果边不存在则返回 0。

关键代码：

```
Integer oldWeight = edges.remove(target);
return oldWeight != null ? oldWeight : 0;
```

通过以上实现，ConcreteVerticesGraph 类和 Vertex 类共同构成了一个有向图的表示，并提供了相应的操作来添加、设置、删除顶点以及获取顶点的源和目标。同时，通过维护表示不变性来确保数据结构的一致性和正确性。

完整实现：ConcreteVerticesGraph 类实现了 Graph 接口，使用列表存储顶点，并提供了操作来管理顶点和它们之间的边。Vertex 类表示顶点，包含名称和边的映射。通过维护表示不变性和封装内部数据结构，确保数据一致性和安全性。

ConcreteVerticesGraph 的测试策略：

testToStringEmptyGraph: 创建一个空的 ConcreteVerticesGraph 实例。调用 toString() 方法并验证返回的字符串表示是否符合预期（应表示为空图）。

testToStringOnlyVertices:创建一个 ConcreteVerticesGraph 实例。添加多个顶点但不添加任何边。调用 toString() 方法并验证返回的字符串表示是否只包含顶点信息且没有边信息。

testToStringWithEdges:创建一个 ConcreteVerticesGraph 实例。添加多个顶点并添加一些边。调用 toString() 方法并验证返回的字符串表示是否包含正确的顶点和边信息。

testToStringMultipleEdgesBetweenVertices:创建一个 ConcreteVerticesGraph 实例。添加多个顶点和多条边（包括相同顶点之间的多条边）。调用 toString() 方法并验证返回的字符串表示是否正确反映多个顶点之间的多条边。

Vertex 的测试策略：

testVertexCreation:创建一个 Vertex 实例并验证其名称是否正确。验证该顶点的边集合是否为空。

testSetEdge:创建一个 Vertex 实例并添加一条新的边，验证返回的旧权重是否为 0。验证边集合的大小是否正确。更新同一边的权重，验证返回的旧权重是否正确。验证边集合的大小是否未改变。

testRemoveEdge:创建一个 Vertex 实例并添加一条新的边。移除该边，验证返回的旧权重是否正确。尝试移除不存在的边，验证返回的权重是否为 0。

测试结果：

图 3.2.2-1 测试结果

详细的测试导出文件在根目录下的 Test Results -ConcreteVerticesGraphTest.html 文件中。

测试覆盖率：

表 3.2.2-1 测试覆盖率

Class	Class(%)	Method(%)	Branch(%)	Line(%)
ConcreteVerticesGraph	100% (1/1)	100% (10/10)	87.9% (58/66)	98.7% (77/78)
Vertex	100% (1/1)	87.5% (7/8)	50% (7/14)	86.7% (13/15)

详细的测试覆盖率文档保存在 htmlReport/index.html 中。

3.3 Problem 3: Implement generic Graph<L>

3.3.1 Make the implementations generic

将问题 2 中编写的 String 类型都改成 L 泛型：

对于 ConcreteEdgesGraph 类的修改：

```
public class ConcreteEdgesGraph<L> implements Graph<L> {
    private final Set<L> vertices = new HashSet<>();
    private final List<Edge<L>> edges = new ArrayList<>();
```

对于 Edge 类的修改：

```
class Edge<L> {
    private final L source;
    private final L target;
    private final int weight;
```

对于 ConcreteVerticesGraph 的修改：

```
public class ConcreteVerticesGraph<L> implements Graph<L> {
    private final List<Vertex<L>> vertices = new ArrayList<>();
```

对于 Vertex 的修改：

```
class Vertex<L> {
    private final L name;
    private final Map<L, Integer> edges = new HashMap<>();
```

在函数和属性定义的时候更改完毕，对于函数内，以及原本便是 String 所以在 toString 函数中没有使用 toString 进行类型转换的，都要进行修改。

这里仅展示 sources 与 checkRep 的修改：

```
//source 部分源码
@Override
public Map<L, Integer> sources(L target) {
    Map<L, Integer> sourceMap = new HashMap<>();
    for (Edge<L> edge : edges) {
        if (edge.getTarget().equals(target)) {
            sourceMap.put(edge.getSource(), edge.getWeight());
        }
    }
    return Collections.unmodifiableMap(sourceMap);
}
//checkRep 部分源码
for (Edge<L> edge : edges) {
    L source = edge.getSource();
    L target = edge.getTarget();
    double weight = edge.getWeight();
```

3.3.2 Implement Graph.empty()

以下是 Graph.empty()函数的实现：

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

调用已经实现的 ConcreteEdgesGraph 的默认构造函数实现即可，同时使用 inline 样式，直接返回新建的对象。

隐藏具体实现：这个方法返回了一个 ConcreteEdgesGraph<L> 的实例，而不是让客户端直接使用 ConcreteEdgesGraph 的构造函数。这意味着客户端代码只需要知道 Graph 接口，而不需要知道具体实现细节。这符合题目中“clients of Graph should not be aware of how we've implemented it”的要求。

泛型支持：这个方法是泛型方法，能够支持任意类型的顶点标签。这与题目中“the implementation of Graph does not know or care about the actual type of the vertex labels”的要求一致。

类型安全：使用泛型确保了类型安全性，避免了类型转换问题。这个实现不会产生编译器警告，也没有使用 @SuppressWarnings 注解来屏蔽警告。

对于 GraphStaticTest 的编写：

测试 Graph.empty() 方法：

testAssertionsEnabled:确保断言已启用。此测试通过 VM 参数 -ea 启用断言，并通过简单的 assert false 检查断言是否生效。

testEmptyVerticesEmpty: 创建一个空图并验证其顶点集合是否为空。使用 Graph.empty() 方法创建图实例。通过调用 vertices() 方法来检查图的顶点集合是否为空。

测试不同标签类型的空图：

testEmptyGraphWithStringLabels:创建一个标签为 String 类型的空图。检查空图实例是否不为 null。检查图的顶点集合是否为空。

testEmptyGraphWithIntegerLabels:创建一个标签为 Integer 类型的空图。检查空图实例是否不为 null。检查图的顶点集合是否为空。

testEmptyGraphWithCustomLabels:定义一个不可变的自定义标签类 Label，并创建一个标签为 Label 类型的空图。确保自定义标签类具有合理的 equals、hashCode 和 toString 方法实现。创建空图实例并检查是否不为 null。检查图的顶点集合是否为空。

测试结果：

图 3.3.2-1 测试结果

详细的测试导出文件在 Test Results - GraphStaticTest.html 文件中。

测试覆盖率：

表 3.3.2-1 测试覆盖率

Class	Class(%)	Method(%)	Line(%)
Graph	100% (1/1)	100% (1/1)	100% (1/1)

详细的测试覆盖率文档保存在 htmlReport/index.html 中。

3.4 Problem 4: Poetic walks

3.4.1 Test GraphPoet

测试策略：

对 GraphPoet(File corpus) 方法：

1. 使用具有已知结构的小型语料库进行测试，检查是否正确构建了诗人的关联图。
2. 使用空语料库进行测试，检查是否能正确处理空语料库。

对 poem(String input) 方法：

1. 使用不需要桥接词的输入进行测试，检查是否能正确生成诗歌。
2. 使用需要一个桥接词的输入进行测试，检查是否正确生成带有一个桥接词的诗歌。
3. 使用需要多个桥接词的输入进行测试，检查是否能正确生成带有多个桥接词的诗歌。
4. 使用没有可能的桥接词的输入进行测试，检查是否能正确处理没有可能的桥接词的情况。

对 toString() 方法：

1. 测试空语料库，检查是否返回了空的字符串。
2. 测试非空语料库，检查是否返回了诗人的关联图的字符串表示。

对 testAssertionsEnabled() 方法：

1. 测试断言是否启用，确保断言已启用。

3.4.2 Implement GraphPoet

抽象与具体表示：

1. 抽象函数（AF）：AF(graph) = 一个基于词语关联图的诗歌生成器，其中 graph 表示词语之间的关联关系。
2. 表示不变性（RI）：
   a) 在构造方法中，使用 map 记录词语之间的关联关系和权重，并在 checkRep 方法中检查边的权重是否大于 0。
   b) 在 poem 方法中，根据输入的字符串和词语关联图生成诗歌。
3. 防止表示暴露：将词语关联图 graph 设置为私有 final 属性，防止外部直接访问。
4. GraphPoet 类设计：
   a) 词语关联图：使用图的数据结构表示词语之间的关联关系。
   b) 构造方法：从给定的语料库文本文件中构建词语关联图。
   c) checkRep 方法：用于检查词语关联图是否符合表示不变式。
   d) poem 方法：根据输入的字符串生成诗歌。
   e) toString 方法：返回词语关联图的字符串表示形式。

实现思路：

构造方法：读取语料库文本文件，将词语关系构建成图。

关键代码：

```
// 读取语料库文件并处理
while ((string = in.readLine()) != null) {
    list.addAll(Arrays.asList(string.split(" ")));
}
in.close();
// 构建关联图
for (int i = 0; i < list.size() - 1; i++) {
    String source = list.get(i).toLowerCase();
    String target = list.get(i + 1).toLowerCase();
    int preweight = 0;
    if (map.containsKey(source + target)) {
        preweight = map.get(source + target);
    }
    map.put(source + target, preweight + 1);
    graph.set(source, target, preweight + 1);
}
checkRep();
```

poem 方法：对输入的字符串进行处理，尝试在每对相邻的词语之间插入桥接词，生成诗歌。

关键代码：

```
// 读取语料库文件并处理
while ((string = in.readLine()) != null) {
    list.addAll(Arrays.asList(string.split(" ")));
}
in.close();
// 构建关联图
for (int i = 0; i < list.size() - 1; i++) {
    String source = list.get(i).toLowerCase();
    String target = list.get(i + 1).toLowerCase();
    int preweight = 0;
    if (map.containsKey(source + target)) {
        preweight = map.get(source + target);
    }
    map.put(source + target, preweight + 1);
    graph.set(source, target, preweight + 1);
}
checkRep();
```

toString 方法：返回词语关联图的字符串表示形式。

checkRep 方法：检查词语关联图是否符合表示不变式。

关键代码：

```
for (String source : graph.vertices()) {
    for (String target : graph.targets(source).keySet()) {
```

```
            assert graph.targets(source).get(target) > 0 : "边的权重必须大于 0";
        }
}
```

实现过程：使用 BufferedReader 逐行读取语料库文本文件。将读取的词语列表构建成图，记录词语之间的关联关系和权重。对输入的字符串进行处理，尝试在相邻的词语之间插入桥接词，生成诗歌。使用 assert 关键字在 checkRep 方法中检查词语关联图是否符合表示不变式。

完整实现：GraphPoet 类使用图的数据结构表示词语之间的关联关系，通过构建词语关联图并根据输入的字符串生成诗歌。该类包含构造方法、poem 方法、toString 方法和 checkRep 方法。

测试结果：

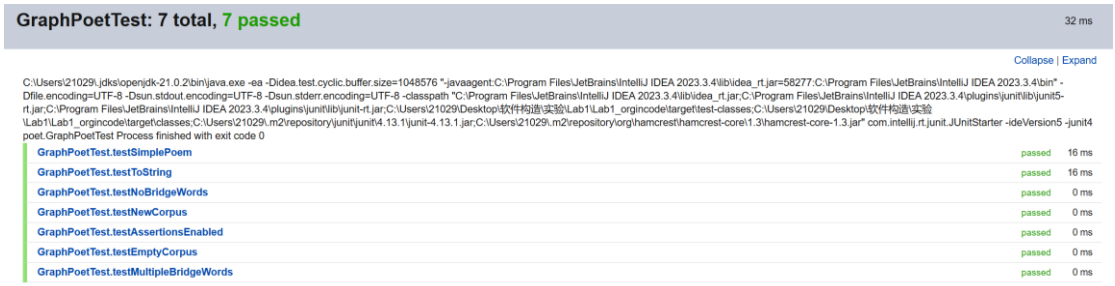

图 3.4.2-1 测试结果

详细的测试导出文件在 Test Results - GraphPoetTest.html 文件中。

测试覆盖率：

表 3.4.2-1 测试覆盖率

Class	Class(%)	Method(%)	Branch(%)	Line(%)
GraphPoet	100% (1/1)	100% (5/5)	83.3% (20/24)	97.5% (39/40)

详细的测试覆盖率文档保存在 htmlReport/index.html 中。

3.4.3 Graph poetry slam

我使用的文本语料库是我自己在 Stray Birds, Rabindranath Tagore, 1916 (ibiblio.org) 网站搜集并经过代码处理的语料数据。

这是我使用的辅助数据处理函数（去除符号）的关键代码：

```
// 逐行读取文件内容
while ((line = reader.readLine()) != null) {
    // 去掉每行中的英文句号，问号和引号
    String newLine = line.replace(".", "");
    newLine = newLine.replace("\"", "");
    newLine = newLine.replace("?", "");
    // 将处理后的内容写入输出文件
    // 如果去除符号后不为空行，则写入输出文件
```

```
    if (!newLine.trim().isEmpty()) {
        writer.write(newLine);
        writer.newLine();
    }
}
```

对于本问题：

原始输入：
"It is the tears of the earth that keep her smiles in bloom."

这句话表达了一个隐喻，意思是地球的眼泪（可能指雨水或其他形式的水）让她（地球）的微笑（花朵）盛开。这是一个富有诗意的表达，描绘了自然界的相互依存。

修改后的输出：
" It is like the tears of love the great earth that keep her smiles in bloom."（划线为新添单词）

这句话在原句的基础上做了一些修改：

1. 添加了"like"：这使得句子变成了一个明喻（simile），即将地球的眼泪比作“爱的眼泪”。
2. 插入了"of love"：这个短语增加了情感层面，将自然现象与爱的情感联系起来。
3. 插入了"the great"：这个短语用来形容地球，使其显得更为宏伟和重要。

这次文本变换通过增加比喻和形容词使原句变得更富有情感和诗意。原始句子描述了一种自然现象，而修改后的句子通过将这种现象与“爱”联系起来，使其表达的内容更加丰富和深刻。

完整的输出在 src/main/java/poet/GraphStructure.txt 中（过多不宜展示）。