CSED332 Assignment 3

Due Tuesday, October 8

Objectives

- Write formal specifications and black-box test cases for abstract interfaces
- Understand the Liskov substitution principle (a.k.a., behavioral subtyping)
- Learn Java programming language (Generics)

Background: Graphs and Trees

- A *directed graph* is a pair $G = (V, E)$, where V is a set of *vertices* (also called nodes) and $E \subseteq V \times V$ is a set of edges that connects two vertices. For example, Fig. 1 shows the graph:
$$V = \{1, 2, 3, 4, 5, 6\}, \quad E = \{(1, 1), (1, 2), (3, 6), (4, 1), (4, 2), (6, 3)\})$$
- A (rooted) *tree* is a directed graph $G = (V, E)$ such that one vertex is designated as the root and there exists exactly one path from the root to any vertex. For example, Fig. 2 show the tree:
$$V = \{1, 2, 3, 4, 5, 6\}, \quad E = \{(1, 2), (1, 4), (2, 3), (2, 6), (4, 5)\}), \quad \text{where 1 is the root}$$

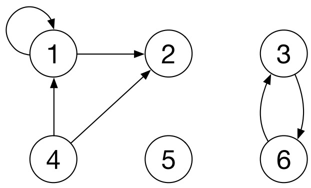

Figure 1: A graph

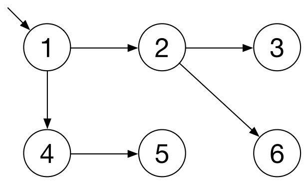

Figure 2: A tree

Problem 1: Abstract Interface Specifications for Graphs and Trees

- In this assignment, we consider generic abstract interfaces for graphs and trees, where vertices are represented as any elements of a given (immutable and comparable) type N:
  - Graph<N>: an interface for direct graphs
  - Tree<N>: an interface for rooted trees, extending Graph<N>
  - MutableGraph<N>: an interface with mutable operations, extending Graph<N>
  - MutableTree<N>: an interface with mutable operations, extending Tree<N>

  Note that Graph<N> and Tree<N> do *not* contain methods for mutable operations, such as adding or removing vertices and edges. These interfaces are described in detail in the source code.
- The interfaces declare *abstract data types* for graphs and trees, which specify mathematical *abstract values* and their associated operations.
  - Abstract values of graphs are pairs $G = (V, E)$, where each vertex in V has type N.
  - Abstract values of trees are triples $T = (V, E, v_{root})$, where v_{root} is the root.

  The implementation may use concrete values with extra information invisible to the client. This is a form of *information hiding*, one of the fundamental concepts of object-oriented programming.
- The goal is to write formal abstract specifications of these interfaces *with respect to abstract values*; namely, a class invariant of each interface, and a precondition and a postcondition of each method.
- Fill out the attached Markdown file `homework3.md`, indicating interfaces and methods for which you need to write formal abstract specifications, including some notations and examples.

Problem 2: Behavioral Subtypes of Graphs and Trees

- The Liskov substitution principle states that if type S is a subtype of T, then code written for objects of type T also operates correctly for objects of type S.
  - In other words, objects of type T can be substituted with objects of type S without altering any of the properties of T, such as class invariants, preconditions, postconditions, etc.
  - As shown in the class, subclassing does *not* guarantee subtyping, and trying to meet the Liskov substitution principle for subclassing is a good software design practice.
- The goal is to identify whether the abstract interfaces satisfy the Liskov substitution principle; that is, to answer the following questions:
  - is Tree<N> a subtype of Graph<N>?
  - is MutableGraph<N> a subtype of Graph<N>?
  - is MutableTree<N> a subtype of Tree<N>?
  - is MutableTree<N> a subtype of MutableGraph<N>?
- For each question, explain your reasoning *using the abstract specifications that you have defined in Problem 1.* For types $\mathsf{S}, \mathsf{T} \in \{\mathsf{Tree<N>}, \mathsf{MutableGraph<N>}, \mathsf{MutableTree<N>}, \mathsf{MutableTree<N>}\}$:
  - If S is a subtype of T, explain why S has a stronger specification than T in terms of their specifications (preconditions, postconditions, and class invariants).
  - If S is *not* a subtype of T, (i) explain which part of the specifications violate the Liskov substitution principle, and (ii) show code written for `T` that behaves differently for S.
- Similarly, fill out the attached file `homework3.md`. Note that you can easily write math expressions using GitLab Markdown: `https://docs.gitlab.com/ee/user/markdown.html#math`.

Problem 3: Black-box Test Cases for Graphs and Trees

- The goal of this problem is to write a high-quality test suite for the interfaces MutableGraph<N> and MutableTree<N> with respect to their specifications.
  - Because only abstract specifications are available, you will write *black-box test cases* for the interfaces, based on equivalence partitioning.
  - E.g., for the method `addVertex`(v) of MutableGraph<N>, there are two equivalence classes based on the description: v is already in the graph, or v is previously not in the graph.
- For each method, write a test method for each equivalence class in the abstract test classes in the `src/test` directory (e.g., two test methods for `addVertex`, which are already given in the code).
  - Two abstract classes are given: AbstractMutableGraphTest<N,G> for vertex type N and graph type G, and AbstractMutableTreeTest<N,T> for vertex type N and tree type T.
  - Each abstract test class contains one object (either a graph of type G or a tree of type T), and eight vertices of type N, along with some example test methods using them.

Problem 4: Implementing Graphs

- In this problem, we will implement a direct graph using an *adjacency list representation*[1]
  - You must use the following representation provided in the class AdjacencyListGraph<N>, a (sorted) map from vertices to the (sorted) set of their adjacent vertices.

```
private final @NotNull SortedMap<V, SortedSet<V>> adjMap;
```

[1] `https://en.wikipedia.org/wiki/Adjacency_list`

- For example, the graph in Fig. 1 is represented as the sorted map[2]

$$\{1 \mapsto \{1, 2\}, 2 \mapsto \emptyset, 3 \mapsto \{6\}, 4 \mapsto \{1, 2\}, 5 \mapsto \emptyset, 6 \mapsto \{3\}\}$$

- Implement the class AdjacencyListGraph<N>, which is a subclass of MutableGraph<N>, using this representation of directed graphs.
  - What are an abstract function and a class invariant for AdjacencyListGraph<N>? Document the abstraction function and class invariant.
  - Implement the method `checkInv` that checks your class invariant. The method `toString` provides a string representation for abstract values.
- Whenever you write a method, check whether your implementation passes your black-box test cases that you have written for Problem 3, following test-driven development practice.
  - The test class StringAdjacencyMutableGraphTest extends your AbstractMutableGraphTest. It contains `setUp()` to initialize abstract graphs and vertices for black-box test cases.
  - You may write more *white-box test cases* to StringAdjacencyMutableGraphTest to achieve more code coverage, if needed.

Problem 5: Implementing Trees

- In this problem, we will write two different implementations of a tree using different representations.
  - DelegateTree<N> uses an instance of MutableGraph<N> to implement its functionality.[3]
  - ParentPointerTree<N> uses *pointers to parent vertices*[4] to represent a rooted tree.
- Implement DelegateTree<N> and ParentPointerTree<N>. Both are subclasses of MutableTree<N> *with the same specification* but with different representations.
  - What are an abstract function and a class invariant of these classes? Document the abstraction function and class invariant for each class (as comments in each class).
  - Implement the method `checkInv` that checks your class invariant for each class. Similarly, the method `toString` provides a string representation for abstract values.
  - You may find that the provided method `findReachableVertices` is useful for implementing some methods declared in MutableGraph<N>.
- Again, following test-driven development practice, run you black-box test cases that you have written for Problem 3 whenever you write a method.
  - There are two test classes that extend your AbstractMutableTreeTest, along with appropriate `setUp()`: IntegerDelegateMutableTreeTest and DoubleParentPointerMutableTreeTest.
  - Similarly, you may write more *white-box test cases* to these test classes to achieve more code coverage for DelegateTree<N> and ParentPointerTree<N>, respectively, if needed.

General Instruction

- Your code need to be compiled using only Maven in a command line for grading. You MUST ensure that your tests pass on your code using mvn test.
- The `src/main` directory contains the skeleton code. You should implement all the methods marked with *TODO*. Before writing code, read the description in the source code carefully.

[2] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/SortedMap.html
[3] https://en.wikipedia.org/wiki/Delegation_(object-oriented_programming)
[4] https://en.wikipedia.org/wiki/Parent_pointer_tree

- The `src/test` directory contains test classes. Use JaCoCo to find out how much coverage your tests have. Upload the JaCoCo report in CSV format from `target/site/jacoco/jacoco.csv`.

- Do not modify the existing interfaces, the class names, and the signatures of the public methods and checkInv(). You can add more private methods if you want.

  - In this assignment, we use *fixed* representations for AdjacencyListGraph<N>, DelegateTree<N>, and ParentPointerTree<N>. *You cannot add even private member variables to these classes.*

- Your submitted tests need to achieve at least 90% **branch coverage**. Your black-box test cases should already give high coverage, but you may add more white-box test cases if needed.

  - Your black-box test cases will be graded according to whether they clearly describe different scenarios from the specifications using equivalence partitioning.
  - *Do not add arbitrary code to your test method to just increase coverage.* In particular, this will severely affect your scores for black-box test cases.
  - The abstract test classes should only depend on abstract interfaces, namely, MutableGraph<N> and MutableTree<N>; importing concrete implementations is not allowed.

Turning in

1. Create a private project with name `homework3` in `https://csed332.postech.ac.kr`, and clone the project on your machine.

2. Commit your changes in your `homework3` project, including `homework3.md` and a JaCoCo coverage report, and push them to the remote repository.

3. The JaCoCo coverage report, generated by mvn jacoco:report, and `homework3.md`, containing the answers of Problems 1 and 2, will be uploaded to the directory `homework3/`.

4. Tag your project with "submitted" and submit your homework. We will use the tagged version of your project for grading.

Reference

- Java Language Specification: `https://docs.oracle.com/javase/specs/`

- Java Generics: `https://docs.oracle.com/javase/tutorial/java/generics/`

- Beginning Java 9 Fundamentals 2nd by Kishori Sharan, Apress, 2017 (available online at the POSTECH digital library `http://library.postech.ac.kr`)

- Maven Getting Started Tutorial: `https://maven.apache.org/guides/getting-started/`