# Problem Set 2: Poetic Walks

ocw.mit.edu/ans7870/6/6.005/s16/psets/ps2

The purpose of this problem set is to practice designing, testing, and implementing abstract data types. This problem set focuses on implementing mutable types where the specifications are provided to you; the next problem set will focus on immutable types where you must choose specifications.

**Design Freedom and Restrictions**

On several parts of this problem set, the classes and methods will be yours to specify and create, but you must pay attention to the **PS2 instructions** sections in the provided documentation.

**You must satisfy the specifications of the provided interfaces.** In some cases, you are permitted to **strengthen** the provided specifications or **add** new methods, but in other cases you are **not permitted to change them at all** .

## Get the code

To get started, download the assignment code and initialize a repository. If you need a refresher on how to create your repository, see Problem Set 0 .

## Overview

The Java Collections Framework provides many useful data structures for working with collections of objects: lists , maps , queues , sets , and so on. It does not provide a **graph** data structure. Let's implement graphs, and then use them to create timeless art of unparalleled brilliance.

**Problems 1-3:** the type we will implement is **`Graph<L>`** , an abstract data type for mutable weighted directed graphs with labeled vertices.

**mutable graph**
vertices and edges may be added to and removed from the graph

**directed edges**
edges go from a source vertex to a target vertex

**weighted edges**
edges are associated with a positive integer weight

**labeled vertices**
vertices are distinguished by a label of some immutable type, for example they might have `String` names or `Integer` IDs

Read the **Javadoc documentation for `Graph`** generated from `src/graph/Graph.java` .

`Graph<L>` is a generic type similar to `List<E>` or `Map<K,V>`. In the specification of `List`, `E` is a placeholder for the type of elements in the list. In `Map`, `K` and `V` are placeholders for the types of keys and values. Only later does the client of `List` or `Map` choose particular types for these placeholders by constructing, for example, a `List<Clown>` or a `Map<Car,Integer>`.

The specification of a generic type is written in terms of the placeholders. For example, the specification of `Map<K,V>` says that `K` must be an immutable type: if you want to make something the key in a `Map`, it shouldn't be a mutable object because the `Map` may not work correctly.

In our specification for `Graph<L>`, we make the same demand about the immutability of type `L`. Clients of `Graph` who try to use mutable vertex labels have violated the precondition. They cannot expect correct behavior.

For this problem set, we will implement `Graph` twice, with two different reps, to practice choosing abstraction functions and rep invariants and preventing rep exposure. There are many good reasons for a library to provide multiple implementations of a type (for example, the `ArrayList` and `LinkedList` implementations of `List` satisfy clients with different performance requirements), and we're following that model.

**Problem 4:** with our graph datatype in hand, we will implement **`GraphPoet`**, a class for generating poetry using a word affinity graph.

Our poet will be able to take an input like this phrase from Tolkien:

> They spoke no more of the small news of the Shire far away, nor of the dark shadows and perils that encompassed them, but of the fair things they had seen in the world together

and generate, with help from The Bard, a stanza addressed to some mysterious figure:

> They spoke no *love,* more *than* of *all* the small news of *all* the Shire far away,
> nor *thou* of *all* the dark shadows and perils that encompassed them,
> but *love* of *all* the *outward* fair *in* things they had *not* seen in *all* the *wide* world together

Read the **Javadoc documentation for `GraphPoet`** generated from `src/poet/GraphPoet.java`.

`GraphPoet` will come in handy when you take 21W.772 next semester.

---

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

Devise, document, and implement tests for `Graph<String>`.

For now, we'll only test (and then implement) graphs with `String` vertex labels. Later, we'll expand to other kinds of labels.

In order to accommodate running our tests on *multiple implementations* of the `Graph` interface, here is the setup:

- The testing strategy and tests for the **static** `Graph.empty()` method are in **`GraphStaticTest.java`** . Since the method is static, there will be only one implementation, and we only need to run these tests once. We've provided these tests. You are free to change or add to them, but you can leave them as-is for this problem.

- Write your testing strategy and your tests for all the **instance** methods in **`GraphInstanceTest.java`** . In these tests, you must use the `emptyInstance()` method to get fresh empty graphs, *not `Graph.empty()` !* See the provided `testInitialVerticesEmpty()` for an example.

**Java note**

`GraphInstanceTest` is an abstract class . Abstract classes and subclassing have their uses, but in general should be avoided.

Unlike `GraphStaticTest` , which works like any other JUnit test class we've written, `GraphInstanceTest` is different because you can't run it directly. It has a blank waiting to be filled in: the `emptyInstance()` method that will provide empty `Graph` objects. In the next problem, we'll see two *subclasses* of `GraphInstanceTest` that fill in the blank by returning empty graphs of different types.

Your tests in `GraphInstanceTest` must be legal clients of the `Graph` spec. Any tests specific to your implementations will go in the subclasses in the next problem.

**Commit to Git.** Once you're happy with your solution to this problem, commit to your repo!

---

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

Now we'll implement weighted directed graphs with `String` labels — **twice** .

For **all** the classes you write in this problem set:

- **Document the abstraction function and representation invariant** .
- Along with the rep invariant, **document how the type prevents rep exposure** .
- **Implement `checkRep`** to check the rep invariant.
- **Implement `toString`** with a useful human-readable representation of the abstract value.

In the future, every immutable type you write will override `equals` and `hashCode` to implement the `Object` contract as described in class 15 on equality, but that's **not** required for this problem set.

All your classes must have clear **documented specifications** . This means every method will have a Javadoc comment, except when you use `@Override` :

> When a method is specified in an interface and then implemented by a concrete class — for example, `add(..)` in `ConcreteEdgesGraph` is specified in `Graph` — **the `@Override` annotation with no Javadoc comment indicates the method has the same spec** . Unless the concrete class needs to strengthen the spec, don't write it again (DRY).

You can choose either `ConcreteEdgesGraph` or `ConcreteVerticesGraph` to write first. There should be **no dependence or sharing of code** between your two implementations.

## 2.1. Implement `ConcreteEdgesGraph`

For `ConcreteEdgesGraph` , you must use the rep provided:

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

You may not add fields to the rep or choose not to use one of the fields.

> **Java note**
>
> We require that class `Edge` be declared in `ConcreteEdgesGraph.java` .
>
> However, you may move `Edge` to be a static nested class or instance inner class of `ConcreteEdgesGraph` if desired.
>
> The identical restriction and permission apply to `ConcreteVerticesGraph` 's `Vertex` .

For class `Edge` , you define the specification and representation; however, `Edge` must be **immutable** .

Normally, implementing an immutable type means implementing `equals` and `hashCode` as described in class 15 on equality. That's **not** required for this problem set. Be careful: not implementing equality means you cannot use `equals` to compare `Edge` objects! Specify and implement your own observer operations if you need to compare edges.

After deciding on its spec, devise, document, and implement tests for `Edge` in `ConcreteEdgesGraphTest.java` .

Then proceed to implement `Edge` and `ConcreteEdgesGraph` .

Be sure to use the `@Override` annotation on `toString` to ensure that you are correctly overriding the `Object` version of the method, not creating a new, different method.

You should strengthen the spec of `ConcreteEdgesGraph.toString()` and write tests for it in `ConcreteEdgesGraphTest.java` . Don't add those tests to `GraphInstanceTest` , which must test the `Graph` spec. All Java classes inherit `Object.toString()` with its very underdetermined spec. `Graph` doesn't specify a stronger spec, but your concrete classes can.

Run the tests by right-clicking on `ConcreteEdgesGraphTest.java` and selecting *Run As → JUnit Test* .

**Commit to Git.** Once you're happy with your solution to this problem, commit to your repo!

## 2.2. Implement `ConcreteVerticesGraph`

For `ConcreteVerticesGraph` , you must use the rep provided:

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

You may not add fields to the rep.

For class `Vertex` , you define the specification and representation; however, `Vertex` must be **mutable** .

After deciding on its spec, devise, document, and implement tests for `Vertex` in `ConcreteVerticesGraphTest.java` .

Then proceed to implement `Vertex` and `ConcreteVerticesGraph` .

You should strengthen the spec of `ConcreteVerticesGraph.toString()` and write tests for it in `ConcreteVerticesGraphTest.java` . Don't add those tests to `GraphInstanceTest` , which must test the `Graph` spec.

Run the tests by right-clicking on `ConcreteVerticesGraphTest.java` and selecting *Run As → JUnit Test* .

**Commit to Git.** Once you're happy with your solution to this problem, commit to your repo!

---

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

Nothing in either of your implementations of `Graph` should rely on the fact that labels are of type `String` in particular.

The spec says that vertex labels "must be immutable" and are "compared using the `equals` method." Let's change both of our implementations to support vertex labels of *any* type that meets these conditions.

### 3.1. Make the implementations generic

1. Change the declarations of your concrete classes to read:

   ```
   public class ConcreteEdgesGraph<L> implements Graph<L> { ... }

   class Edge<L> { ... }
   ```

   and:

   ```
   public class ConcreteVerticesGraph<L> implements Graph<L> { ... }

   class Vertex<L> { ... }
   ```

2. Update both of your implementations to support any type of vertex label, using placeholder `L` instead of `String` . Roughly speaking, you should be able to find-and-replace all the instances of `String` in your code with `L` !

   This refactoring will make `ConcreteEdgesGraph` , `ConcreteVerticesGraph` , `Edge` , and `Vertex` generic types.

   - Previously, you might have declared variables with types like `Edge` or `List<Edge>` . Those will need to become `Edge<L>` and `List<Edge<L>>` .
   - Similarly, you might have called constructors like `new ConcreteEdgesGraph()` or `new Edge()` . Those will need to become, for example, `new ConcreteEdgesGraph<String>()` and `new Edge<L>()` . Depending on context, you can use diamond notation `<>` to avoid writing type parameters twice.

   When you're done with the conversion, all your instance method tests should pass.

**Commit to Git.** As always, once you're happy with your solution to this problem, commit to your repo!

### 3.2. Implement `Graph.empty()`

1. Pick one of your `Graph` implementations for clients to use, and implement `Graph.empty()` . Unlike `ArrayList` and `LinkedList` , where clients who only want a `List` are forced to suffer a dose of rep exposure by calling a concrete constructor, clients of `Graph` should not be aware of how we've implemented it.

2. Because the implementation of `Graph` does not know or care about the actual type of the vertex labels, our test suite with `String` labels is sufficient.

   However, since this is our first generic type, to gain confidence that you do indeed support different types of labels, add a few additional tests to the provided **`GraphStaticTest.java`** that create and use `Graph` instances with a couple different types of (immutable) labels.

**Java note**

You may use `@SafeVarargs` where necessary, for example if you write testing helper functions that take a variable number of arguments .

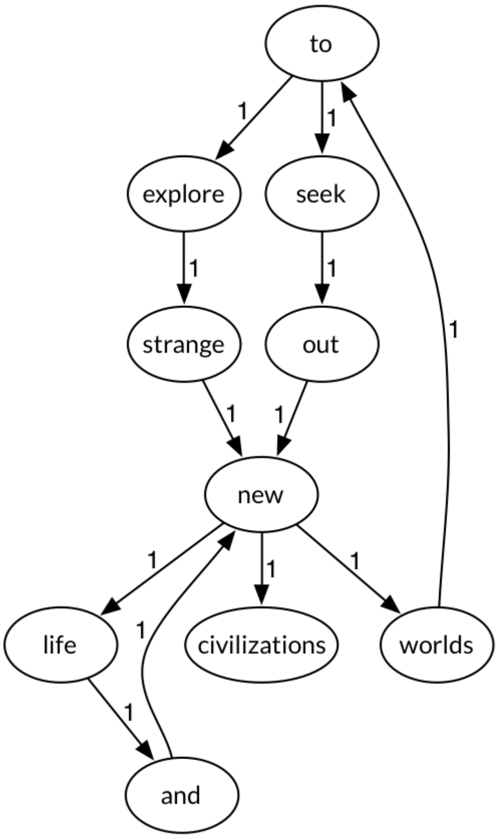
At this point, all of your code (implementations and tests) must have **no warnings** from the compiler (warnings have a ⚠ symbol , as opposed to the ⊗ for errors) and no `@SuppressWarnings` annotations (which would disable the warning, nice try).

Run *all* the tests in the project by right-clicking on the `test` folder and selecting *Run As → JUnit Test* .

**At this point in the problem set, we've completed the test-first implementation of `Graph` .**

- You should have a green bar from JUnit and excellent code coverage from EclEmma.
- This is a good opportunity for self code review: remove dead code and debugging `println` s, DRY up duplication, and so on.
- And of course, commit to your repo.

---




## Problem 4: Poetic walks

Graphs — what are they good for? Poetry!

The specification of `GraphPoet` explains how a poet is initialized with a corpus and then, given an input, uses a word affinity graph defined by that corpus to transform the input poetically.

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

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

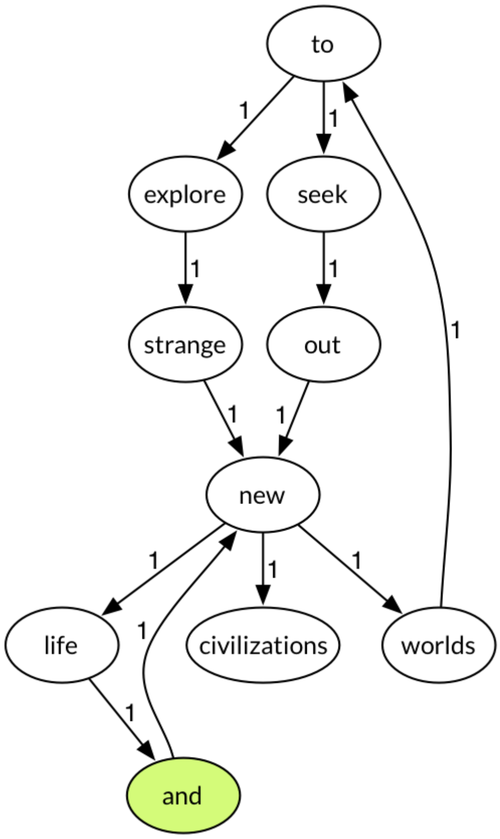
`GraphPoet` will use this corpus to generate a word affinity graph where:

- vertices are case-insensitive words, and
- edge weights are in-order adjacency counts.

The graph is shown on the right. In this example, all the edges have weight 1 because no word pair is repeated.

From there, given this dry bit of business-speak input:

> Seek to explore new and exciting synergies!




the poet will use the graph to generate a poem by inserting (where possible) a *bridge word* between each pair of input words:

| **w 1** | | **w 2** | **path?** | **bridge word** |
|---|---|---|---|---|
| *seek* | → | *to* | no two-edge-long path from *seek* to *to* | — |
| *to* | → | *explore* | no two-edge-long path from *to* to *explore* | — |
| *explore* | → | *new* | one two-edge-long path from *explore* to *new* with weight 2 | *strange* |
| *new* | → | *and* | one two-edge-long path from *new* to *and* with weight 2 | *life* |
| *and* | → | *exciting* | no vertex *exciting* | — |
| *exciting* | → | *synergies!* | no vertices *exciting* or *synergies!* | — |

Hover over rows above to visualize poem generation on the right. In this example, the outcome is deterministic since no word pair has multiple bridge words tied for maximum weight. The resulting output poem is:

> Seek to explore strange new life and exciting synergies!

Exegesis is left as an exercise for the reader.

### 4.1. Test `GraphPoet`

Devise, document, and implement tests for `GraphPoet` in **`GraphPoetTest.java`** .

You are free to add additional methods to `GraphPoet` , but you may not change the signatures of the required methods. You are free to strengthen the specifications of the required methods, but you may not weaken them. Note how this might mean your tests for `GraphPoet` are not usable against someone else's implementation, because you've strengthened or added to the spec.

Your tests should make use of one or more sample corpus files. Put those files in the `test/poet` directory — the same directory as `GraphPoetTest.java` . Reference them in the code with, for example:

```
new File("test/poet/seven-words.txt")
```

While you still must test and implement `GraphPoet(File)` , remember that you may add operations to `GraphPoet` . Adding a creator that takes a `String` , for example, might simplify your tests for `poem(..)` at the cost of additional tests for the new creator.

### 4.2. Implement `GraphPoet`

Implement `GraphPoet` in **`GraphPoet.java`** .

You must use `Graph` in the rep of `GraphPoet` , but the implementation is otherwise entirely up to you.

For reading in corpus files, there are at least three Java APIs to consider:

**Java note**

Another option, if you want to program in a functional style: use `Files.lines(..)` , which returns a `Stream<String>` .

- `FileReader` is the standard class for reading from a file. Wrapping the `FileReader` in a `BufferedReader` allows you to use the convenient `readLine()` method to read whole lines at a time.
- The utility function `Files.readAllLines(..)` will read all the lines from a `Path` . Our poet takes a `File` , but `File` provides a `toPath()` method to obtain a `Path` .
- Class `Scanner` is designed for breaking up input text into pieces. It provides many features and you'll need to read its specs carefully.

### 4.3. Graph poetry slam

If you want, update the `main(..)` method in **`Main.java`** with a cool example: an interesting poem from minimal input, a surprising poem given the corpus and input, or something else cool.

Put the corpus for your example in `src/poet` — the same directory as `mugar-omni-theater.txt` .

---

## Before you're done

- Make sure you have **documented specifications** , in the form of properly-formatted Javadoc comments, for all your types and operations.

- Make sure you have **documented abstraction functions and representation invariants** , in the form of a comment near the field declarations, for all your implementations.

  With the rep invariant, also say how the type prevents rep exposure.

  Make sure all types use `checkRep` to check the rep invariant and implement `toString` with a useful human-readable representation of the abstract value.

- To gain static checking of the correct signature, use `@Override` when you override `toString` .

  Also use `@Override` when a class implements an interface method, to remind readers where they can find the spec.

- Make sure you have a thorough, principled test suite for every type.

---