Project 1 – Graph ADT

CSc 260, Winter 2023

Aaron G. Cass
Union College

Objectives

In this project, you will develop a piece of software according to an API that I give you. In the process, you will learn how to work from a relatively small, simple API to a relatively large, more complex API in a progressive manner. You'll also use unit test cases and generate your own.

You will:

- Become comfortable using:
  - version control and configuration management tools,
  - build tools, and
  - unit testing frameworks.
- Become comfortable thinking abstractly about how a piece of software might be implemented.
- Gain some experience with class invariants and method pre- and post-conditions.

To Do

Follow these directions **carefully**:

1 Initialize git repositories

1. If you do not have an account on our gitlab server (`cs-gitlab.union.edu`), please create an account. For the account name, **you must use your Union College user name**. You'll need to create public/private key pairs and configure your cs-gitlab account to use them.
2. Once you have an account, use the web interface to create a new repository/project. Please call it exactly `csc260-w2023-project1`. **Share** the project with me by giving me *Maintainer* privileges.
3. On your development computer, **clone** the repository with a command like this:

   ```
   git clone git@cs-gitlab.union.edu:<uname>/csc260-w2023-project1
   ```

   where `<uname>` is your user name.
4. Cloning creates a copy of the repository from the server into a **working directory** where you can make edits to the project. If you used the command above, your working directory will be called `csc260-w2023-project1`. Now, change into that working directory [e.g. `cd csc260-w2023-project1`].
5. **Download the starter code** from the course web-site. The starter code includes:
   - A `src` directory with java source files in it. The `src` directory has `main` and `test` sub-directories, where your main source code and test cases will be, respectively. The starter code already has a `Graph.java` source file for a graph ADT and some test cases written in jUnit4.
   - A build file to be used with the gradle build system.
6. When you move the starter code to your project directory, **make sure** the `src` directory and the build file are **directly** inside the project directory (i.e. not down one additional level inside a `graphV1` directory).

7. Once the starter code is in your project directory, tell git to manage the files: `git add src` and then `git add build.gradle` will tell git to manage revisions to all the files that are in the `src` directory (and sub-directories) and the file `build.gradle`. If you add new files to your project, you can manage them with `git add ...`. **Note:** all the files in the `build` directory are generated from those in the `src` directory, so they should **not** be added to the repository.

8. You now need to **commit** these changes to the git repository. Every time you commit changes, you tell git that you have made a change to the files in your working directory that you want to record so you can get the files back to this state later.

   Use `git help commit` to see the options for the commit command. In our case, we can use a simple commit command:

   ```
   git commit -a -m 'initial commit'
   ```

   The `-m message` is used to tell git what message to record with this commit. Every commit needs a commit message, and the messages **should be descriptive of the changes you've made** to the software since the last commit (note that this initial commit message is *not* very descriptive, but your future commits need to have messages that will remind you later what you changed in that version). If you omit the message on the command line, git will launch an editor for you to provide a message (it will launch an editor you probably won't like, unless you set the `GIT_EDITOR` environment variable to use an editor you do like).

9. You can **push** these changes to the server repository with a command like (use `git help push` to understand the role of the command line options):

   ```
   git push origin main
   ```

2 Build the software

Now that you have the code in a git repository, you are ready to build the software. Try one of the following build commands:

- `gradle compileJava` compiles the java files in `src/main/java`.
- `gradle compileTest` compiles the jUnit4 test cases in `src/test/java`.
- `gradle build` compiles all the code, compiles the tests, and runs the tests.
- `gradle javadoc` creates javadocs for the code in `src/main/java`.

Notice that each of these commands produce output in the `build` directory (it will be created if it is not already there). Also note that the gradle build system manages dependencies between these different build tasks. So, for example, if you have changed the main source code, `gradle compileTest` will compile the new source code before compiling the tests.

Also notice that the code you are given does not pass the tests. So, if you use a gradle task that runs the test cases, you will see that the build fails – it will still compile the source and tests first though.

If you ever want to clean up the `build` directory, you can use `gradle clean`.

To see what other build tasks are available, use `gradle tasks --all`.

3 Implement the Graph ADT

1. **READ** the javadocs for the Graph ADT. This will give you a sense of what your are trying to implement.

2. **READ** the test cases in `src/test/java`. This will give you an even finer-grained understanding of what you are trying to implement. Make sure you understand what the tests are doing and **why** they expect the behavior they expect.

3. **Use a text editor** to edit the `Graph.java` to implement the specified methods. Your goal is to implement the methods to satisfy the specifications described in the javadocs *and* to pass the tests. I'd prefer that you use a regular old text editor (I like Emacs or Atom) instead of a full Integrated Development Environment (such as Eclipse or IntelliJ). The IDE does a lot for you, and we want to be sure you understand all of the steps involved.

4. **DO NOT** write all of the methods at once. Work on one method at a time. When you think you've made a reasonable implementation of one method, use gradle to rebuild and test the code. Because you've read the test cases, you should be able to figure out which tests should pass if you've written this one method correctly.

5. **Use git** often. Get in the habit of using git to commit changes as you work. For example, you could adopt a policy that after you have a new method working (passing the tests), you commit those changes. Remember to use good commit messages that you or someone else (like me) could refer to in order to remember what you were doing at that point in development. You can also **push** those changes to the server.

6. **DO NOT** change the API of the Graph ADT. My test cases depend on that API staying constant.

7. You have quite a bit of flexibility in how you implement the Graph ADT:

   - You can use an adjacency list or adjacency matrix design. I think adjacency lists are a bit easier to implement, but it's your choice, as long as the API doesn't change.
   - You are not limited to just `Graph.java` – you *may* (but are **not** required to) add additional classes that `Graph` uses to get the job done.
   - I encourage you to use the collections classes provided in `java.util`. For example, you might use classes that implement `java.util.Set` or `java.util.Map` in order to write the code for `Graph`.

4 Turn in your implementation

We will use your git repo on the server to share code with each other. So, to turn in your implementation, make sure everything is committed and then push to the server, using the same command you used before (`git push origin main`).

Be sure to check that you've pushed everything correctly, which you can do in one of two ways:

1. Use the web interface to inspect what's there

2. Clone a new copy of your project to make sure it has all you need:

   (a) Create a new temporary directory somewhere (not in your working directory) and clone the repository from the server:

   ```
   git clone git@cs-gitlab.union.edu:<uname>/csc260-w2023-project1
   ```

   (b) Navigate through the created directory to make sure all of your code is there. You can even run gradle build tasks to make sure it still works.

   (c) If this all worked right, that means the server's repository has all of your code. You can then delete the temporary directory and go back to working in your working directory.

5 Refactor

I will give you a new API to follow. Refactor your code to follow the new API. The basic implementation will stay the same, but the design will differ slightly.

You will need to **refactor the tests** as well so that they use the new API instead of the old API. Make sure your new code passes the tests.

When you are done, make sure you **push your changes** to the repository on our server.

6 Write tests for an expanded API

I will give you an expanded API to follow. The expanded API will have javadoc comments that specify some method pre- and post-conditions as well as class invariants. Based on this documentation, write tests to test the new API:

1. Your tests should go in a separate package: `edu.union.adt.graph.tests.<uname>` (where `<uname>` is your user name) so that I can create a single test suite with your tests **and** tests from other students without having to worry about naming conflicts. Therefore:

   - The java files for your tests should have a `package` statement.
   - The java files must be in the appropriate folder:

     ```
     src/test/java/edu/union/adt/graph/tests/<uname>
     ```

   Otherwise, your tests will not properly be in the correct package.

2. Your tests can assume that the implementation passes the tests for the basic API, so your tests only need to test the new methods (though the tests can use the existing methods).

3. Do not add your tests to the test file I've given you – add a test file and make sure that it is included in the test suite so that it will be run when you use `gradle test`.

4. You should try to make an overall test suite that is efficient. Don't add lots of tests that are really testing for the same defects. You are only allowed to add **at most 10 test cases**.

5. You are **not allowed** to write the code for the extended API until after you've written tests. Imagine that you are writing tests for someone else's implementation, so you don't have the code in hand right now, and you don't need to know how the implementation works to write good tests.

When you are done, make sure you **push your changes** to the repository on our server.

7 Implement the extended API

Now that you understand what the extended API is supposed to do (now specified in the javadocs **and** in test cases you have written), implement to this new API:

1. Again, only work one method at a time.
2. Again, test often.
3. Again, use git regularly.

When you are done, make sure you **push your changes** to the repository on our server.

Evaluation

Because there are several steps in this project, there are several aspects I will evaluate:

- Defects in your code. Does your code pass my tests? Does it pass your tests? Does it pass tests written by your classmates?
- Test quality. Do your written tests exercise the code of your classmates enough to find defects?
- Process. Do you use git appropriately? Are you working a small part at a time, or coding all at once?
- Refactoring. Does your refactoring produce code that still passes all the tests? Are you following the design changes correctly?

Due Dates

Step	Due Day
Project assigned	t
Initial Graph implementation	t + 5 days
Refactoring	t + 7 days
Tests for expanded API	t + 12 days
Expanded API implementation	t + 19 days