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 a git repository

Create a personal git repository for your development. You will use this repository to keep track of revisions you make as you develop the software for this assignment.

On the department machines, you can do the following in a command line shell:

1. Create a directory anywhere under you home directory [e.g. mkdir csc260Project1]. You can put this directory anywhere you’d like and it can be named whatever you want. We’ll call this directory your working directory.
2. Change into your working directory [e.g. cd csc260Project1].
3. Initialize a new git repository: git init. This will create some files in a hidden directory. These files are used by git to keep track of versions for you.
4. 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.
5. 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.
6. 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. 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).

## 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) 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.

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.
   - 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 git to share code with each other. Up until now, you’ve been managing versions using a local git repository. Git also has the ability to work with remote repositories. You will push your changes to a remote repository I’ll set up for you.

1. Before you proceed, make sure you have all of your changes committed to your local git repository.
2. Name a remote repository for your local git repository:
   ```
git remote add turnin git@cs-gitlab.union.edu:<repo-name>
   ```
   Replace `<repo-name>` with the name I give you for your repository on our server.  
   This tells git to add a remote repository named `turnin` to your local repository. This is just associating a name with a url. It doesn’t actually download or upload any files. You can name the remote whatever you want – it’s just giving an alias to the repository on the server and only you will see this alias.
3. Push changes to the remote repository:
   ```
git push turnin master
   ```
   This tells git to push your changes to the “master” branch of the repository you’ve named “turnin”.

4. Check that you’ve pushed everything correctly.
   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:<repo-name>
   ```
   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.
   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 might need to refactor the tests as well. 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.

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 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?
- 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

<table>
<thead>
<tr>
<th>Step</th>
<th>Due Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project assigned</td>
<td>t</td>
</tr>
<tr>
<td>Initial Graph implementation</td>
<td>t + 7 days</td>
</tr>
<tr>
<td>Refactoring</td>
<td>t + 9 days</td>
</tr>
<tr>
<td>Tests for expanded API</td>
<td>t + 14 days</td>
</tr>
<tr>
<td>Expanded API implementation</td>
<td>t + 21 days</td>
</tr>
</tbody>
</table>