SIMON FRASER UNIVERSITY
Department of Computing Science
CMPT 276: Assignments

Assignment 3: Test-Driven Development

This is an individual assignment. Each student has to complete the assignment separately.

The TAs will mark the history and outcome of the assignment on SFU GitHub on the specified deadline. See the Schedule page on Canvas for deadlines.

1 Test-Driven Development

Test-Driven Development (TDD) is an approach where tests are developed before the software is fully developed, based on the software's specifications or requirements. In other words, test cases are created first and thus will initially fail. Then new functionality is added repeatedly in order to pass the tests.
In this assignment, we want to follow this process for implementing and testing generic sorters.
Consider the interface below, which specifies a Sorter that sorts the elements of an array in place.

```
/**
 * Generic interface for sorting an array of elements in-place.
 */
public interface Sorter<T extends Comparable<T>> {
   void sort(T[] list);
}
```

1.1 Setup

You must use Java for implementation and JUnit for testing. You must use Apache Maven[1] for building the assignment and running the tests. You will need to install and use an automated tool for measuring test coverage. An example of such a tool is the EclEmma tool[2] that can be installed as a plugin for Eclipse.

1.2 Functional Testing

Your first task is to apply functional (specification-based/black-box) testing techniques to create a suite of at least 10 tests to test your sorters thoroughly. The tests should verify the sorters' behaviour for different arrays that can have various lengths (e.g., 0, 1, or many items) and data types (e.g., array of Strings or array of Integers).
Continually commit these tests to your repository. In these tests, you may assume for now that you can create a sorter by calling a method `createSorter()`. All tests will initially fail as you haven't implemented any sorters yet.

1.3 Implement the Sorters

Next, implement the sorters by creating two subclasses of the Sorter interface that implement sorting algorithms of your choice. These algorithms must be efficient in the sense that they perform, on average, `O(n log n)` comparisons to sort n items. For example, one of these could implement a QuickSorter class that uses C.A.R. Hoare's famous QuickSort algorithm.
Write Javadoc comments for your classes to be able to automatically generate your documentation.
Repeatedly rerun the tests as you add functionality and continually commit and push your changes.

[1] `http://maven.apache.org/`
[2] `http://www.eclemma.org/`

1.4 Structural Testing

Then, measure and report the statement and branch coverage of your tests for each of your sorters. Investigated the insufficiently covered parts of your code. If the coverage is less than 100%, extend your test suite to achieve the highest coverage you can. Measure and report the coverage of your final test suite, including both functional and structural tests you have written based on our testing guidelines. Continue to commit and push your changes.

2 Deliverables

- **The code:** Create a directory named "Assignment3" in your "276Assignments" SFU GitHub repository. This directory will contain the Maven project: your main Java files (with Javadoc comments) (`src/main/java`), test files (`src/test/java`), and pom.xml in a Maven standard project structure. Continually commit and push to the remote repository. Always pay attention to the design and quality of your code.
- **The report:** List the two testing algorithms you have chosen and briefly explain their coverage results after 1) having only functional tests, and 2) adding structural tests (max 1 page).