OOP and Data Structures	Discussion 7 — Abstract classes and refactoring
CS2110 Summer 2019	Due Wednesday, July 24 at 5PM

For this discussion, we recommend that you work in groups of two or 3, but that everyone does everything on their own computer to build muscle memory. Have all group members do each step at the same time, and make sure everyone has it before moving on. If you work together this way, you may form a group on CMS and only submit one copy of the code.

We just want you to have gone through the process; if you submit you will get full credit.

Refactoring

While discussion sorting, we came up with several different sorting algorithms, all of which share (at least parts of) their specifications.

We wrote one giant class that contains all of these sorting methods, as well as several helper methods:

```
class Sorting<E> {
  constructor

  // private helpers
  void swap(List<E>,int,int)
  void compare(List<E>,int,int)

  // sorts
  void insertionSort(List<E>)
  int  indexOfMin(List<E>, int, int)
  void selectionSort(List<E>)
  void mergeSort(List<E>)
  void mergeSort(List<E>, int, int)
  void merge(List<E>,int,int,int)
}
```

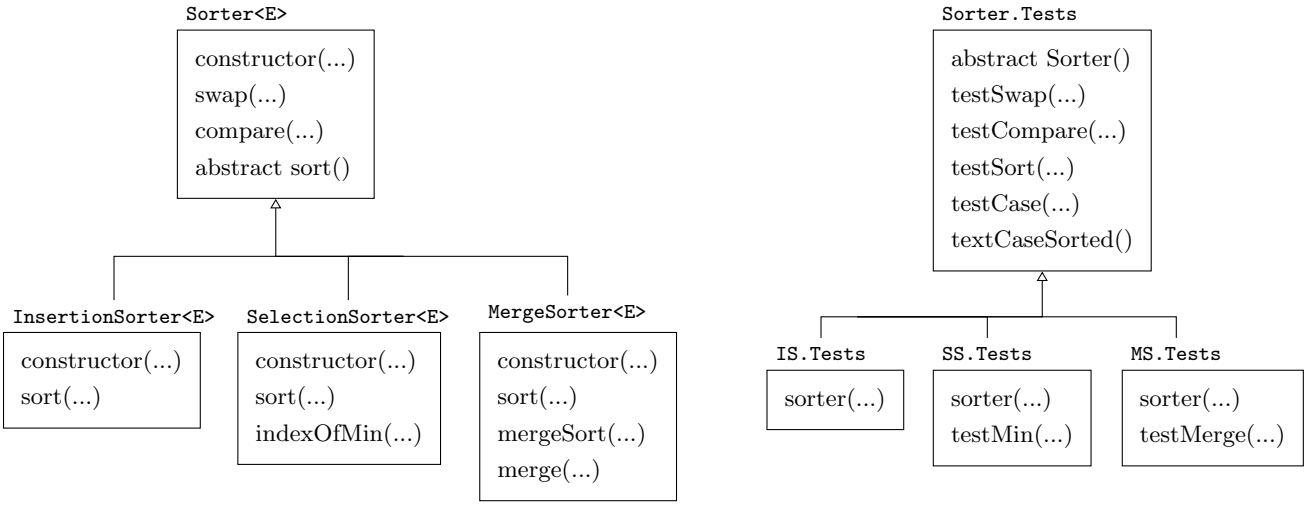
```
class Sorting.Tests {
  // helpers
  sorter()
  testCase()
  testCaseSorted()

  // tests for helpers
  testSwap()
  testCompare()
  testIndexOfMin()
  testMerge()

  // tests for sorts
  testMergeSort()
  testSelectionSort()
  testInsertionSort()
}
```

Organizing the code this way has several problems:

- **Specification duplication.** The different sorting algorithms all have the same specification, but nothing in the code points out the connection between these. That means there is extra effort that goes into writing and understanding the specifications, and that they can accidentally diverge over time.
- **Test duplication.** The fact that multiple methods share a specification means that we must duplicate the tests. The methods testMergeSort, testSelectionSort, etc. are all almost identical to each other, and duplicated code leads to duplicated bugs.
- **Large files with no protection.** Figuring out what code to look at if a problem is encountered is hard; the lest code you need to look at, the better. With all of your code in one big file, potentially any part of that file could mess with your private state. By separating unrelated methods into different classes, we reduce the set of code you need to understand while debugging.
- **Can't mix and match.** Imagine you wanted to experiment with the running time of different sorting algorithms. You would like to be able to write your measurement code once and have it work for any sorting algorithm.

When you discover unnecessary duplication, you should *refactor* your code: rearrange it to increase the sharing between different objects while separating out the parts that are different. This discussion walks you through refactoring the `Sorting` class and its tests.

We will reorganize the code as follows:



The steps below walk you through this process:

1. Create a new project and git repository. We will collect `.git` directories for this assignment.
2. Load the sample code from CMS into your new project. Make sure it compiles — you may need to add JUnit 5 to your build path or import some assertions. Commit the code.
3. We are now thinking of different sorting algorithms as objects. Objects are nouns; so "Sorter" or "SortingStrategy" would be a more appropriate name for the class. Right-click on the class and look in the "refactor" menu; Eclipse will rename the class and update all references for you.

   Make sure everything compiles, and commit.
4. We now want to make the common sorting method that our three sorter classes will implement.
   (a) Start by adding an abstract `sort` method:

   ```
   1 | public abstract void sort(List<E> a);
   ```

   The `abstract` keyword says that there is no default implementation; different subclasses will implement it differently.

   (b) You should copy the specification from `insertionSort`. We're planning to delete those duplicates soon, so this time we won't worry about copying.

   (c) The code doesn't compile; abstract methods can only occur in abstract classes (otherwise we could create a `Sorter` object and ask it to sort, but there is no `Sort` method in the `Sorter` class). Make `Sorter` abstract.

   (d) The code still doesn't compile, because the test class creates a new `Sorter` object. We want different test classes to create different `Sorter` objects, so we should make the `Tests.sorter` method abstract as well.

   (e) Get the code to compile and commit it.
5. We're going to ask eclipse to generate a bunch of code for us. We will want all of the generated code to throw `NotImplementedError`s. We can make eclipse do this automatically, as follows:
   (a) Right-click on your project and select Properties → Java Code Style → Code Templates

   (b) Click "Enable project-specific settings," or if you prefer to make this change for all projects, click "Configure workspace settings."

(c) Select Code → Method body. Press "Edit" and replace with `throw new NotImplementedError()` (leaving the comment if you wish).

(d) You can also change the Constructor Body.

(e) If you changed the project settings, the `.settings` file will have been modified; commit it.

6. Now we will create class `InsertionSorter<E>`. In the new class dialog, make `Sorter<E>` the superclass. You can ask Eclipse to create constructors from the superclass and to override inherited abstract methods. Make the generated class compile and commit it. Since the methods aren't implemented, it would be good to advertise this in your commit message.

7. Now we want to move the `insertionSort` code into the `InsertionSorter<E>` class:

   (a) Move the `insertionSort` method from `Sorter` to `InsertionSorter`, and make it the body of the `sort` method. You can delete its specification, since this is now inherited from the superclass's `sort` specification.

   (b) `InsertionSorter` doesn't compile. The helper methods `compare` and `swap` are `private`, and thus not available to the code in the

   `InsertionSorter` class. We want the methods to be `protected`, which means available to all subclasses of the containing class.

   (c) `testInsertionSort` no longer compiles, because `Sorter` doesn't have an `InsertionSort` method. In fact, this is a general test that should work for any `Sorter`. Change the method name to `testSort` and have it to call `sort` instead of `insertionSort`. Delete the other identical methods (`testMergeSort`, etc.).

   (d) Get the code to compile and commit it.

8. Our tests are no longer running! The `Tests` class is abstract, so there aren't any concrete tests to run. Create a concrete `Tests` class inside of `InsertionSorter` that inherits from `Sorter.Tests`. You will need to implement the `sorter` abstract method; return `new InsertionSorter<Integer>(Comparator.naturalOrder())`.

   You should now be able to run the tests (all should pass). Make sure the code compiles and runs, and commit it.

9. You can now move the code for `selectionSort` into its own file in the same way you did for insertion sort. You should also move the `indexOfMin` method into the `SelectionSort` class, along with its tests. You may need to change the access modifiers of some methods. You may also need to give `sorter()` a more restrictive return type.

   Make sure the code compiles, then commit it.

10. Continuing in the same fashion, factor `mergeSort` and its helper methods and tests into their own class.

11. Create a zip file containing your git repository and submit it on CMS.