CPSC 2151

Lab 8

Due: Friday, October 16th at 10:00 pm

In this lab, you will be working with JUnit to create automated testing for your implementations of the `IQueue` Interface. You will need your completed code for Lab 6 for this assignment. The instructions and provided code in Lab 7 will also be useful as an example of working with JUnit and includes a working `makefile` that you could use as a starting point to create your own.

Requirements

1. Open up your project from Lab 6 and set up the `test` directory like we did in Lab 7. Add a class called `TestArrayQueue`.
2. Before adding any test cases for `TestArrayQueue`, create a `private` method called `MakeAQueue` that takes in no parameters and returns an `IQueue<Double>`
    a. All this method will do is call the constructor for `ArrayQueue` and return the new `ArrayQueue` object (**Note:** we are coding to the interface here by returning an `IQueue` instead of an `ArrayQueue`)
    b. We are also creating an indirection for the constructor call. In all our test cases in this file, instead of calling the constructor for `ArrayQueue`, we'll call `MakeAQueue`. This accomplishes the same thing, but it means the only reference to `ArrayQueue` will be in the `MakeAQueue` method (i.e. This means when we want to test a different implementation, we just need to change the constructor call inside of `MakeAQueue`, and no other piece of code needs to change).
3. Add in 3 distinct test cases to test your `IQueue`'s `enqueue` method
    a. You'll need to use the `get` and the `length` method to confirm that `enqueue` was successful
        i. If this test case fails, this means it could be an issue with `get`! Check that as well
4. Add in 3 distinct test cases to test your `IQueue`'s `dequeue` method
5. Add in 3 distinct test cases to test your `IQueue`'s `clear` method
6. Add in 3 distinct test cases to test your `IQueue`'s `peek` method
7. Add in 3 distinct test cases to test your `IQueue`'s `endOfQueue` method
8. Add in 3 distinct test cases to test your `IQueue`'s `insert` method
9. Add in 3 distinct test cases to test your `IQueue`'s `remove` method
10. Add in 3 distinct test cases to test your `IQueue`'s `get` method
11. Create a new class called `TestListQueue`. This will be an exact copy of `TestArrayQueue`, with one key difference. The `MakeAQueue` method in `TestListQueue` will call the `ListQueue` constructor instead of `ArrayQueue`. If you coded to the interface, this should be the only change you need to make to the file for it to work on the `ListQueue` implementation.
12. Update your `makefile` from Lab 6 to add in the following targets
    a. `test` – compiles all the code including the `TestArrayQueue` and `TestListQueue`
    b. `testArr` – runs the JUnit code with the `TestArrayQueue` class
    c. `testList` – runs the JUnit code with the `TestListQueue` class
    d. All the targets required from Lab 6 should still be present and working

e. **Note:** your `makefile` will be worth more points than it was in the past!

13. Move your code to SoC Unix and ensure your `makefile` is working.
14. If any of your test cases fail, find the fault that caused the failure and fix it.

Comments and Contracts:

Since we are just writing JUnit code, we do not need any contracts. You may want to (but aren't required to) include a comment with each method saying why you think it is a distinct test case. That way if your TA thinks they are not distinct, you have your explanation as why they are to prevent losing points.

TIPS and additional Requirements

- Our test cases will only be testing one method at a time, but in order to do so you will need to rely on other methods working. For instance, in order to check if `dequeue` is working you'll need to call `enqueue` on the queue to create the input.
- When you are creating your expected output, you cannot use the same method that you are testing. For instance, when testing `enqueue` you can't just make another queue and compare to see if they are equal, because you would need to call the `enqueue` method on the "expected output" queue. You can compare the results of `toString`() with an expected string that you set up. Or you can iterate through the queue with `get` to see if everything matches.
- JUnit code has different best practices and magic numbers are allowed. We have to hard code in our expected output and our input.
- Follow the naming conventions for test cases described in the lecture. Remember if a test case fails, we only get the name of the method, so it should be named in a meaningful way.
- Each JUnit method should only test one test case. It may need multiple (JUnit) assertion statements to do so, but it should still only be testing one test case. For the `enqueue` method, you may have to call `enqueue` multiple times to set up the input (if your input involves a non-empty queue). That is fine since it is still only testing one test case, it just needs to call the method multiple times to do so.
- Make sure your test cases are distinct. Adding 7.3 to an empty queue and adding 5.1 to an empty queue are not distinct. Two test cases are distinct if there is a reason to expect that one would pass while the other would fail. There's no reason to suspect that `enqueue` would react differently to those situations.
- Even though `IQueue` is generic, we only need to test it with one data type passed in as the generic parameter. Keeping test cases the same, but changing the generic parameter is not creating distinct test cases. There is no reason to suspect it would work for a `Double` but not an `Integer`, since `IQueue` does not care what the data type is.
- Test cases that are prevented by the compiler (non-matching data types) or test cases that violate the preconditions are not valid test cases. Also, we will not be considering test cases that are using `NULL`s or `NaN`s for this assignment. We want to include test cases that are meaningful inputs that a client would want to provide, not inputs that would occur due to an error in the client code.

Groups

You may, but are not required to, work with a partner on this lab. Your partner must be in the same lab section as you, not just the same lecture section. If you work with a partner, only one person should submit the assignment. You should put the names of both partners in a comment at the top of the JUnit test files in order for both partners to get credit. This assignment may take more than just the lab time. Make sure you are able to meet outside of class to work on the assignment before you decide to work with someone else. Remember to actively collaborate and communicate with your partner. Trying to just divide up the work evenly will be problematic.

Before Submitting

You should make sure all of your code will run on SoC Unix before you submit. Make sure you correctly set up the directory structure to match the package name. JUnit is a little trickier to run on SoC Unix. Make sure you review those instructions in the video. The slides for that video are in the lecture on JUnit testing. You should use the `makefile` provided in Lab 7 as a starting point, modify it so it works for Lab 8 and include it in your submission. No late submissions will be accepted.

Submitting your file

You will submit your files using handin in the lab section you are enrolled in. If you are unfamiliar with handin, more information is available at https://handin.cs.clemson.edu/help/students/. You should submit a zipped directory with your package directory and your `makefile`. The TA should be able to unzip your directory and use any of the targets as specified above.

NOTE: Make sure you zipped up your files correctly and didn't forget something! Always check your submissions on handin to ensure you uploaded the correct zip file.