ASSIGNMENT 4

CS 454: Principles of Concurrent Programming / Spring 2022

Description

In this assignment, you will build a management utility to track diagnostic tests and their results. A diagnostic test can be taken at a testing site, which is part of a lab. It requires a sample being taken from a patient (e.g., a nose swab), and can detect the presence of a pathogen (i.e., positive), detect the absence of a pathogen (i.e., negative), or return an invalid result. Once samples are taken, they need to be stored at a controlled temperature. The capacity for testing sites to store such samples is limited. For efficiency reasons, samples may be moved between testing sites. Once a test has a result, its sample is destroyed.

Changes between Assignment 4 and Assignment 1 are highlighted in this document.

Your submission should extend the following abstract class:

```
abstract class Lab {
  abstract TestingSite createTestingSite(int capacity);

  abstract DiagnosticTest createDiagnosticTests(int id);

  abstract boolean sampleDiagnosticTest(TestingSite ts, Set diagTests);

  abstract boolean positive(TestingSite ts, Set diagTests);

  abstract boolean negative(TestingSite ts, Set diagTests);

  abstract boolean invalid(TestingSite ts, Set diagTests);

  abstract boolean moveDiagnosticTests(TestingSite from, TestingSite to, Set movedTests);

  abstract Set getDiagnosticTests();

  abstract Set getDiagnosticTests(TestingSite ts);

  abstract boolean sampleDiagnosticTestAsync(TestingSite ts, Set
diagTests);

  abstract boolean positiveAsync (TestingSite ts, Set diagTests);
  abstract boolean negativeAsync (TestingSite ts, Set diagTests);

  abstract boolean invalidAsync (TestingSite ts, Set diagTests);

  abstract boolean moveDiagnosticTestsAsync (TestingSite from, TestingSite to,
Set movedTests);

  abstract Set getDiagnosticTestsAsync ();

  abstract Set getDiagnosticTestsAsync (TestingSite ts);


  ~~abstract List<Action< TestingSite>> audit(DiagnosticTest diagTest);~~

  ~~abstract List<Action< DiagnosticTest>> audit(TestingSite ts);~~
}
```

Each operation (method) behaves as follows:

- **createTestingSite**: Creates a testing site that can store a number of samples taken for tests not yet finished – capacity
- **createDiagnosticTest**: Creates one test with a given id. The id is unique across the same Lab.
- **sampleDiagnosticTests**: Takes a sample from patients for each diagTest and stores it in the given TestingSite ts.
    - This operation either adds all the tests, if the site has enough capacity, or none.
    - For instance, attempting to add two tests to a site that only has room for one should not change the contents of the site.
    - If all the tests are added to the site, this operation returns true. If the site remains unchanged, this operation returns false.
- **positive**: Mark all the given diagnostic tests provided as positive, which should be present on the given site.
    - Similarly to **sampleDiagnosticTests**, this operation either marks all the tests or none.
    - Trying to mark a test that is not in the current site results in failure of the whole operation (i.e., no tests are marked as positive).
    - If all the tests are marked, this operation returns true. If any test cannot be marked, then no tests are modified and this operation returns false.
- **negative**: Similar to positive described above, but marks all tests as negative
- **invalid**: Similar to positive and negative described above, but marks all tests as invalid

- **moveDiagnosticTests**: Moves `diagnostic tests` currently present in the `from` site to the `to` site.
    - If there is not enough room in the `to` site, this operation should fail and return `false`.
    - If any dose is not present in the `from` site, this operation should fail and return `false`.
    - This operation returns `true` if it succeeds in moving all the doses between sites.
- **getDiagnosticTests**: Gets the tests that are sampled (i.e., added to a site, and not positive, negative, or invalid).
    - Without arguments, this operation lists all the tests that the lab produced that have been sampled but still don't have a result.
    - With a `site` argument, this operation lists all the tests currently in that site have been sampled but still don't have a result.
- **Asynchronous methods**: Each method described above has an asynchronous version
    - The asynchronous methods should return as fast as possible
    - The asynchronous methods return a Result object, which the caller can then use to retrieve the result of the operation
    - The asynchronous methods do not wait for the operation to be performed
- ~~**audit**: Returns an audit log that tracks tests and site contents.~~
    - ~~With a `diagnosticTest` argument, returns a list of all the sites in which the test was~~
        - ~~The order of the list matters~~
        - ~~When moving, tests should be removed from one site before being added to another site~~
    - ~~With a `site` argument, returns a list of all the tests doses that passed by that site~~
        - ~~The order of the list matters~~
            - ~~If an operation changes many tests at once, the order between those tests does not matter.~~
            - ~~However, all those tests should be on the list after preceding operations and before later operations~~

Besides the `Lab` interface, your solution should also implement the `DiagnosticTest` interface for each individual dose:

```
interface DiagnosticTest {
    enum Status { READY, SAMPLED , POSITIVE, NEGATIVE , INVALID}
    Status getStatus();
}
```

Each test should behave as follows:

- All tests are created as `READY`
- A `READY` test can take a sample, in which case it becomes `SAMPLED`

- Once a test is sampled, it cannot become READY again or be used again
- A SAMPLED test can become POSITIVE, NEGATIVE, or INVALID
- Once a test has a result, it cannot be sampled again, and the result cannot change

Correctness Requirements

Your implementation should keep the following properties at all times:

1. **getDiagnosticTests** operations never list more doses than a site's capacity
2. **getDiagnosticTests** operations never list more items for the whole lab than the sum of the capacity of all the sites.
3. Adding sampled tests to a site successfully results in those tests being listed in later **getDiagnosticTests** operations.
4. Once a test is positive, negative, or invalid; that test is not listed in later **getDiagnosticTests** operations.
5. ~~Each test is listed in one site at most by **getDiagnosticTests** operations.~~
6. ~~Tests are never "in-transit" due to move operations (i.e., **getDiagnosticTests** operations not listing doses removed from the from site and still not added to the to site).~~
7. Once the status of a test is observed to be POSITIVE, NEGATIVE or INVALID, it cannot be observed to be anything else from that point on.
8. ~~The current contents of any site can be explained by following the entries in the audit log, by the order in which they appear in the log.~~
9. ~~The current state and location of any test can be explained by following the entries in the audit log, by the order in which they appear in the log.~~
10. Move operations cannot ever lose any of the tests being moved. Eventually, all the tests will be either in the destination testing site (success) or in the source testing site (fail).

Concurrency Requirements

In this assignment, you are provided with an implementation of `TestingSite` that has the following properties:

- The provided Testing Site already has a set sampledTests to contain all the tests in the site that are sampled
- Each site is associated with its own worker thread
- The set of sampledTests can be modified only by the worker thread of that site
    - If any other thread attempts to add/remove tests from a site, the code throws an exception that will cause all tests to fail
- Asynchronous methods must preserve order: If a thread samples a test on a site and then marks it as positive it using `sampleDiagnosticTestsAsync` followed by `positiveAsync`, then both results should (eventually) be true
- Your implementation should not use busy-waiting

Entry Point

You should create a new class, on a new file, where you will implement your solution. You should change method Lab.createLab so that it creates an instance of the class you added. You cannot change any other part of the code that is provided to you.

```
abstract class Lab {
    static Lab createLab() {
        throw new Error("Not implemented");
    }
}
```

Due Date and Resubmission Policy

This assignment is due on **April 2 2022** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

To resubmit this assignment, your **original grade** (as defined by the autograder) should be **equal to or higher than 30%** for undergraduate students, and **50%** for graduate students. You can resubmit your assignment until **April 9 2022** (following Saturday) at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

Bonus Points

This assignment has a total of **10% bonus points**, which you can earn by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment-4` label, and non-anonymous to the instructors to count towards the bonus.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 70%. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 7 tests, that will check if your submission outputs the expected result. Each test is worth 10%.

Together with the code, you should submit three video screen-cast (**through Gradescope**) that answers the three questions below by explaining how your code works (one video per question). The questions focus on concurrency/multi-threading and are worth 10% each. You can record such a video without installing any software by using the following website: https://screenapp.io/#/

1. How do you avoid busy-waiting in your implementation?
2. Is it correct to modify `TestingSite.sampledTests` without grabbing a lock? Why?
3. How do you ensure that a failing move operation does not result in tests being lost?

The maximum length for each video is 2 minutes, instructors will stop watching at the 2 minute mark (nothing past that point in the video will be graded). This video should be a screencast of your IDE open on the code submitted, and you should highlight the code. Note that longer videos are not better videos, and you should record a video as short as needed to show all the expressions and answer the questions above.

The final grade for the assignment will be the grade of the original submission, for assignments without a resubmission; or the average between the original grade and the resubmission grade, for assignments with a resubmission. The grade of the original submission includes any bonus points.

Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your submission with `System.out.println("expected result").`

The academic integrity policy described in the syllabus applies to this assignment. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.