ASSIGNMENT 3

CS 454: Principles of Concurrent Programming / Spring 2022

Description

In this assignment, you will update your Assignment 1 management utility to track diagnostic test sampling and distribution between a laboratory and many testing sites.

Changes between Assignment 3 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 Set getDiagnosticTests(List<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.
    - With a `List` argument, this operation lists all the tests currently in the sites provided that have been sampled but still don't have a result.
- ~~**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:

Correctness 1. **getDiagnosticTests** operations never list more doses than a site's capacity

Correctness 2. **getDiagnosticTests** operations never list more items for the whole lab than the sum of the capacity of all the sites.

Correctness 3. Adding sampled tests to a site successfully results in those tests being listed in later **getDiagnosticTests** operations.

Correctness 4. Once a test is positive, negative, or invalid; that test is <u>not</u> listed in later **getDiagnosticTests** operations.

Correctness 5. Each test is listed in <u>one site at most</u> by **getDiagnosticTests** operations.

Correctness 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).

Correctness 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.

~~**Correctness 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.~~

~~**Correctness 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.~~

Concurrency Requirements – Progress

Your implementation should allow multiple threads to make progress at the same time, according to the following properties:

Progress 1. **getDiagnosticTests** operations should all make progress in parallel.

Progress 2. The following operations should all make progress in parallel when they are applied to different testing sites: **moveDiagnosticTests, sampleDiagnosticTests**, **positive**, **negative, invalid**.

For instance, consider 3 threads executing at the same time as follows:

1. moveDiagnosticTests from site 1 to site 2
2. positive on site 3
3. sampleDiagnosticTests on site 4

In this example, all 3 threads should make progress at the same time as they operate on different testing sites.

Progress 3. **moveDiagnosticTests, sampleDiagnosticTests**, **positive**, **negative, invalid** and **getDiagnosticTests** operations that do not target the same site should all make progress in parallel.

For instance, consider 3 threads executing at the same time as follows:

1. **sampleDiagnosticTests** to site 1
2. **getDiagnosticTests** on site 2
3. **getDiagnosticTests** on site 3, and 4
4. **getDiagnosticTests** without arguments (on the whole lab, sites 1, 2, 3, and 4)

In this example, thread 1, 2, and 3 should all make progress in parallel because threads 2 and 3 read the contents of testing sites not being changed by thread 1. Thread 4 cannot make progress in parallel with Thread 1, and it is acceptable for only one of these threads (i.e., either 1 or 4) to make progress until the other finishes its operation.

Concurrency Requirements – Correctness

Your implementation should be linearizable. All operations in any execution should appear to take place in a single point. You can use locks to ensure linearizability.

Linearizability 1. When adding multiple tests D1 and D2 to a site C with **sampleDiagnosticTests**, it is not possible for other threads to see D1 added and D2 missing. Other threads calling **getDiagnosticTests** either see all tests D1 and D2 in C, or none.

Linearizability 2. If two threads attempt to add the same test D at the same time to sites C1 and C2, only one thread will succeed. This holds if the sites are different (C1 ≠ C2) or the same (C1 = C2).

Linearizability 3. When moving multiple tests D1 and D2 between sites C1 and C2 with **moveDiagnosticTests**, it is not possible for other threads calling **getDiagnosticTests** to see D1 in C1 and D2 in C2. Either all tests moved are in C1 or C2. This does not affect other tests in C1 or C2 not affected by the move.

Linearizability 4. When marking as positive, negative, or invalid tests D1 and D2 from a clinic with **positive, negative,** or **invalid**, it is not possible for other threads calling **DiagnosticTest.getStatus** to see D1 as SAMPLED and D2 as POSITIVE/NEGATIVE/INVALID. Either all the tests are SAMPLED or all the tests are POSITIVE/NEGATIVE/INVALID. This does not affect other tests in the same site not being marked as positive/negative/invalid.

Linearizability 5. If two threads attempt to mark as positive/negative/invalid the same test D at the same time, only one thread will succeed.

Bonus Extra Functionality

Besides implementing all of the above, you can claim a 10% bonus by implementing **DiagnosticTest.weakGetStatus**:

- **weakGetStatus** performs less work than **getStatus**, and should execute faster. You don't have to show that it executes faster, but you have to explain why you expect it to.
- Property **Correctness 7** holds for **weakGetStatus**
- Property **Linearizability 4** does not hold for **weakGetStatus**, it is possible for other threads to see D1 as SAMPLED and D2 as POSITIVE/NEGATIVE/INVALID (or the other way around).
- **weakGetStatus** does not have data-races.

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 **March 12 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 **March 19 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-3` 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%.

You need to pass Test 3 to get credit for Tests 4, 5, 6, and 7. This requirement prevents submitting an unchanged Assignment 1 and still get points without any effort towards the progress/concurrency requirements explained above.

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. Which operations can result in deadlocks, and what steps did you take to avoid them?
2. How do you ensure concurrent progress on operations that get the contents of the same testing site? (Properties **Progress 1** and **Progress 3**)
3. How do you ensure linearizability when sampling, moving, and marking tests as positive/negative/invalid? (Properties **Linearizability 1**, **Linearizability 3**, and **Linearizability 4**)
4. (Bonus) How did you ensure that **weakGetStatus** performs as described in this document?

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.