CS2103/T AY2425S1 – Exam Feedback

Given below are some general exam feedbacks, **based on questions that had a high percentage of incorrect answers**.

Errata

1. **Part 1 question on 'Sequence Diagrams Notation Errors'** (the one in which you were asked to pick the cells with notation errors):
   The highlighted cells didn't match the answer options and some cells with errors did not appear among answer options. This was a result of using the wrong image.
   This may have caused confusion, extra stress, and may even have slowed you down, affecting other questions as well.
   We'll be giving double marks for the correct answer (and partially correct answers), to compensate for the extra effort it required. However, there will be no loss of marks if you skipped the question or gave the wrong answer (i.e., the question will be treated as a 'bonus question').

[Requirements] iP was greenfield

While you were given a project template to start with (including a trivial amount of code), your iP was essentially a *greenfield* project that created a new product from scratch, unlike the tP that evolved an existing product (i.e., *brownfield*).

[Design] Correct Implementation of the Singleton Pattern

A correct implementation of the Singlelton Pattern requires the following things:

- the constructor should be `priviate`.
- the singleton instance should be `static`.
- the *accessor* method (e.g., `getInstance`) should `static` too, and should reuse the existing singleton object (if any) instead of creating a new one.

[Design] Correct Implementation of the *layered architecture*

A correct application of the *n-tier* (aka *layered*) architecture style requires that lower layers are not dependent on upper layers (i.e., lower layers should not access upper layers).

[Implementation] Expecting the caller to check something first does not always mean we should use assertions to verify it

Normally, if something should have been checked before calling a method, we use an assertion to verify it. In the code below, it is OK to use an assertion if the caller should have checked if `data` is `null` before calling `process()`.

```
void process(data Data) {
    assert data != null : "caller should have checked if data is null";
    // ...
}
```

However, for things that can change any time, checking first doesn't guarantee that the result of the check will be the same afterward. For such cases, we should use exceptions instead of assertions. In the example below, the server may be available when the caller checked it, but it may be unavailable when this method is called (as a server can go down any time), causing the assertion to fail, and the program to crash.

```
void connectToRemoteServer(server Server) {
    assert isServerAvailble(server) : "caller should have checked if the server is availabl
e";
    // ...
}
```

So, it is more appropriate to use an exception (example shown below) rather than an assertion (as in the code above) for such cases.

```
void connectToRemoteServer(server Server) {
    try {
        // ...
    } catch (ServerUnavailableException e) {
        // handle it
    }
}
```

Note: We discussed a similar example in one of the weekly briefings.

[Project Management] iP/tP used Gradle for dependency management

Gradle is both a build tool and a dependency management tool. It is used to manage third party dependencies (e.g., JUnit, JavaFX) in both iP and tP.

[Project Management] iP (and tP) was breadth-first

Both iP and tP were done in breadth-first fashion, each iteration creating a fully working product.

[Project Management] The course doesn't say iterative is better than waterfall etc.

This course doesn't advocate iterative process over the waterfall process, egoless teams over strict-hierarchy teams, breadth-first iterative over depth-first iterative etc. Most these alternatives have pros and cons, and which one is the better choice depends on the context.

That said, the course *does* take the following stances:

- in integration approaches: late one-time big-bang integration is bad; early, frequent, incremental integration is better.
- when planning for unforeseen delays in project scheduling: inflating schedule estimates is bad; setting aside explicit buffers is better.

[Testing] Positive test cases need not be positive numbers

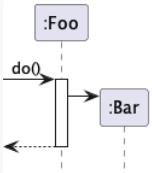
Don't confuse positive/negative test cases with positive/negative numbers. `-1` can be a positive test case while `5` can be a negative test case.

[Testing] Test inputs must be of correct type

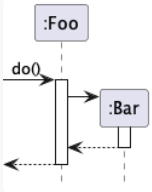
Test inputs must be of the correct data type, if the SUT's programming language is strongly-typed. For example, a Java method `foo(int)` cannot take `null` or `"abc"` as test inputs.

[UML: Sequence Diagrams] Activation bar of constructors can be omitted

The activation bar (and the return arrow) is optional and can be omitted, even for constructors.



The above diagram can be drawn as follows, as well:

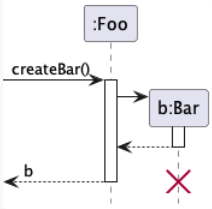


[UML: Sequence Diagrams] Returned objects may not be ready to be garbage collected

If an object created inside a method is returned from the method, we should not indicate it as ready-to-be-garbage-collected, as the object might continue to live after the method has returned.

```
class Foo {
    Bar createBar() {
        return new Bar();
    }
}
```

The diagram below is incorrect (i.e., it should not have the X):



Topic not in the textbook (?)

While concepts below are not specifically mentioned in the textbook, you should have encountered them during the iP and/or the tP, and hence, in the scope of the exam.

- 'Fat' JAR files
- Smoke testing

The following are mentioned in the textbook, although a few mentioned they are not:

- UML *stereotypes* (for an example, see the explanation of the Singleton design pattern)