Lab E Review Form

This form was completed by group: 22
The code reviewed was written by group: 1

The code that you are reviewing should be familiar because each group started with the same stub code.

Reminder

- **This document must be submitted as a PDF (not docx). However, if you wish to submit multiple documents, you can submit them as a zip file.**
- **The submitted document must have your group number. For example, if you are in group 14, the document should be named 14.pdf**
- **This document will be shared with the group whose code was reviewed. Do not include any names or student IDs in this document.**
- **You must provide the kind of feedback you would like to receive - helpful, accurate, and appropriate. Failure to do so may affect your lab grade. If you receive feedback which does not meet these expectations, inform the instructor(s).**

Form
Follow the instructions and answer the following questions. Explain your answers, e.g., if the file is named incorrectly, state what it was named and what it should have been named.

You may optionally remove any grey text in your final submission (instructions to the reviewer and additional details, retaining headings and questions) to make your report more legible.

Style

Preparation

- Reference Lesson03 (Programming Style) to answer this section.

Questions

1. Are the lines short enough in length?

   Yes, all the lines included in the lab submission were 80 lines or less.

2. Are the methods short enough for you to comprehend the entire method at once?

   Yes, when looking at the method lengths, they are short enough to understand what the method does and how object variables affect each component of the entire file

3. Are the names of and variables and methods which were not explicitly named in the exercise sensible?

Yes, when considering all the extra variables outside the included ones, they apply to the problem. Such as the methods for adding and deleting, or the compelled boolean flag.

Scope

Preparation

- Refer to Lesson25 (Scope) and Lesson02 (Java Classes) to answer this section.

Questions

4. Review the scope where variables are declared and used within the ToDoList and Task classes. Are there instances where the scope of a variable could be reduced to improve encapsulation or readability? Provide examples if applicable.

In ToDoList.java line 11, the argument-free constructor had no body. Instead, instance variables were declared at instantiation. Its better practice to assign those values in the constructor.

5. Should static be used in this exercise? Why or why not? If yes, where?

No, static should not be used. The lists created are all made with new objects, so there's not a danger of overloading other values.

6. Is static used appropriately?

Yes, static was not used.

7. Considering the immutable nature of String and the boolean value used in the Task class, should any of the fields (e.g., id, title, isCompleted) or local variables within methods be marked as final to indicate they do not change once initialized? How would this affect the code's clarity and error prevention?

Within Task.java, line 32 updateTitle() updates the title, hence it shouldn't be final. isCompleted is a progress checker variable, therefore it is changed and shouldn't be final. There is an argument that id may be final, as once the id is set you may not want it changed, but as is it is fine for id to stay as not final. So, we agree with the implementation of this code.

Documentation

Preparation

- Refer to Lesson01 (Introduction to Java) to answer this section.
- If the Java code is not in a directory structure aligned with the package edu.ucalgary.oop, create the corresponding directory structure and put the code in it. Refer to Lesson16 (Packages) if it is unclear what directory structure is required.

Questions

8. Can you run javadoc from the command-line? If not, what was the error message?

   yes I can run javadoc from the command-line

9. Does the program contain an appropriate header in the comments?

   No, no header was included for any of the classes

10. Is a description of the class provided?

    No description was provided for any of the classes

11. Does each method have a description?

    No description was provided for any of the methods

12. Does each method describe @return and @param if applicable? The description should explain what is expected, not simply name a variable.

    No description was provided for any of the methods

13. Are the description of the class, the descriptions of the method, and the return values and parameters present in the generated documentation? If they are present in the Java file, but cannot be seen in the generated documentation, explain what mistakes were made in writing the Java documentation.

    So there were no descriptions included at all, so when javadoc is ran all you can see are the names of the methods, their parameters and the return values

14. Are comments used to explain the code? Do they make sense?

    There were very few comments that were included in the ToDoList class. The other classes don't have any comments whatsoever. The couple of comments that were provided do make sense though.

Code Review (Reading)

Preparation

The following structures are all acceptable when the code is unzipped. For this example, assume a group number of 15. Only Task.java is shown, but ToDoList.java and IToDoList.java should also be included, at the same level.

- edu/ucalgary/oop/Task.java
- 15/edu/ucalgary/oop/Task.java
- Task.java
- 15/ Task.java

Questions

15. Is the code part of the package edu.ucalgary.oop?

    Yes the code is packaged correctly within the proper edu.ucalgary.ooop folder.

16. Are the files named correctly?

    Yes, all the files are properly named according to the instructed format and descriptions.

17. Does the submission contain only the files which were meant to be included in the exercise?

Yes, the submission included only the files necessary for the program to run. The test file was not included.

18. Is the submission free of extra directories and does it conform to the expected directory structure?

    Yes, the submission that was included does not contain the edu/ucalgary/oop which is acceptable for file submission guidelines.

19. Review the Task class. Does it include all necessary attributes and methods, such as constructors, setters, and getters? Provide examples for missing attributes and methods.

    Yes, the task class includes 2 constructors, one with a description and one without a description. The former can be omitted due to description never being used. All getters and setters are in place for returning the proper values or setting them when called.

20. Compare the implementation of the code to your own group's implementation. Is there anything that your group (reviewer) did more efficiently which you think the other group (reviewed) could learn from?

    The task class submission contained the same implementation as our group. Both implementations for ToDoList are similar, group 1 has shorter code, but it is more complex with index setting, compared to our group using general commands to check and assign, both implementations are done well.

21. Is there anything that the other group (reviewed) did more efficiently than your own group (reviewer) that you learned from?

    Yes, for some of the methods such as complete task, is slightly more efficient, however uses a few more complex index associations such as subtracting within the parameters. I believe that group 1’s implementation provided good insight into understanding different ways to do problems.

22. Review the ToDoList class. Does it correctly implement the IToDoList interface? Does it include the necessary concrete method to accurately maintain the current state of tasks and fulfill the required operations?

    Yes, the class ToDoList implements all interface functions as concrete implementations. Yes, the concrete implementations of each interface method was properly done, and maintains the current state and can edit the task blocks.

23. Evaluate the deep copy mechanism used in the concrete method to maintain the saveState functionality in ToDoList class. Is this the most efficient way to achieve undo functionality, considering memory and performance? Could this be optimized or implemented differently for better efficiency?

The deep copy mechanism was included within the Task class and handled cloning properly. I believe that the deep copy method is efficient for the current task at hand, however depending on whether it is necessary to deep copy, it is possible just creating a basic copy using .copy may work as well. Using super.clone() is a safe way to ensure the deep copy works properly.

24. Review the methods for adding, editing, and deleting tasks for error handling and input validation strategies. Are there checks to ensure the operations are performed on valid Task objects or IDs that exist within the current task list? How can the code be improved to handle invalid inputs?

    There are no checks to ensure the validity of the parameters being passed. The code can be improved by adding a simple IF statement that checks if the input is valid, then throws an exception if the inputs are invalid.

Code Review (Execution)

Preparation

- Take ToDoTest.java from Lab E materials. Modify the tests by changing the expected values and the provided values. For example, in testDeleteTask, you might change the task from “Learn Java” to “Practice testing”. Compile and run the tests. Reference 18B (Testing) on how to compile and run tests.

Questions

25. Include your modified tests for the following methods: testEditTask(), testUndoDeleteTask() and testMultipleUndos(). How does modifying the test file from the one which was provided ensure more thorough testing?

    For taskEditTask(), we changed the expected edit to Learn Python instead of Learn JUnit, this runs and passes all tests, changing only the expected output provides a test fail which can help with debugging the edit task feature.

```
@Test
public void testEditTask() {
    expectedTask = new Task(id:"1", title:"Learn Java");
    toDoList.addTask(expectedTask);
    toDoList.editTask(id:"1", title:"Learn Python", complete:true);
    Task actualEditedTask = toDoList.listTasks().get(index:0);

    assertEquals("Task title should be updated after edit", "Learn Python", actualEditedTask.getTitle());
    assertTrue("Task should be marked as completed after edit", actualEditedTask.isCompleted());
}
```

    For testUndoDeleteTask(), if we delete the 2nd task instead of the first one, it throws the proper error, also, if adding the task to, for example ID4, it and delete and undo, it does not retain functionality, it throws a index out of bounds error. So it is only possible to undo and delete task 1.

```
@Test
public void testUndoDeleteTask() {
    expectedTask = new Task(id:"2", title:"Learn Java");
    toDoList.addTask(expectedTask);
    toDoList.deleteTask(id:"2");
    toDoList.undo();
    actualTasks = toDoList.listTasks();

    assertFalse("Task list should not be empty after undoing delete task", actualTasks.isEmpty());
    assertEquals("Task list should contain the undone task after undoing delete", expectedTask, actualTasks.get(index:0));
}
```

For testMultipleUndos() again, similar to testUndoDeleteTask() is it not possible to edit the task such that the initial bound is not ID1, if we change the ID added to 2 and 3, then delete task 2, it fails and throws a index out of bounds error. This test can be used for testing even more than 2 tests, so it's helping for actually testing the full working program, rather than one functionality.

```
@Test
public void testMultipleUndos() {
    Task firstExpectedTask = new Task(id:"2", title:"Task 1");
    Task secondExpectedTask = new Task(id:"3", title:"Task 2");
    toDoList.addTask(firstExpectedTask);
    toDoList.addTask(secondExpectedTask);
    toDoList.deleteTask(id:"2");
    toDoList.undo(); // Undo delete
    toDoList.undo(); // Undo second add
    actualTasks = toDoList.listTasks();

    assertEquals("After multiple undos, task list should have 1 task", 1, actualTasks.size());
    assertEquals("The remaining task should be the first one added", firstExpectedTask, actualTasks.get(index:0));
}
```

26. Did all the tests pass? If any tests failed, note which tests failed and the error messages. Explain what the error means. Suggest how the authors can approach debugging their code.
    - o You should **not** debug the code.
    - o You do not need to provide advice for debugging each test. If multiple tests are failing and you suspect a common origin, you can provide one piece of advice for all related tests.
    - o If not all tests passed, run the code against the original tests and see if it passed. You may have introduced an error in your modification of the tests.

    Yes, all the tests passed. But there was a very weird message in the debug console:

```
3
[[]]

[[], [Task@fadd2227]]

[Task@fadd2227]

2
[[]]
```

[]

3
[[]]

[[], [Task@798676f]]

[[], [Task@798676f], [Task@798676f, Task@7986b4f]]

[Task@798676f, Task@7986b4f]

[[], [Task@798676f]]

[Task@798676f]

This is possibly due to the fact that the constructor is empty. Also they included ‘description’ as a field, which could be removed.