

Software Quality

Ola 3

Task Service with Comprehensive Testing, Static Code Analysis, and Reviews

By Kristofer, Mads, Michael and Søren

Refine the Application:	2
Test using Equivalence Partitioning and Boundary Value Analysis:	3
Static Code Analysis:	4
Peer Code Review:	5
Group 1:	5
Group 2:	6
Software Review:	7
Reflection:	8
Static code analysis tools:	8
Importance of mocking in unit test:	8
Value of code and software reviews in quality insurance:	8
Influences of Equivalence Partitioning and Boundary Value Analysis:	8

Refine the Application:

We continued to develop our application by defining three different business rules that we then implemented into the application alongside both equivalence partitioning tests and boundary tests for each of them.

The first business rule refers to the title of a task. We decided that the text size should minimum be of length 3 and maximum of 100 characters.

The second business rule is the dateline. Here we decided that the user shouldn't be able to set a dateline into the past and that it would be unrealistic to set a date too far in the future, so we set a maximum of 100 days into the future.

Lastly we have the description that we decided to set a minimum length size of 5 and having no maximum length.

Test using Equivalence Partitioning and Boundary Value Analysis:

Title size:

For the equivalence partition test for the title size we select three numbers from the partition and then create a new string containing only the character "x" with the number of characters being equal to the randomly selected number.

For both equivalence and boundary, the table below shows the partition and rather the selected partition is valid or invalid, as well as what the test expects and the actual result.

Partition:	Valid/Invalid partition	Expected:	Actual:
0..2	Invalid	Error	Error
3..100	Valid	No Error	No Error
101..1000	Invalid	Error	Error

Dateline:

For the dateline we make use of the Date type which allows us to add days and receive a valid date in return. Here we check if the difference in days is positive and below the maximum number of days allowed.

For both equivalence and boundary, the table below shows the partition and rather the selected partition is valid or invalid, as well as what the test expects and the actual result.

Partition:	Valid/Invalid partition	Expected:	Actual:
-10..-1	Invalid	Error	Error
0..100	Valid	No Error	No Error
101..1000	Invalid	Error	Error

Description:

The description business rule is similar to the title size business rule with the difference being that the description doesn't have a maximum limit to the string length.

For both equivalence and boundary, the table below shows the partition and rather the selected partition is valid or invalid, as well as what the test expects and the actual result.

Partition:	Valid/Invalid partition	Expected:	Actual:
0..4	Invalid	Error	Error
5..100	Valid	No Error	No Error
101..1000	Valid	No Error	No Error

Static Code Analysis:

To analyze our codebase, we used ESLint with a ruleset designed to catch various errors and issues in the code. However, since we are using Typescript, and utilizing Typescript's strict type checking, many of the more severe bugs (such as null pointer dereferences) are already caught at compile time. As a result, most of the issues flagged by ESLint were minor code style violations, such as missing semicolons at the end of lines, indentation and spacing. Issues like these definitely do matter, as consistency in code style across the codebase can greatly improve readability, particularly as it grows larger. However, all of these issues would be resolved by simply using libraries such as "prettier", which can automatically format the code and fix code style violations.

That being said, ESLint did catch several instances of three somewhat more severe issues:

1. **Console.log statements:** As developers, we all know that console.log statements are the quick and easy way to debug code during development, however, these statements are not always properly cleaned up afterwards, which can end up cluttering the code and make it more difficult to read. Using ESLint we were able to get a quick overview of all console.log statements in the codebase and remove them accordingly.
2. **Unused variables:** Another common issue that can arise during development is unused variables and these can be difficult to properly maintain, without the use of a tool like ESLint. Unused variables in the code can lead to clutter, making it harder to read and maintain. They may also indicate potential logical errors or incomplete code. Removing them reduces unnecessary memory usage and improves code efficiency. Additionally, keeping your code free of unused variables aligns with best practices, making your codebase cleaner and more readable.
3. **Const vs Let:** ESLint caught some instances of the use of *let* for variables which were not reassigned after initialization. Using *const* instead of *let* for variables that don't change after initialization enhances code clarity and prevents accidental reassignment. It communicates intent, ensuring the variable remains constant, which can reduce bugs. Additionally, it aligns with modern JavaScript best practices, making the code easier to maintain and potentially aiding in performance optimization.


All in all, ESLint did not catch any massive issues or breaking bugs, but this is in large part due to Typescript and the strict type checking, which already catches a great deal of potential issues.

Peer Code Review:

Group 1:

Review Goals:

- Better code quality:

```
const handleAddTask = async () => {  
  try {  
    const newTask: Task = {  
      text: text,  
      isCompleted: false,  
      description: description,  
      ...(deadline && { deadline: deadline }),  
    };  
    await addTask(newTask);  
     setText(""); 3 days ago • ola3start  
    setDeadline("");  
    setDescription("");  
    await fetchTasks();  
  } catch (error) {  
    console.error("Error adding task:", error);  
  }  
}
```

In this code example, the three marked lines of code are used in several different places. Making this a function, for example “clearInputFields”, would reduce the overall amount of lines of code and improve code readability.

- Finding defects:

A defect is found in our test, but only when we run all tests at once. We have separated the tests into different files, based on the subject of the tests. When running each file separately, they all return a successful result, but when running them together, two of the total tests returns an error.

- Learning/Knowledge transfer:

The code within the functions doesn't provide any information, other than the names of other functions used, to convey a greater understanding of the codebase. A comment about the name of the function which includes a small description of what it does and the resulting change in the application, as well as a description on the expected input and how they would be handled.

- Increase sense of mutual responsibility:

Naming conventions and placement of the function match the rest of the project and its purpose. Its purpose is made clear to any other developer looking through the file. It does however lack code comments which could help deepen the understanding and thought process and went behind its original construction and later updates.

- Finding better solutions

```

6     test("Partition 1 Invalid", () => {
7         for (let i = 0; i < 3; i++) {
8             let l = getRandomInt(partition1[0], partition1[1]);
9
10            let testTitle: string = "";
11            for (let j = 0; j < 1; j++) {
12                testTitle += "x";
13            }
14
15            expect(() => {
16                checkAddTaskBoundary(testTitle, 'description', "undefined", false)
17            }).toThrow(Error)
18        }
19    });

```

A potential improvement would have been to test every possible integer within the partition, instead of only testing three random ones (line 8).

Group 2:

Review Goals:

- Better code quality:

```

87     async function handleClick(id: string | undefined): Promise<void> {
88         if (!id) return;
89         try {
90             await deleteTask(id);
91             console.log("Task deleted:", id);
92             await fetchTasks();
93         } catch (error) {
94             console.error("Error deleting task:", error);
95         }
96     }
97
98     async function handleImageClick(
99         id: string | undefined,
100         isCompleted: boolean | undefined
101     ): Promise<void> {
102         if (!id || typeof isCompleted !== "boolean") return;
103         try {
104             await completedTask(id, isCompleted);
105             console.log("Task completed/uncompleted:", id);
106             await fetchTasks();
107         } catch (error) {
108             console.error("Error deleting task:", error);
109         }
110     }

```

These two functions both have very vague names and as a developer it makes it difficult to know exactly what their purpose is. More precise names, such as “deleteTask” and “completeTask” would have been more accurate and descriptive. Also, the catch console log in the second function is a bit off point.

Software Review:

Functional correctness:

The purpose of the project was to make a to-do list manager, where a user can create, edit, delete, mark complete, assign categories and sort the list of tasks based on category. The application is complete and all the above functionality has been achieved.

Non-functional aspects:

Tests have been made across the project to help ensure the code continues to work as intended in the future and since it's a rather small application, no performance issues have been observed.

Adherence to best practices:

The structure of the project is quite clear and easy to get into. The project could benefit from an increased number of comments, but this has not been a priority since all members have been part of developing the project and understanding functions/elements hasn't been a problem.

After review improvements:

Going over what is our application we have a frontend where the user can easily perform the required actions, create, edit, delete, mark as completed, assign categories and sort the lists of tasks based on categories. We can't see any changes that we need to make to the frontend, as well as the code itself for the frontend is up to a standard that all in our group are happy with and can work with later on.

The same goes for our backend which again didn't find any areas that need refactoring.

For both the frontend and backend, we have created unit tests, integration tests, some containing mocking, and for the frontend we have a specification-based test and for the backend we have equivalence and boundary tests, which ensures our code is properly covered.

Overall we have not made any additional improvements to our application after reviewing our software.

Reflection:

Static code analysis tools:

Overall using the analysis tool ESLint didn't have a major effect on our code as it only found minor things that are more into the code standart. For some members of our work group it didn't have any effect as they already had a static code analysis tool active. In the IDE IntelliJ there is a static code analysis program running by default that functions much in the same way as ESLint in that it has rules it helps enforce.

Importance of mocking in unit test:

Mocking in tests is an important part of testing functionality that relies on input from other functions that may not work properly during a unit test. Early in the development process the backend may not have access and/or the functionality to get or set data, and so we mock the response from the database, returning an object containing data or error response status, and that allows us to work independently from other areas of the backend then our current work area.

Value of code and software reviews in quality insurance:

When working on our assigned areas of the application we often end up with a sort of tunnel vision and doing a software review helps us keep an oversight of what is done and what needs more work. It also helps ensure that what we make works currently throughout the entire application. In the same way, code reviews help us ensure what we have made stand up to the standard we have set and helps us refactor when needed.

Influences of Equivalence Partitioning and Boundary Value Analysis:

When making the tests for the dateline, the equivalence and boundary tests helped us understand an error when calculating the difference in days between the dateline and the current date, as well as how to fix it.