Kyle Lund

CS-320

10/12/2023

Project 2

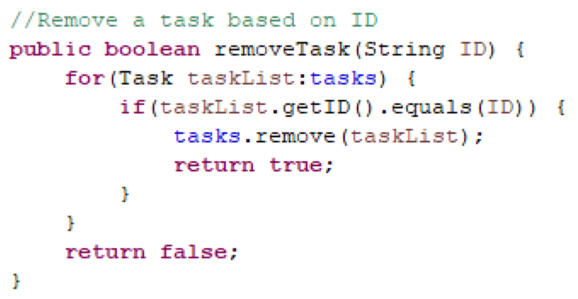
Summary

J-unit Approach

My testing approach aligns with the software requirements as I focus on testing my code to ensure requirements are met. One example of this is in the Task Service class, one requirement is to have the task name and description be updateable by using the task's ID. To ensure the requirements I test my code to make sure by using the task's ID a user can update the task description or name. To further align with the requirements, I also test the character count such that if I am updating the name of the task, I still must be withing the character limit. To ensure the effectiveness of my Junit test I focus on features that need to be tested and then find the coverage percentage of my tests. If the coverage is at least 80% or higher then I know I have tested a good amount of code.

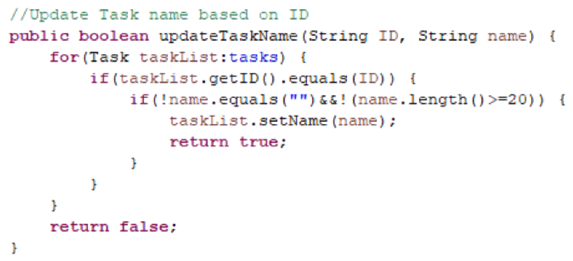
Writing J-unit Tests

Technically sound code is code that is well written, organized, well tested, and bug free. To make my code technically sound I try to make it easy to read without reducing the quality of the code.

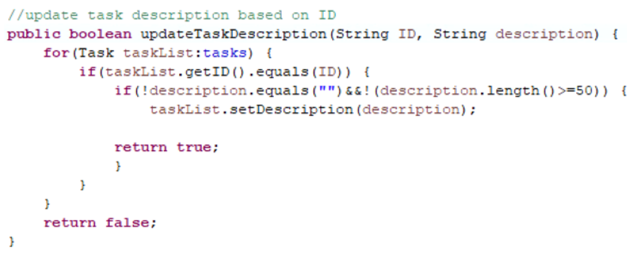


```
//Remove a task based on ID
public boolean removeTask(String ID) {
    for(Task taskList:tasks) {
        if(taskList.getID().equals(ID)) {
            tasks.remove(taskList);
            return true;
        }
    }
    return false;
}
```

The code above is an example, it is easy to read and understand, this code will remove a task based on the ID provided and will check the task list to find a task with the same ID. If the ID matches, then the task is removed, if no ID matches nothing will happen. Efficient code is code that does not have any unnecessary or redundant processes. Efficient code will have high reliability and speed.



```
//Update Task name based on ID
public boolean updateTaskName(String ID, String name) {
    for(Task taskList:tasks) {
        if(taskList.getID().equals(ID)) {
            if(!name.equals("")&&!(name.length()>=20)) {
                taskList.setName(name);
                return true;
            }
        }
    }
    return false;
}
```



```
//update task description based on ID
public boolean updateTaskDescription(String ID, String description) {
    for(Task taskList:tasks) {
        if(taskList.getID().equals(ID)) {
            if(!description.equals("")&&!(description.length()>=50)) {
                taskList.setDescription(description);

            return true;
            }
        }
    }
    return false;
}
```

These two lines of code were created to allow the users to change the tasks name or description. To be efficient, a user can change only the name or description by themselves, without having to update both. The code could have been written in a way that made the user change both at the same time, but with separate changes the efficiency of changing task information increases.

Reflection

Testing Techniques

Most of the techniques I used fall in the black box testing techniques. The first technique I used was equivalence partitioning. This test is to find values and inputs that include valid and invalid values. An example would be testing a phone number that has to be ten characters long, no more and no less. I would test a number that is longer than ten characters as well as less than that to make sure the code works as intended. I also used a technique called boundary value analysis, which goes hand in hand with equivalence partitioning. Boundary testing is essentially testing the boundaries of a value, I have a class that requires an ID that must be within a certain character limit as well as not being Null. The boundary test would test above, below, and at the character limit, this also includes testing both valid and invalid values. The next technique is state transition testing, this tests the reaction of the system to the same input. An example from my work is adding a new task, the code will check to see if the task already exists, then check to make sure it is a valid input, and finally adds the task to the task list.

Other Techniques

A technique that I did not use was decision table testing. Decision table testing will test different combinations of test inputs that will result in different output. A simple example would be a person shopping at a store. If they are a member they get a discount, in this case we would test to see if the customer is a member, then a discount is applied, if not a member, then no discount is applied. Another technique I did not use was system testing, this test the end-to-end points of the system to see how everything works together.

Practical Uses

Equivalence partitioning and boundary value analysis both works together. These techniques are very important in all aspect of development project as they test input values that are both valid and invalid, while creating a software it is important to make sure the inputs work within the required restrictions as well as rejecting invalid inputs properly. Transition testing is also important in development as software may have different branches or test cases. One example could be a developer making an ATM software that accepts cards, the branch could be if card is accepted by bank proceed forward if it’s not with the bank reject the card and end process. System testing for developers is a must as it can be beneficial in all cases. The technique tests the system as a whole and checks to make sure every function is working properly. Testing the system’s functions is good practice and will work with any development project.

Mindset

When it comes to caution in my code, I try to be aware and cautious of the complexity of my code. An example is in the phone number length being only ten characters long, I could make my code complex by restricting the requirements or I could make it simpler by using a length limit instead of limiting the character counts above and below ten. I have tried to limit my bias in testing my code by using JUnit tests and making sure my coverage percentages were acceptable high enough. I know by testing my own code I could be biased but by testing it thoroughly and ensuring my test coverage percentage is high I can ensure I am being little to no biased at all. Being disciplined about code quality is one of the most important things to remember as a good quality code can provide better reputation with clients, better user experience, and save time and money. Cutting corners in code can lead to major problems later down the line that can cost a lot of time and money to fix. An example of this could be a banking company programming a new mobile banking site that a developer did not fully test and cut corners. This could potentially cause major issues in their members' bank accounts as well as vulnerabilities that could cost the company a major amount of money and reputation to fix. If this developer just tested their code and did not cut corners this major problem could have been avoided. I plan to avoid technical debt by keeping my own ego and biases down as much as possible. By doing this I can test and review my code avoiding major issues by not having a hot head that my code and work is flawless when mistakes could be made. Another idea is to have a good team of people that holds everyone accountable, a team that communicates and cooperates with each other can also avoid these disastrous mistakes.

Sources

Types of Black Box Testing Techniques. (n.d.). TestDevLab Blog. https://www.testdevlab.com/blog/types-of-black-box-testing-techniques

What Is White Box Testing | Types & Techniques for Code Coverage | Imperva. (n.d.). Learning Center. https://www.imperva.com/learn/application-security/white-box-testing/

Boni Garcia. (2017). *Mastering Software Testing with JUnit 5 : A Comprehensive, Hands-on Guide on Unit Testing Framework for Java Programming Language*. Packt Publishing.