Chelsea Alysson Ongjoco

Professor Phillips

Software Test Automation QA

20 Oct 2024

Project Two

To what extent was your testing approach aligned to the software requirements? Support your claims with specific evidence.

My testing approach was directly aligned with the software requirements, as each test was specifically made to prove that the software requirements were met.

For example, when given the Task Service requirements:

- The task service shall be able to add tasks with a unique ID.
- The task service shall be able to delete tasks per task ID.
- The task service shall be able to update task fields per task ID. The following fields are updatable:
  - Name
  - Description

In my code, each one was tested:

```
//The task service shall be able to add contacts with a unique ID.
@Test
void testAddContact() {
```

```
// Check to see the number of tasks increased by 1
assertTrue(countAfter - countBefore == 1);

// Check if the added task has all of the correct information
assertTrue(ts.getInfo("12345", "name").equals("Polly"));
assertTrue(ts.getInfo("12345", "description").equals("Insert sample description here"));
```

// the task service shall be able to delete tasks per task ID

```
@Test
void testDeleteTask() {
```

```
// Deletes the task
ts.deleteTask("12345");
countAfter = ts.getCountofTasks();

// Check to see the number of task decreased by 1 after deletion
assertTrue(countAfter - countBefore == -1);


// Should error when trying to update a task that doesn't exist
```

```
// The task service shall be able to update contact fields per contact ID. The following fields are updatable
@Test
void testUpdateContact() {
```

```
// update name
ts.updateTask("123", "name", "Sus
assertTrue(ts.getInfo("123", "nam

// update description
ts.updateTask("123", "description
assertTrue(ts.getInfo("123", "des


// taskID shall not be updatable
Assertions.assertThrows(IllegalAr
```

Defend the overall quality of your JUnit tests for the contact service and task service. In other words, how do you know that your JUnit tests were effective on the basis of coverage percentage?

My JUnit tests had full coverage of all the requirements and thus had full coverage of the code. When testing if the code throws an illegal exception when one of the Strings was too long, I do not just check if it fails if the ID is too long, I also check if the name and description parameters are covered as well.

Same goes for checking if it throws an exception for null… I check all parameters.

```
// Checks if too long

@Test
void testTaskIDTooLong()
    Assertions.assertTh
        new Task("12345
    });
}

@Test
void testNameTooLong() {
    Assertions.assertTh
        new Task("12345
    });
}

@Test
void testDescriptionTool
    Assertions.assertTh
        new Task("12345
    });
}
```

```
// Checks if given null input

@Test
void testTaskIDisNull() {
    Assertions.assertThrows(I
        new Task(null, "Polly
    });
}

@Test
void testNameisNull() {
    Assertions.assertThrows(I
        new Task("12345", nul
    });
}

@Test
void testDescriptionisNull()
    Assertions.assertThrows(I
        new Task("12345", "Po
    });
}
```

How did you ensure that your code was technically sound?

I ensured my code was technically sound by making sure the fields in the tests were accurate to the situations they are supposed to represent.

For example, when testing to make sure the code throws an exception when the test description is too long, I made sure the added object's description field was *actually* longer than 50 characters.

```
@Test
void testDescriptionTooLong() {
    Assertions.assertThrows(IllegalArgumentException.class, () ->{
        new Task("12345", "Polly", "Insert sample description herellllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllll
    });
```

If it was shorter than 50 characters and the test runs, went green and says that it did throw an exception 'as expected', that would be a problem because the code is *not* supposed to throw an exception when it is less than 50 characters.

Same with the opposite scenario, if I put less than 50 characters, and it turns red and says that it did not throw an exception 'as expected', then that would mean the code is working as it should but it looks like the code is faulty because of the faulty test that showed up as red.

How did you ensure that your code was efficient?

I ensured my code was efficient by utilizing methods. For example in the Task class: I made a method that would throw an exception based on the given character limit and if the given String was null.

```
public static void testValidLength(String word, int charLimit) throws IllegalArgumentException {
    if ((word == null || !(word.length() <= charLimit))) throw new IllegalArgumentException ("The " + word + " is null or too long");
}
```

With this, in the constructor I was able to clean up the code by just calling the method "testValidLength".

```
// Constructor of Task
public Task (String taskID, String name, String description) {

    testValidLength(taskID, 10);
    testValidLength(name, 20);
    testValidLength(description, 50);
```

If I didn't make this method and call it, the code (in the constructor) would look similar to:

```
public SchoolClass(String name, String id) {
    if(name == null || name.length()>10) {
        throw new IllegalArgumentException("Invalid name");
    }
    if(id == null || id.length()>10) {
        throw new IllegalArgumentException("Invalid id");
    }
```

Where I would have to manually make these if statements within the constructor, which could easily accumulate and get messy.

Another way I was efficient with my code:

```
// Checks if given null input

@Test
void testTaskIDisNull() {
    Assertions.assertThrows(IllegalArgumentException.class, () ->{
        new Task(null, "Polly", "Insert sample description here");
    });
}
```

There is no other code in the test besides the actual assertion (which is the test).

So the program only focuses on the tests, and not fluff code.

What were the software testing techniques that you employed for each of the milestones? Describe their characteristics using specific details.

I utilized a lot of black box testing for the milestones. Black box testing is testing that focuses on testing the functions of the software (rather than the internal workings of it). Some examples would be: void testAddAppointment(), void testDeleteAppointment()

This is because it focuses on whether the Appointment class fulfills the functional requirements. Black box testing is very helpful for making sure the program fits the requirements, because it directly tests to see if external results are what is expected.

I also implemented some white box testing (which focuses on the internal parts/logic of the code rather than just if the class has the necessary functionality). For example, in that same class, there is the: void testNonUniqueAppointmentID()

Instead of focusing on the functional requirement of adding an appointment, it focuses on the requirement that the function itself cannot accept a non unique ID (in other words, it focuses

on the quality of the function, which is the internal workings of the code, rather than the external behavior of the class (which would be black box testing)).

What are the other software testing techniques that you did not use for the milestones? Describe their characteristics using specific details.

I did not utilize manual informal testing throughout the milestones because I was able to use JUnit to test the code automatically (and with a red/green visual). JUnit was able to save me from the tediousness of manual testing: for example doing a lot of System.out.println() statements, and console analysis to look for potential errors. Manual testing can be tedious in comparison to JUnit when the JUnit page is already made.

For each of the techniques you discussed, explain the practical uses and implication for difference software development projects and situations

For black box testing, the practical use there is to make sure the class completes all its functional requirements. This is very practical because sometimes people don't necessarily need to see the internal workings of the code (or maybe some people lack the technical expertise but still want) to see if it works or meets a certain criteria. With this, the tester can easily check if the class behaves as expected.

For white box testing, the practical use here is to make sure that all the internal structures and workings of the code are up to par. This is helpful to see from a deep level whether the code is of high quality. Looking deeper like this is practical because sometimes it is impossible to test every unique (but nevertheless important) scenario during black box testing, so seeing if the code functions well from the inside 'covers' all the scenarios that are hard to replicate or test when just testing external behaviors.

For informal manual testing, it can be helpful during software development as the developer can test the code relatively quickly, rather than make a separate formal JUnit test in a separate tab. The rapid feedback can be very helpful to find a lot of problems in the code, especially early on in code development.

Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ caution? Why was it important to appreciate the complexity and interrelationships of the code you were testing?

I employed a lot of caution when I was creating my code. Since code will not function as intended unless everything is logically and syntactically sound, it is important that the code is made well. When making code, it is important to be cautious to prevent mistakes in the development process so that there is less need for a back and forth between testing (especially with complex code because this process will be even longer), figuring out the problem and fixing it. The process of fixing problems will be shorter if the problems were not made in the first place or if it is easy to identify the problem.

When making the code I made sure when adding the "IllegalArgumentException" to add a description on why the program threw an exception. This description is very helpful during testing because it gives me more input/information if there is a problem or if the code is working as intended.

```
        }   else {
            throw new IllegalArgumentException ("Invalid input");
        }

    } else {
        throw new IllegalArgumentException ("This contact does not exist");
```

Assess the ways you tried to limit bias in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code?

Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you avoid technical debt as a practical in the field?

According to Merriam-Webster (2019), bias is defined as "an inclination of temperament or outlook" and is a real concern when it comes to testing the code. This is because a developer may have the bias of thinking their solution is right and that may impact the coding. For example, if the coder is overconfident and is biased towards thinking that most of their code is right, they might not make enough tests to cover all of the requirements because they are confident that the software meets the standards even without testing them.

If I looked and saw that the testAppointmentIDisNull was working properly, and started thinking that if any other parameter was null that it would also work properly because I became confident that the function was made correctly, that may be problematic because now there is a possibility that all the tests will run and look fine despite problems in the code (that weren't detected because the tests didn't cover everything). This is the risk one runs if they cut corners in developing tests.

To avoid this and bias in general, I would make sure to test every parameter for the same requirement, even if it is repetitive. I need to be disciplined and make sure I do not rely on my own confidence/bias to see if the code is good because the whole purpose of testing is to find as many errors as early as possible so they can be fixed. This (being cautious and disciplined with coding and testing) also prevents technical debt because the fewer errors there are in the code, the less problems developers will encounter in the future when building upon the base project.

```
// Checks if given null input

@Test
void testAppointmentIDisNull() {
    Assertions.assertThrows(IllegalArgumentExceptio
        new Appointment(null, "12 2 2025", "Insert
    });
}

@Test
void testDateisNull() {
    Assertions.assertThrows(IllegalArgumentExceptio
        new Appointment("12345", null, "Insert samp
    });
}

@Test
void testDescriptionisNull() {
    Assertions.assertThrows(IllegalArgumentExceptio
        new Appointment("12345", "Polly", null);
    });
}
```

References

Merriam-Webster. (2019). *Definition of bias*. Merriam-Webster. https://www.merriam-webster.com/dictionary/bias