Gunnar Johnson

CS-320

Robert Tuft

6/21/2024

Summary and Reflections Report: A Journey Through Software Testing

Section 1. **Introduction**

This report reflects on my experience developing and testing a multi-feature software system, encompassing Contact Management, Appointment Management, and Task Management. Through this project, I've gained invaluable insights into the importance of rigorous testing methodologies and their impact on software quality. In this report, we will explore how these testing methods were applied to my own software with examples from my code as well as the impact testing had for me on a conceptual level.

Section 2. **Testing Approach and Alignment with Requirements**

My testing strategy centered on comprehensive unit testing, closely aligned with the software requirements. For the Contact Management, Appointment Management, and Task Management features, I implemented a comprehensive unit testing approach. I created separate test classes (ContactTest, ContactServiceTest, AppointmentTest, AppointmentServiceTest, TaskTest, and TaskServiceTest) for each feature. This separation allowed me to focus on testing individual components thoroughly while also ensuring that the service layer functioned correctly.

For each feature, I started by testing the basic attributes and methods of the main class (Contact, Appointment, Task). I then moved on to testing the service layer, which included operations like adding, updating, and deleting entries. This bottom-up approach helped me ensure that the foundational elements were solid before testing more complex operations.

2.1 *Unit Testing Strategy*

I adopted a bottom-up approach, starting with basic attribute and method tests before progressing to more complex service layer operations. This methodology ensured a solid foundation before tackling intricate functionalities and introducing complexity to my code. I utilized a multitude of overarching, broad strategies in my approach which I will cover and address. These strategies were the framework of my projects for this course and without applying them, no internal progress could have occurred for me as a developer.

Example #1 from ContactTest class:

```
@Test
void testcontactIdTOOLONG() {

    //initialize a contact object
    Contact contact = new Contact();

    Assertions.assertThrows(IllegalArgumentException.class, ()-> {
        //set contactId to longer than 10 characters
        contact.setContactId("Testing12345");
    });
}
```

This test case directly addresses the requirement that contact IDs must not exceed 10 characters, demonstrating the close alignment between tests and specifications.

2.2. *Comprehensive Coverage*

Built in coverage tools such as the ones in Junit allow for measurable insight on how well your code is functioning, how comprehensive your test cases are, and provide an overhead view of your software that provides deep insights into your work that are difficult to get without the use of outside tools. Using inbuilt coverage tools with Junit allowed me a deep understanding of my code that I was simply not used to and forced me to reflect on how I develop good software.

Regardless of whether I chose to employ a specific coverage tool, I ensured test effectiveness through:

1. Testing all public methods
2. Incorporating both positive and negative scenarios
3. Addressing edge cases and boundary conditions

Example #2 from AppointmentTest class:

```
//tests setter for appointment date

@Test
void testsetAppointmentDate() {
    Appointment appt = new Appointment();
    appt.setAppointmentDate("11-22-2042 03:15");
    assertTrue(appt.getAppointmentDate().equals("11-22-2042 03:15"));

}
```

Example #3 from AppointmentTest class:

```
//tests setting invalid date and checks that the method throws the correct exception

@Test
void testappointmentDateInvalid() {
    Appointment appt = new Appointment();

    Assertions.assertThrows(DateTimeParseException.class, ()-> {
        appt.setAppointmentDate("Abcdefgl234");
    });


}
```

These two tests, Examples #2 and #3, demonstrate coverage of both valid and invalid inputs, ensuring robust functionality. This ensured meeting technical standards for what is good software by following industry standards, good practices, and many other programming concepts outside the scope of this reflection. Thorough testing and being committed to writing good code means that efficiency occurred naturally, but efficiency of any software can be proven through testing and runtime analysis when it is needed. These concepts can be seen in the examples I have provided.

Section 3. **Reflection on Testing Techniques**

3.1 *Employed Testing Techniques*

Throughout this project, I utilized several key testing techniques throughout every single phase in the software development cycle that I was in:

1. Unit Testing
2. Automated Testing (via JUnit)

3. Acceptance Testing

4. Regression Testing

5. Continuous Integration (CI) as a concept and a mindset

Each technique played a crucial role in ensuring software quality. Unit testing formed the backbone of my strategy, while automated testing via JUnit allowed for consistent and repeatable test execution. Acceptance testing ensured alignment with requirements, and my CI approach caught issues early in the development process. Finding and eliminating issues early in the development process enabled me to build from a solid base layer of code allowing me to introduce complexity as I saw fit and with relative ease.

3.2 *Testing Techniques Not Employed*

While my testing was comprehensive for this project's scope, I recognize the importance of additional techniques for larger-scale applications:

1. Security Testing

2. Performance Testing

3. Usability Testing

4. Formal Integration Testing

In future projects, especially those dealing with sensitive data or expecting high traffic, incorporating these techniques will be crucial for ensuring robust, scalable, and user-friendly software. I will carry the concepts I worked with for the entirety of my career thanks to my deep experience with testing during this course.

Section 4. **Mindset and Approach to Testing**

4.1 *Cautious and Methodical Mindset*

I approached this project with a cautious, methodical mindset, recognizing that thorough testing is fundamental to software quality. This caution manifested in rigorous input validation and careful consideration of edge cases. Being cautious is good with any endeavor or challenge we face in life, not just software. Being cautious allows us to slow down and analysis while still taking action without the constraints of anxiety induced inaction that ends up with nothing being done in a project. Caution combined with calculated methodical action allows for productivity to flourish while ensuring quality is produced.

Example #4 from Contact class:

```
//basic checks for adherence to requirements are included in the setters for length and setting a variable to null
public String setContactId(String contactId) {
    this.contactId = contactId;
    if (contactId == null) {
        throw new NullPointerException("Invalid Contact ID...");
    }
    if (contactId.length() > 10) {
        throw new IllegalArgumentException("Invalid Contact ID: Contact ID too long...");
    }

    return null;
}
```

This method demonstrates careful input validation ensuring all requirements are met before setting a contactId.

4.2 *Appreciating Complexity and Interrelationships*

Understanding the interplay between different system components was crucial, especially when implementing operations that affected multiple aspects of the system. By creating a solid foundation for myself, complexity and interrelationships required no extra thought, and I could build from a strong base easily while respecting those two concepts within my software. Embracing the challenges you face with an open mind enables incredible work to be done.

Example #5 from ContactService class:

```
//set the parameter for contactId to final. this ensures it cannot be changed by this method.
//this method for its use in the main branch, simply wont give the user an option to change contactId if it set.
//this ensures immutability to my understanding.
public Contact UpdateContact(final String contactId, String firstName, String lastName, String number, String address) {

    //call the getContact method so if you attempt to update a nonexistent contact, it will tell you.
    //initialize contact object to contact returned by GetContact.
    Contact contact = GetContact(contactId);

    //gets the index value for the contact being updated by its contact Id
    int index = getContactIndex(contactId);

    //initialize a new object to updated information
    Contact updatedContact = new Contact();

    //sets updatedContact object to the values passed to the method
    updatedContact.contactId = contactId;

    updatedContact.setFirstName(firstName);

    updatedContact.setLastName(lastName);

    updatedContact.setNumber(number);

    updatedContact.setAddress(address);



    //passes the contacts index to the delete method and removes it from the list entirely
    delete(index);

    //immediately replaces the contact that was removed with the new contact information passed to this method.
    set(updatedContact);

    //sets the contact object for this method to the newly added contact so it can be returned
    updatedContact = GetContact(contactId);

    //returns the updated contact object
    return updatedContact;
}
```

This method showcases the balance between maintaining data integrity (immutable contactId) and allowing necessary updates.

4.3 *Limiting Bias in Code Review*

As both developer and tester, I was acutely aware of potential bias. To counteract this, I employed strategies such as comprehensive test cases, separation of concerns, and reliance on automated testing. Separating myself from my work in this way, in my opinion, is necessary for me to create good, valuable software for anyone including myself.

Example #6 from TaskTest class:

```
// task ID too long
@Test
void testTaskIdTooLong() {

    //task object
    Task task = new Task();

    //asserts correct error is throw by this method
    Assertions.assertThrows(IllegalArgumentException.class, ()-> {
        task.setTaskID("OverTenCharacters");

    });
}
```

This test checks an edge case that might be overlooked when focusing solely on typical use scenarios.

Section 5. **Commitment to Quality and Avoiding Technical Debt**

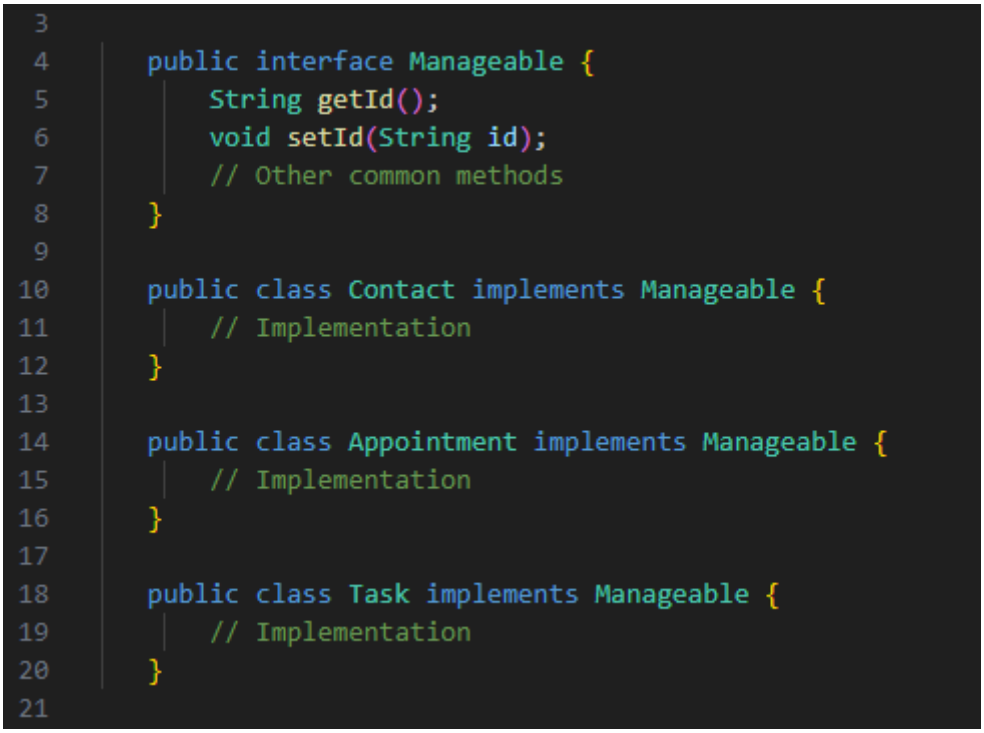
My experience reinforced the critical importance of a disciplined commitment to quality in software engineering. Not cutting corners in writing or testing code is crucial for:

1. Ensuring reliability and building trust with users
2. Improving maintainability for future updates
3. Increasing long-term efficiency by catching bugs early
4. Ensuring reusability and portability
5. Creating valuable software

To avoid technical debt, I plan to:

1. Prioritize testing from the outset of development
2. Regularly refactor code to improve structure and reduce duplication
3. Stay updated with evolving best practices and testing techniques
4. Embrace formal continuous integration tools in future projects
5. Stay organized and practice cautious, considerate coding

Example #6 shows a potential future refactoring I would like to do with my code from this project:



```
3
4       public interface Manageable {
5           String getId();
6           void setId(String id);
7           // Other common methods
8       }
9
10      public class Contact implements Manageable {
11          // Implementation
12      }
13
14      public class Appointment implements Manageable {
15          // Implementation
16      }
17
18      public class Task implements Manageable {
19          // Implementation
20      }
21
```

This refactoring would reduce duplication and improve maintainability across the system, enabling what is now a set of simple, separate programs to become one software that can scale effortlessly in terms of functionality and complexity.

Section 6. **Conclusion**

This project has been a profound learning experience, reinforcing the importance of rigorous testing in software development. By adopting a disciplined, cautious approach and employing a range of testing techniques, I've developed a robust, maintainable system that meets all specified requirements. This experience opened my eyes to the possibilities of software development that creates not only useful code, but consistently reliable tools that individuals or enterprises can use to do good and provide value to the world.

Moving forward, I'm committed to building on this foundation, continuously improving my testing strategies, and embracing new techniques as I tackle more complex projects. This commitment to quality isn't just about personal pride; it's about creating reliable, efficient software that provides real value to users and truly stands the test of time.

As I progress in my career as a software engineer, the lessons learned from this project will serve as a constant reminder of the critical role that thorough testing plays in developing high-quality software. I look forward to applying these insights to future projects, continually refining my skills, and contributing to the creation of robust, reliable software solutions. Let this reflection stand as my commitment to myself to not only produce software in line with my moral values, but

produce software that is effective and reliable, because of the principles behind this commitment, so I can make the world a better place through the things I create.