**Project Two**

**CS 320**

**Chelaine Echols**

**Summary**

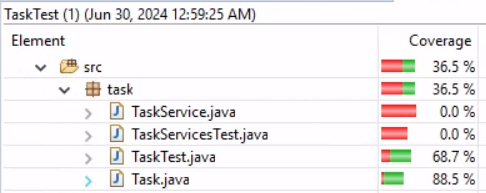
***Unit Testing Approach***

**Software Requirements.** To ensure the programs created aligned with software requirements, I used the requirements as a sort of checklist while writing the JUnit tests. For example, in the ContactService class, one of the listed requirements was: "The contact service shall be able to update contact fields per contact ID" (Module Three Milestone Guidelines and Rubric). The updateable fields were first name, last name, number, and address. To ensure that requirement was fulfilled, I created tests for each of those fields ensuring the fields could be updated or changed by providing the contact ID. The screenshot below shows the first name example.

I went through all of the software requirements and made sure to have tests for each. Additionally, requirements were tested to ensure they would behave as expected if given incomplete or incorrect information. The screenshot below shows the JUnit test for updating an address field; however the contact has not been added to the HashMap and therefore cannot be updated. The correct action would be to throw an IllegalArgumentException, so that is what is A computer code on a black background

Description automatically generatedtested.

**A screenshot of a computer

Description automatically generatedQuality of JUnit Tests.** The quality of my JUnit tests are verified through the test coverage feature. When the JUnit tests are run with coverage, the program will tell you what percentage of code has been run and successfully tested. Industry standard is 80 percent coverage. As you can see in the two screenshots below, the TaskTest had 88.5 percent coverage on the Task class, and the TaskServicesTest had 80.5 percent coverage on the TaskServices class.

Additionally, the remainder of my programs exceeded 80 percent coverage. The Contact class achieved 100 percent coverage, ContactService had 89.4 percent coverage, Appointment class had 87.6 percent coverage, and AppointmentService achieved 100 percent coverage.

***Writing JUnit Tests***

**Technically Sound Code.** To ensure my code was technically sound, I followed industry standards and best practices. I avoided repetition and redundancy, initialized and declared variables, included proper code comments, used proper syntax, and organized **A screen shot of a computer program

Description automatically generated**my code in a proper and readable way. Below are some examples from my code.

**A computer code on a black background

Description automatically generated**

**Efficient Code.** Included in industry standards and best practices is efficient code. That is, code that is not unnecessarily complicated or redundant. The decision to use HashMaps instead of ArrayLists in my programs proved to be incredibly efficient.

A screen shot of a computer code

Description automatically generatedInstead of using an ArrayList and a for loop, where the program would have to loop through each object every time it is searching for something, I implemented HashMaps. I could then use the containsKey() method which is much more efficient and cuts down on unnecessary code.

**Reflection**

***Testing Techniques***

A computer code with many colorful text

Description automatically generated with medium confidence**Software Testing Techniques Used.** While testing the programs created for Modules Three, Four, and Five, several different software testing techniques were implemented. Equivalence Partitioning was used to create JUnit test scenarios. This was done by creating test cases within the valid value set and then additional test cases from outside the valid value boundary. For instance, in the JUnit test for the Contact class entitled testContact, creating a Contact instance was tested by first providing correct input.

A computer code with many colored text

Description automatically generated with medium confidenceThen, incorrect input values were tested. In this case, that included an ID that had too many characters and an ID that was null.

This is to ensure not only that correct values perform correctly, but also that incorrect values perform correctly, which in this case was by throwing an IllegalArgumentException.

A computer code with white text

Description automatically generatedIn addition, Use Case Testing was implemented during the testing process. This tests the potential interactions between system and users. In the assignments for Modules Three, Four, and Five, this would include the user giving the system input in the form of names, IDs, dates, etc. A specific example of this is in the JUnit test for TaskService. A requirement of the program was the ability to change the Task name by providing the ID number and the new name. The screenshot below shows the code created to test that.

This is testing that the name change is successfully made and that the name verification is performed correctly. In this case, the name cannot be null and cannot be more than 10 characters.

A screenshot of a computer

Description automatically generated The final testing technique implemented in this process was Decision and Statement Coverage. This technique checks that a sufficient percentage of statements and decisions are executed and behave as expected. This is executed by using a built-in feature of Eclipse, where the JUnit test is run with coverage. An acceptable level of coverage is 80%. Shown below are the coverage test results for the appointment and appointmentService classes. You can see appointmentTest covered 87.6% of the appointment class, and appointmentServiceTest covered 100% of the appointmentService class.

A screenshot of a computer

Description automatically generated

**Other Software Testing Techniques.** There are many testing techniques that were not used for this project. These include Decision Table Testing, State Transition Testing, and Experience Based Testing. Decision Table Testing includes creating a table showing all possible combinations of conditions and actions. In this case, such a table was not necessary as there was not multiple conditions creating multiple outcomes. Decision Table Testing would be ideal for programs that have complex business rules and multiple input combinations that result in different outcomes.

State Transition Testing is utilized for programs where input conditions trigger system transitions. This was not a necessary testing technique because the programs created do not have transition states. State Transition Testing is useful for systems that have multiple states and transitions, and where the order of events can significantly impact how the program behaves.

Experience Based Testing has no specific structure to it. It is purely based on the software tester and their previous experiences. Examples would be checklists, error guessing, and exploratory testing. I do not have the experience necessary to implement this testing technique. Experience Based Testing would work best for a software tester who has many years of experience with many different programs.

**Practical Uses and Implications.** Test design techniques can be separated into two basic categories: white-box and black-box. White-box tests a code internal structure, while black-box analyzes a programs behavior. Of the techniques I implemented, equivalence partitioning and use case testing are examples of black-box testing, and decision and statement coverage are examples of white-box testing. Both types of testing are crucial in the software development process. For instance, after completing all black-box testing for my project, I can be somewhat confident that the program works as intended. However, it isn't until I run the coverage that I know how much of my code was actually tested and if all decisions were run. A set of JUnit tests can pass, but they are limited to the scenarios the developer wrote. Tests such as decision and statement coverage help to verify how much of the code has actually been tested and if more JUnit test cases need to be implemented.

***Mindset***

**Mindset and Caution.** While working on this project, I adopted the mindset of one step at a time and continuous testing. There were times I became stuck on a small part of code or testing. In these instances, I continued making small changes to the code and tested at every step. For example, while working on the first section of the program, the contact services, I thought my code was correct but my JUnit tests were failing. I could not figure out why the tests were failing, so I went into the code and looked at every single detail. By utilizing the coverage test, I realized the problem was occurring during the adding a contact phase of the code and test. This narrowed the problem down to one method, addContact(). From there, I reached out to the professor for help and he advised me to change from an arrayList to a HashMap. Once I implemented the HashMap, all tests ran successfully. This situation expanded my knowledge and experience in problem solving and debugging code. It also reinforced the concept of interrelationships of code and how one small error can affect an entire program. Because there was an issue when adding a new contact, the program could not complete the other tasks such as changing contact information or deleting contacts.

**Bias.** Bias can be a huge stumbling block for testing code. It is understandable why in the professional field, developers and testers are separate departments and most of the time a developer does not test their own code. It is extremely helpful when testing to see a program from a fresh perspective, and when you test your own code that is impossible. For instance, when I ran into the issues described above with the addContact() method, I could not figure out what the problem could be. When I wrote the code, I was confident it was correct. However, when I asked the professor for help, he knew right away what could be changed. Obviously, he has much more experience and knowledge in this area, but he also didn't write the code. When you look at a program from an outsider's perspective, it is much easier to see what can or should be changed. When testing your own code is necessary, limiting bias is also necessary. This can be done by seeking outside opinions, using clear and specific objectives for program behavior and outcomes, and using data such as coverage tests to ensure testing is sufficient according to industry standards.

**Quality Assurance.** Quality assurance is perhaps one of the most important aspects of software development. After all, if a program does not perform according to expectations, then the program is useless. Client satisfaction, which is a part of quality assurance, is also an integral part of software development. Most often, developers are creating programs for other clients, not themselves. The client's vision is the ultimate end goal. Cutting corners during the development phase will always lead to unhappy clients and poor quality programs. A commitment to outstanding quality from the beginning ensures realization of goals and requirements in the end. Additionally, discipline in development can save time and money for the developers and the clients. Poor quality will lead to more errors, more testing, more fixing, more time, and more money. Ultimately, a commitment to high quality programs as a software engineer is what I look forward to when I enter the professional field. I will never feel comfortable giving a client a finished product unless it can represent my work, and my work will always be of the highest quality.

**References**

Module Three Milestone Guidelines and Rubric - CS-320-11372-M01 Software Test, Automation QA 2024 C-3 (May - Jun). (n.d.). https://learn.snhu.edu/d2l/le/content/1610138/viewContent/32506381/View