Benjamin Gagnon

CS-320

08/20/2022

Summary and Reflections Report.

In order to ensure code functionality and system stability testing was used heavily throughout the design process. Testing was primarily facilitated through the use of JUnit tests which were designed to verify that each component of the system met the provided requirements. In order to ensure that the software classes were operating effectively I employed the use of JUnit test fixtures, namely dummy objects that stored and handled known inputs so that I could ensure test repeatability and reliability. Some examples of these objects include the dummy contact objects in the ContactTest class which were created to hold predetermined information. These objects were also used to check the functions in each of the new classes by having acceptable and unacceptable data passed during testing. These tests were designed to verify that each class could determine acceptable and unacceptable data and that it would only store data that met the criteria expressed in the design requirements. Testing often included strings that were too long and null pointers all of which were rejected by each of the new classes and not stored as valid input. This is extremely important for the Task, Contact, and Appointment classes as these were the primary classes that would be handling user entered information that had a good chance of being unacceptable. For the Service classes care was put into making sure that any object stored within the service could still be accessed through the service without having to access the object directly. This kind of testing was facilitated by initializing dummy objects for the services, editing the information stored in these objects through the service and then comparing the returned data with what is expected.

The tests developed were quite sound as almost all of the code was covered under the tests. Each requirement outlined in the design document was accounted for and verified during testing as well as each function contained within the classes. Each restriction as to the type of data that could be entered was also tested in order to ensure the software requirements were met. Static testing was conducted throughout the entire coding process in order to ensure proper syntax, variable names, readability and efficiency. In order to ensure efficiency I refactored the code to utilize internal function calls instead of repeating segments of code that are present elsewhere in the class. A good example of this is in each of the constructors for the Task, Contact, and Appointment classes as instead of verifying input or assigning data to variables within the object I passed the data to the “:set” functions that already handled this type of validation and assignment. Lines 10 through 26 in the Task class illustrate the calling of internal functions in order to decrease system complexity and size while also improving readability. I did not test efficiency using a JUnit test as manual code review was easier for me to understand as the software had efficiency considered for from the start. During unit testing I also ensured that my code was technically sound by initializing objects in different ways that they may encounter when the system is deployed. A good example of this is line 26 through 43 in the AppointmentTest class as these tests verify that the Appointment class can handle different types of passed input data during date changes. Whether the system passes a date in the form of a string or a date in the form of a LocalDate object the code is sound and operates effectively the same either way.

During this project I relied heavily on Unit testing as well as diligent static testing in the form of manual review and refactoring during creation. Since each of the 3 sections that the software requirements outlined had little interaction with each other during this project, testing could be carried out on each section while the others were being worked on. The main portions that needed to be verified before other components could be worked on were the Task, Appointment and Contact classes as their associated service classes could not be properly tested until each of the base classes were verified and functioned according to the criteria in the design document. The main testing techniques that were utilized involved testing components during initial creation in order to ensure software integrity before moving onto testing the functions of each class. When I first initialized the appointment class I found that there were errors in the date field which caused a runtime error and subsequently caused the editor to stop the program. I found out that this was caused by improper usage of the Date data type that I had not worked with before. In order to mitigate this problem I changed the Date object to LocalDate and implemented some lines to convert entered string values to LocalDate objects. Once the base classes were functioning properly I then could move onto the service classes as testing these classes required me to not only test the functionality contained within the service class, but also that the functionality of each of the base classes remained consistent. This style of integration testing ensured that the service classes and the base classes operated in the desired manner when integrated together.

In this project I didn’t put heavy focus on testing how the class functions operated and only focused on designing tests that verified their results. I could have tested these aspects of the system by utilizing Structure-Based techniques to ensure that elements of the system are operating correctly and efficiently. To verify that everything is operating correctly I could have created a flow chart before implementing my code. This would have allowed me to understand the actions that are taking place within each object and to better understand the interactions inside the program. Manual review, review using test cases, and flow chart mapping all offer ways to ensure that a particular piece of software is meeting the desired level of functionality in almost any project however since I was designing the project from the start I chose to focus on designing automated tests and handle efficiency while each class was being made.

The main area where I employed caution was with the failure states as I found that error handling can often lead to complex issues in other systems I have worked with. In order to accomplish testing of bad data rejections I had to put the classes in situations where a failure or error may occur. The main thing that I tried to do was demonstrate this in each area where a possible error may occur by passing unacceptable information during the test. While doing this I tried to make simple and easily correctable failure situations for the system without passing it extremely deviant data. This allowed me to focus on slowly building the system up to a point where most of the small errors are accounted for. By working slowly and starting from small errors and working my way up I ensured that I could determine where the errors were coming from and how to correct them before moving onto higher level issues. In this system there was not much complexity as most of the classes operated in a similar manner and only the service classes interacted and held objects of another class type so I did not have to worry about very large errors that encompassed a lot of code.

While reviewing my code I also tried to avoid bias by not cherry picking the test data to give me passing results. With this project it was rather simple as there were clearly defined criteria that had to be met in order for the classes to accept the data as valid, so all I had to do was use data that violated these criteria for my tests. In my tests of the Task class I intentionally passed a null as an input for the name setting function which violated one of the criteria so the system did not store the input and printed and error message in the terminal.

Small errors in a system can and will eventually accumulate into larger and more disruptive issues if left unchecked. This is why it is so important to maintain quality throughout development of any piece of software as small mistakes and cut corners can create issues for not only other portions of the software but also the users and owners of the software should these mistakes be exploited. This kind of commitment also applies to tests as an incomplete or improper test could allow fatal flaws to work their way through into a release of the software. Frequently developing good tests for the software and maintaining good practices can help eliminate some of this risk while working as a programmer however it requires constant work and dedication throughout the entirety of development. Seemingly simple mistakes such as improper input sanitation or verification could lead to a system collapse as there may be no way to handle the incompatible or malicious data. In order to prevent this kind of situation from happening while working In the field a constant effort should be put into trying to develop tests with total coverage as well as conducting thorough static testing to ensure readability and safety.

Citations:

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) - 1.6.1 Testing Shows the Presence of Bugs.* BCS The Chartered Institute for IT. Retrieved from   
<https://app.knovel.com/hotlink/pdf/id:kt00UC2GV1/software-testing-an-istqb/testing-shows-presence>