CSS-320

Project Two

Mark Alan Leo Rossmiller

Grand Strand Systems has enlisted my work as a professional software developer and testing engineer. I was given a list of design documents for the code requirements, and given the task of completing the code to meet the specifications, and writing tests which prove to satisfaction that the code I have written is correct and follows the requirements set forth by the company.

All three packages that I worked to implement their designs required two main classes: one class was the base structure needed for encapsulating the data structures and providing methods to peek inside a specific instance of that object, in order to either set or get a certain variable maintained by the package. It required instantiation of an object that meets the requirements set forth in the design documents, and the ability to set variables with valid data, according to the specifications for the variables given to me by the company. This mainly required that fields were non-null and that they could not exceed a maximum string allowance for a particular type of variable. Each package also required that I write code for a class that manages the objects of that particular package. These manager classes handled creation of objects, storing objects in an array with a unique ID, being able to update fields in a particular object, and the ability to destroy an object when it is no longer wanted by the package manager.

I always began my work on a package by first writing code for the base object class. There were three main packages that needed to be coded: Contact, Task, and Appointment. I used the design document to set out the specific requirements for variables used by these main classes. Each class had similar but unique requirements that needed to be handled by my code, so I used the specification to flesh out the class, making sure to abide by specific lengths for variable strings, making sure that if a field was null that the code returned a null object instead of an object which would break the code wherein it was used. In the object manager code, it was important to use a unique identifier for each object being created, and to check for a null return from the object instantiation routine, and not to store a null object in the array that was being maintained by the code. Deletion needed to check if the object identified by the call was in fact part of the array first, before trying to delete a non-existent object.

When I set about testing the code, I again looked at the design documents, and I used the specifications set forth as a basis of what I should be testing for. If a variable needed to be non-null (which was always the case), I tested against both null strings as well as valid strings, according to the specifications. Using Junit assertions, I compared the results of each set method in the main package, that it was returning the proper values for an object which tried to manipulate its data structure. The most complicated of these to work with was the Date reference in the Appointment class, because instead of using a string value, it required me to handle a Java object that manipulates calendar data in a special class. In order to check validity of a date for the Appointment class, I needed to verify that a date was of future origin, and not a time in the past. Using Junit assertions, I checked against values of dates that were set in the past as well as those set in the future, and I required the code to return an error if a datefor an Appointment object was of past origin.

When testing for the object manager classes, I needed to verify that the code handled situations correctly when an ID was either valid or invalid. I also needed to assert that when a deletion occurred for an invalid ID, that the manager code would handle that as an error, and that it would not break the code.

My Junit tests could have used more elaborate coding in order to ensure that things were being checked properly, and to give better insights into what was going on during the tests; but my code was instead kept simple and basic, just doing the necessary work of verifying that the Java classes which were written were in line with the specifications set forth in the design documents. It is impossible to test for every use case, but since what we were doing was white box testing, it is satisfactory enough to test the boundary cases in such a project. So what was done during my testing was mainly to test known scenarios where a certain variable was either inside the specifications, and a valid string or date value; or whether it was outside the specifications, and was an illegal input for the class’ data structure components, as set forth during the software design phase. I intentionally kept my tests simple and to the point, rather than creating extensive possibilities to test against, because exhaustive testing is quite simply very expensive, unmanageable, and not very useful.

When I approached this project, I used a step-based approach. I isolated a specific requirement and wrote the code necessary to realize that design decision for a computational realization. I didn’t use any elaborate, obfuscated, or creative coding during my process, and instead focused on laying down a simple, easy-to-read, and straightforward solution to a particular problem being developed for. The code I wrote is well-organized, and will be easy to maintain for future developers who come after me.

The tests, likewise, are also simple in design. They meet the minimum requirements to prove that the software is technically sound, and meets its design specifications; without clouding the test code with unnecessary or elaborate lines of instructions.

What the code could have used more of is comments that exemplify why a particular choice was made when I was writing the code. Although the code is simple and straightforward as-is, perhaps things could have been made more clear with more extensive, non-executable documentation. Writing in the design specifications that were used during development as in-line comments perhaps would have made it easier for someone coming to the code as a new developer, who might need to alter the code to account for changing design specifications.

Because I was both the developer as well as the software tester during this project, it is possible that some amount of bias could have entered into the development process. I knew this going into the project, so when I tested the code, I tried to use the specifications as a strict guide to write my Junit assertions, and to clear my head of the actual code being tested during this process. Bias during testing for code which is created by the same person is a serious deficiency during the testing process; and for this reason, companies often employ teams which focus either on development or on testing, and not on both at the same time. Since we did not have this luxury, I feel confident that I did my best to keep my biases and opinions of the original code as much apart from the testing process as possible.

Finally, as a software tester it is important to concern one’s self with the quality and reliability of the code under review, and not to cut corners during the testing process. It is one thing to avoid excessive testing, or exhaustive test case scenarios; these do not contribute to the overall quality of the software being developed. But it is important to test against known valid and invalid cases, and to insure that the software being developed handles such scenarios as expected so that when the code goes into production, bugs are minimized, software failures are shored up against, and systems, and possibly lives, are kept safe.