Kathryn McNeil

CS 320 Software Test Automation & QA

Southern New Hampshire University

6/15/2022

I formed tests around the software requirements. Doing this ensured that there was functionality for every requirement. I was able to go through every requirement like a checklist, creating and running tests for each one. I noticed at one point that I had no actual functionality for certain requirements by doing this. For instance, I tried creating a test that would make sure there were no duplicate IDs. It did not work originally because there was no functionality to test if an ID already existed in the hash map. This prompted me to go through and adjust my code so it would better match the requirements.

I know my JUnit tests were effective because they had 84% coverage. It was required to have at least 80% coverage, so my tests met this requirement. I was able to ensure my code was technically sound through these tests. Every functionality I added had a test making sure it worked. For instance, the program had functionality to create a new appointment, contact, and task object. For every single one of these objects there was also a test ensuring the object could be successfully added. If any errors occurred in my tests then that told me something was not right. As mentioned earlier I discovered issues with my ID functionality this way.

The tests also helped me ensure my code was efficient. For instance, looking at test coverage I saw that my setterers “setID”, “setDate” and “setDescription” were never used in Appointment.java or AppointmentService.java. This meant that I had code that was not being used. Not only is this bad practice, but it could also lead to less efficient code. Making sure code is concise and non-repeating will make the best code possible. It will also keep the code efficient.

In this project I statically as well as dynamically tested my code. First I observed all my requirements and then created code to match them. Throughout the process I was constantly checking to make sure I was accounting for all requirements and that my code was correct. Then, I dynamically tested my code by creating JUnit tests and running them. If any issues occurred I would go back and statically test my code, looking for any issues that may have caused a problem. In this way I went back and forth statically and dynamically testing my code until it was to my satisfaction. By doing both I was able to make sure the code ran, was well written, and covered all requirements given.

Although I tested my code statically, I did not test my requirements. This is an important part of program development because if requirements are missing the code will be missing functionality. It is important to make sure all requirements are gathered before stating to write code as it prevents the need for backtracking. If requirements are complete and well thought out then you can create code in good faith that it will provide all the functionality a client wants. If requirements are not statically tested then time may be wasted writing code that has functionality that is not needed or inaccurate. This can slow down timelines as well as cause deadlines to be missed.

Working on this project made me think about my code differently. Previously I would create code and all I cared about was its syntax as well as what it output when I ran it. By running JUnit tests, it got me thinking about every piece of code I wrote. For every functionality I added I was thinking about ways it could go wrong. For instance, every variable had to have a limit on its length to prevent security or memory issues. To prevent errors no variable could have a null value. This is not something I have included in my code consistently in the past. It also got me thinking where I should catch errors. For instance, every variable caught an illegal argument if it went over a certain length or was null. This was another thing I did not consistently add to my code in the past. Working on this project got me thinking about how to code in a way which prevents unexpected errors by accounting for them ahead of time.

Bias can certainly cause issues when checking code. For instance, for all three milestones I originally had it so that an ID was generated for the object every time an object was created. This is a good way to do it because it prevents any two objects from having the same ID as +1 is added to the ID every time. However, this is not actually what the requirements wanted. It took me three milestones to realize this, and I believe this was due to my bias. I assumed how the ID should be created and did not read the requirements close enough. Only when I went to create project one did I realize that there should be functionality testing that no two objects have a duplicate ID. I did not have this functionality because of how I originally coded it, so I went back, adding functionality for users to add their own IDs and to ensure that the same ID was not used twice.

It is important for more than one person to check code so as to prevent issues from being overlooked. The above example was a relatively harmless one as an ID was still being created and there were no duplicates being made. This does not mean that a very large mistake could not be overlooked in the future though. By having more than one person look at a code multiple perspectives are being used. One person may have a good understanding on how one part of the code should be done, while another may have a good idea about a different part of the code. By combing different perspectives, it is much easier to find issues in code. It also opens up room for debate. Perhaps if someone looked at my code and said that the user should be able to add their own ID, then I could refute it and say automatically generating IDs would cause less problems. It could be debated and the decision that is best for the program and users could then be decided on.

It is important to be committed to high quality work as a software engineer so as to prevent the loss of money, privacy, as well as avoid harm to individuals. Depending on the kind of work one is doing, serious damages can be done from coding malpractice. At best someone’s password may be hacked violating their privacy. However, much more serious problems may arise from not testing code. In my discussion post for this week, I discussed how in 2012 Knight Capital Group lost $440 million as well as had to be bought out by another company after old code was mistakenly uploaded to a server. The old code had not been properly tested, and when an engineer installed the code into eight different servers no one checked them after to make sure they were correctly installed. Because of their lack of testing the company went under and a significant amount of money was lost.

Resources:

Dolfing, H. (2019, June 5). Case Study 4: The $440 Million Software Error at Knight Capital. Henrico Dolfing. https://www.henricodolfing.com/2019/06/project-failure-case-study-knight-capital.html

Hackett, O. (2019, July 30). "Top 10 most catastrophic computer failures in history. Listverse. https://listverse.com/2019/07/30/top-10-most-catastrophic-computer-failures-in-history/