Mac Grier

CS 320

6/22/2025

**Project Two: Summary and Reflection**

My approach to designing the application held software requirements as the foundation upon which my systems were built. I first laid out pseudocode of each class and wrote what each class should do. In the Appointment, Task, and Contact classes, their variables had requirements for what could be held in them. Using that requirement, I designed constructors and set functions that would throw an exception if an improper value was submitted. For example, my constructor in appointment.java used the following line in the constructor:

if (description == null||description.length() > 50){

throw new IllegalArgumentException("Invalid description");

}

Then I used those requirements to shape my unit tests as well. If a variable should not be able to be set to null, I tested setting it to null in the constructor and setting it to null in the set method. I did that for task.java using the line:

*assertThrows*(IllegalArgumentException.class, () ->{task.setName(null);}).

At each step in my development process, I used the requirements listed to realign my program with the purpose I was designing towards.

In my unit tests I made certain that the unit tests I created had a high percentage of coverage. In all my tests I maintained an 80% minimum coverage of the class. To do this, I checked the coverage and then noticed things I was not testing with my unit tests and added those parts. Incorporating as much of the base class into my unit tests as possible helped me check for more bugs and helped me better understand my code by fully comprehending the purpose of each function.

In my Task, Appointment, and Contact class unit tests, I began by testing that the constructor worked properly. Once it instantiated an object of that class, it used the get functions to test that the information entered was properly stored inside using assertTrue functions. For example in TaskTest, I used the following test:

public void testTaskClass(){

Task task = new Task("Todd","Three bananas picking themselves");

*assertTrue*(task.getName().equals("Todd"));

*assertTrue*(task.getDescription().equals("Three bananas picking themselves"));

}

Then I tested the constructor and set functions using invalid variables like null. I used assertThrow which tested for the class properly throwing an exception for invalid input. For example, in ContactTest I used the following test for the constructor:

void testAddress(){

*assertThrows*(IllegalArgumentException.class, () ->{

new Contact("1","1","4254254251","1234567890123456789012345678901234567890");

});

*assertThrows*(IllegalArgumentException.class, () ->{

new Contact("1","1","4254254251", null);

});

}

And this test for the setFirstName function:

void testSetFirstName() {

Contact contact = new Contact("Joe", "Mama", "4254254251", "3828 Piermont Drive NM");

*assertThrows*(IllegalArgumentException.class, () ->{

contact.setFirstName(null);

});

*assertThrows*(IllegalArgumentException.class, () ->{

contact.setFirstName("String that is way too long to be a first name");

});

}

The tests I created did not particularly test for efficiency, but they did have an element of it. Having correct code is efficient since if it didn’t work it would be extremely inefficient. My focus on unit testing did give me a basis for creating small functions for testability which ends up being more efficient. In the future if I needed to create unit tests for efficiency, I would create tests that capture the run time of each function and test whether that run time is less than the acceptable amount of time.

The first checks I did on my coding were simple code reviews. This is checking over my code for logical errors or any inconsistent naming conventions or similar things. This was assisted by my IDE using its built-in static testing tool. It underlined many issues that I was able to catch before attempting to unit test the program that were very minor in nature. Later, I would go back and do reviews using the requirements from the assignments to make sure I was capturing the functionality needed.

Unit testing was the form of dynamic testing that I used for these classes. I created the tests, ran them to make sure things were functioning properly, and used coverage to ensure I was testing as much of the class as possible.

There were several tests that I did not use because of the nature of these projects. Integration testing would have been an important set of tests to run if we had an overall program utilizing the different classes I created. It could have involved testing to make sure data was properly transferred between classes and interacting correctly. In a similar vein, I also did not conduct system testing because there was no overall system being created. This would have tested the overall function of the system with regard to the requirements set for it. If these classes had end-users or shareholders I or my team could contact, I could have also used acceptance testing where I could have had them test to see if the system functioned as they expected it to. This would give me feedback which I could use to improve the program further.

My approach to testing was very cautious in nature because testing is the step where critical mistakes are found, and improper testing can result in releasing a program that does not function. I double and triple checked the unit tests I wrote to ensure that they were testing for the right cases, and that they checked for any edge cases that could appear. My caution lead to me adding exception handling to each of my set functions so that a case of improper input could be handled the same way as in the constructor.

public void setFirstName(String newFirstName){

if(newFirstName == null || newFirstName.length() > 10){

throw new IllegalArgumentException("Invalid first name");

}

firstName = newFirstName;

}

Understanding the complexities of my code helped me fight errors like with the above function where I failed to check the newFirstName argument for null and length, and instead checked the existing firstName variable for them.

While I was designing the functionality for the service classes, I found that a list was an easy way to store each object and access them. I understood how the .get() method worked and so I created the getIndex() function to make it easier for the service to find where in the list an object with an ID was.

private static int getIndex(String ID){

for (int i = 0; i < *appointmentList*.size(); i++){

if (*appointmentList*.get(i).getID() == ID){

return i;

}

}

return -1;

}

The creation of this function also helped me create unit tests. It let me find objects I created using their IDs.

To remove bias in my testing, I created rubrics with the requirements for each function I was testing and checked off things that were properly handled. During manual reviews, I made sure that my variable names followed a standard naming procedure disregarding how I would like them to look. I tried to apply consistent criteria for my classes so that I was not biased towards my work being correct.

Another way I helped avoid bias was writing my unit tests before I wrote my code. I didn’t always do this, but when I was creating tests I realized I should have methods that delete an object using an ID. I created the framework of the test and created the function with the goal in mind. This helped me avoid bias towards my preconceived ideas about how a delete function should work. I knew it should use the ID as the identifier so I had that as the passed in argument and that shaped the final delete function.

public static void deleteAppointment(String ID){

*index* = *getIndex*(ID);

*appointmentList*.remove(*index*);

}

Another way I removed bias was to refactor my code after the fact. I first made the system with the requirements in mind and then went back and changed it to be clearer and more efficient. For example, in ContactService I created each of my update variables and delete object functions which used a for loop to find the index of the designated object. After I reviewed the code, I realized how inefficiently I had made the program, and so I created the getIndex() function that I continued to use in the other service classes for the same purpose.

public static void updateFirstName(String updateID, String newFirstName){

*indexNum* = *getIndex*(updateID);

*contactList*.get(*indexNum*).setFirstName(newFirstName);

}

Eliminating bias helped me make my code more concise and efficient.

Technical debt occurs most often when you take shortcuts in development. It is very tempting to skip steps like planning and testing as you go. What I have learned is that skipping these steps leads to it being harder and slower to continue to develop. In the contact class, I did skip some of the planning steps and that led to me passing in a class that did not function properly. My concept in my head worked, but because I didn’t plan it out properly, I ended up having to rewrite it with corrections from my professor.

My commitment to quality lead me to continuously improve on the work I did. With the classes I wrote, I took care to take what I had done in the previous version and improve upon it. There was a temptation to copy and paste much of what I had done in the previous classes I created but I knew if I did that, I would be cutting corners and avoiding designing with my new requirements in mind. There are plenty of parts that I would have included if I was not careful like in the AppointmentService class. I could have easily written a function that allowed for access to the set function of Appointment, but that was not in the requirements and because I paid close attention to those requirements I did not do needless work and created a class that functioned according to the requirements.