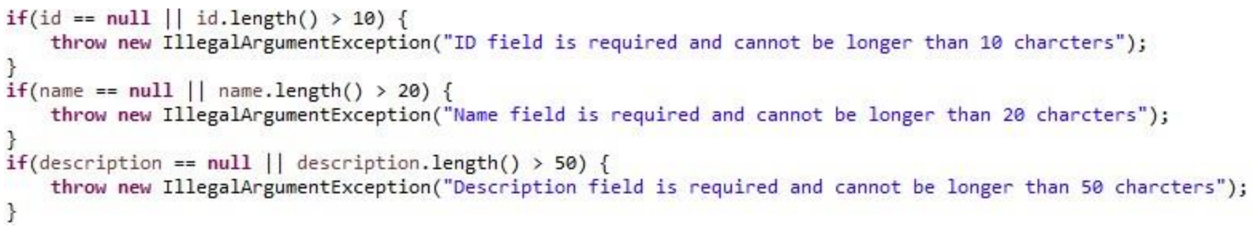
Daniel Gornicz
CS-320-R1877
Project Two

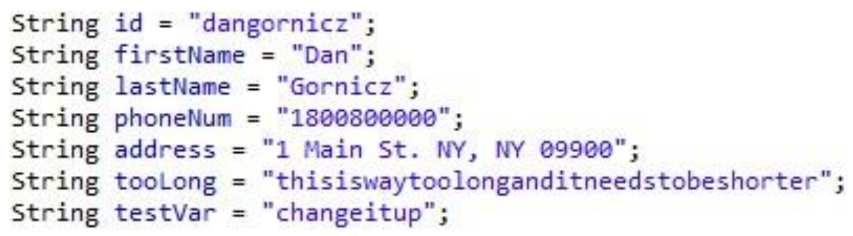
The course project was a very good opportunity to employ the testing techniques that we've been learning about t roughout the semester. In order to ensure that my approach was aligned with the software requirements outlined in all 3 milestones, I used a general approach. While writing each class, I laid out my test objectives, which I wrote based on the requirements and intended (and unintended) functionality. For example, for the Contact object constructor I had to test that all the object was able to be created and then that for each field, the necessary conditions are fulfilled. If a field needed to be a certain number of characters and could not be instantiated null, then I would make sure that I wrote tests to ensure that those conditions were met (See pic below).



```
if(id == null || id.length() > 10) {
    throw new IllegalArgumentException("ID field is required and cannot be longer than 10 charcters");
}
if(name == null || name.length() > 20) {
    throw new IllegalArgumentException("Name field is required and cannot be longer than 20 charcters");
}
if(description == null || description.length() > 50) {
    throw new IllegalArgumentException("Description field is required and cannot be longer than 50 charcters");
}
```

As you can see I made sure to test both positive and negative scenarios, as we learned in our reading. I also tried to keep the tests simple and easy to understand for someone else attempting to decipher my testing files. However, my approach did not end up ensuring that all the requirements were met. Even with thorough review of requirements while writing and ensuring 100% test coverage of the code once it was written, I still missed the requirement that ensured the phone number field in the contact class was only numerical characters (digits). I think in a real-world scenario, working with other developers helps to ensure that these types of oversights don't happen. A review process with multiple people brings different perspectives that can help to identify mistakes or omissions. Even considering this oversight, I definitely believe that my unit tests for both all 3 of the milestones were incredibly effective. I had 100% coverage for my testing throughout these 6 classes (Contact.java , ContactService.java , Task.java, TaskService.java, Appointment.java, and AppointmentService.java). I used the “EclEmma” tool that is provided with Eclipse to assess the coverage of my classes.

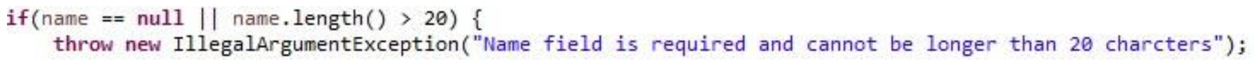
Technically sound code is important to the functionality of your overall program. As I stated before, I made sure to test both positive and negative scenarios (see screenshot from before). I also made sure that the test data I used corresponded with real applications of the software (see pic below).



```
String id = "dangornicz";
String firstName = "Dan";
String lastName = "Gornicz";
String phoneNum = "1800800000";
String address = "1 Main St. NY, NY 09900";
String tooLong = "thisiswaytoolonganditneedstobeshorter";
String testVar = "changeitup";
```

What I mean is that the data looks like a real contact entry. Initializing these variables that are used throughout the class at the top and reusing them throughout the classes and test files also allowed me to write less redundant code. Ensuring that these variables were initialized correctly and used in the correct type when passed to different methods throughout the class also ensured the technical soundness of the code. I think the main factor is creating technically sound code is that everything functions as it should and there are no unexpected scenarios that come up. By this definition I ensured my code was technically sound by making sure none of the runs of my tests failed and by ensuring that I had 100% coverage for all of my classes, and that there was no excess code/variables that were not used anywhere. Everything used in all the classes and test files had a purpose. I also made a point to write comments over certain methods/sections that I felt would benefit from clear explanation. This makes it a lot easier for someone reading your code for the first time to understand the functionality.

In terms of efficiency I tried to ensure that each part of my code was written in a way that prevented unnecessary lines of code. For example the if statements in the constructors (and in the setter methods) checks both the null and character length conditions in one statement (see screenshot below).



```
if(name == null || name.length() > 20) {
    throw new IllegalArgumentException("Name field is required and cannot be longer than 20 charcters");
```

I also used class variables to test with instead of filling parameters, new objects, and other instances with new variables each time (you can see this at the beginning of each class file). Obviously efficiency can also be assessed when using certain loops for searching/sorting, and since I used a Map (more specifically a hash map) I avoided having to search for contacts/tasks with loops that I wrote and implemented. The Map class in java has the built-in method 'containsKey(id)' which allowed me to avoid using these for-loops to search for the 'id' of a contact/task/appointment. I relied on the incredibly efficient hashing algorithm to do the work for me – it is O(1) for operations such as add and much more efficient than a for-loop for searching (using the aforementioned 'containsKey()' method).

I used both static and dynamic testing techniques when working on the 3 milestone assignments. The most obvious type of testing done was running the program using JUnit. This includes assessing the test coverage included via the 'EclEmma' tool. These are both dynamic testing techniques and since these were all small code bases on their own, it was easy to run the JUnit testing as many times as necessary to ensure that the code was working properly. However, if these were larger code bases it would be important to employ static testing techniques throughout the process to find defects early and save time/resources. For example, the IDE gives syntactical errors, type errors, etc which allowed me to correct the bugs and defect before compiling and running at which time I would have run into those errors. I also used static testing when surveying the specifications and determining how to implement and write my code in order to execute these specifications CORRECTLY. I also performed an informal review process (just with myself lol) when I got back the contact milestone and was informed that I needed to pass and object into the add___ method instead of the individual parameters. I had to analyze the ways in which I could change my previous logic to work for the task milestone while implementing the add____ method in the correct way.

I didn't use any type of formal charting or tabling throughout my testing. I didn't create a UML to help me define the way in which I was going to write my code and ensure that specifications were met (static testing technique). I didn't use anything like the decision table testing that we read about, where a table is created to track the all possible input conditions that can occur and their resulting consequences. However if I was working on a project with multiple other people and the different parts of the software became large and complex enough, I could certainly see how using diagrams would help make building seem less daunting and how using decision tables could help keep track of all the moving parts and how the different pieces of code may react when combined.

The JUnit testing and test coverage are certainly the most helpful and practical in terms of their potential applications for larger projects (in my opinion). Knowing that a class/piece of code passed the testing and knowing the coverage of your testing and how comprehensive it is gives you tangible data that results in confidence from other members of your development team, who may not be super familiar with the code you're writing and how exactly you implemented it. However, I think if I were to work on a larger I would want to implement diagramming, which is a type of testing that I didn't use. I think having the specifications and certain implementation information written for the entire team to see I would be super helpful for those situations where someone has to help with a piece of code that they haven't been writing and aren't familiar with. Having documentation/diagrams that lay this information out simplifies the learning process and allows for easier collaboration. Additionally, I believe that type checking/syntax error handling by an IDE is the most useful type of testing. This is something that we all take for granted but software development would be a nightmare and cause compounding issues if we could only check for these types of things at runtime.

I was very confident when writing the first milestone (contact), but because of the oversight I made that I discussed earlier, I had to employ more caution when writing the 2nd two milestones. I also had to make sure that all of the classes interacted with each other as expected. For example, when adding or updating a contact in the ContactService.java class I had to ensure that the constructor and setter methods in Contact.java all made the proper conditions were met for each parameter of the contract class. I had to make this consideration for the task classes and assignment classes as well. Another aspect of employing caution when writing and testing is making sure that your biases don't affect the quality, effectiveness, and functionality of the code. I'm again referencing the 'phone number' type check to illustrate this point. I figured that since I read and wrote that code it was correct and I didn't need to go back and review the requirements because I felt that I hadn't made a mistake. Using multi-person review processes can help to combat these biases. Overall it is very important to ensure the quality of your software. Making sure that everything is written and structured in a way that meets all the functionality and also provides the flexibility to make changes if necessary. I think the easiest way to avoid technical debt is to consider the readability of the code while writing. Making comments and limiting the complexity of the code make it as easy as possible to make changes at any point in the development process. For example, using the Map class in java instead of a bunch of for-loops for updating the different classes and objects not only declutters the code but makes it easier to change up (less code required to change, the easier the change is to make and thus the less technical debt you are accruing).

References:

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing: An ISTQB-BCS certified tester foundation guide - 4th edition*. BCS, The Chartered Institute for IT.

Garcia, B. (2017). Chapter 6: From Requirements to Test Cases. In *Mastering software testing with junit 5*. essay, Packt Publishing.