I made sure that my JUnit tests were effective by being sure that I had tests that checked to be sure that valid inputs, invalid inputs, updates, and that illegal inputs all produce their respective expected outputs. Specifically, I tested for duplicate IDs, strings within character limits, and made sure that dates were only accepted if they were in the future. I also made sure to use strong assertions, to guarantee that invalid inputs were not allowed. These things made sure that the tests had a large coverage percentage. I tried to keep the tests independent, so that just because one test failed it wouldn’t impact any of the others, and the tests would otherwise continue.

I made sure my code was technically sound by making sure that as many things as possible had the expected behavior, made sure that the inputs were properly constrained, and any exceptions were handled. A specific example would be “assertThrows(IllegalArgumentException.class, () -> new Task(null, "Sample Task", "Good Description"));” which tested to be sure that a null taskId wouldn’t be accepted. A second example would be “assertThrows(IllegalArgumentException.class, () -> new Appointment("A12345", null, "Good description"));”, which ensures that an appointment without a date wouldn’t be accepted. I made sure the logic was correct with, for example, this segment, “service.updateTask("12345", "Different Task", "Different Description");  
 Task updatedTask = service.getTask("12345");   
assertEquals("Different Task", updatedTask.getName());   
assertEquals("Different Description", updatedTask.getDescription());” to make sure that, in this case, tasks could update properly. Finally, “assertThrows(IllegalArgumentException.class, () -> task.setDescription("Really long description that exceeds 50 characters and should fail"));” is an example of making sure that the length validation properly works.

Making sure the code was efficient was fairly easy because the project was relatively simple, so keeping redundant operations to a minimum was almost guaranteed, but using HashMaps also kept the lookups for the tasks, appointments, and contacts efficient. My method for deleting tasks “public void deleteTask(String taskId) { if (!tasks.containsKey(taskId)) { throw new IllegalArgumentException("Task ID not used");

} tasks.remove(taskId); }”, for example, would keep any deletions in constant time.

I used unit testing to make sure the individual components of the project worked on their own, so I did things like make tests that would make a mock task, appointment, or contact in order to make sure that those specific systems would function as expected. I also used boundary value tests to make sure that things like strings that were longer than were to be accepted wouldn’t be.Equivalence tests helped to let me be slightly lazier and make fewer tests, but keep the resulting coverage high. I used negative testing to make sure that any bad inputs, like nulls where not accepted or IDs that don’t exist into trying to delete that ID, would properly be handled. I also had tests to be sure that any updated appointment,contact, or task would still function correctly after it had been updated.

I’m not sure there’s a specific term for my mindset when I was working on this project, but as with most times that I do any coding, I try and mentally break down everything into its most basic elements, and ‘simply’ list them. So, for example, a contact is made up of its first name, last name, phone number, and address. In this specific instance, there wasn’t much interaction between the different parts of the project, but typically different parts interacting is important to work around. In most cases, something going wrong in one part of the project can create cascading failures later on, which obviously can create incorrect behaviour, but because of how far removed it is from the actual problem, can be hard to identify the cause of.

I don’t know that I’d say I worked to avoid bias specifically, however I did try to be sure that my tests were as comprehensive as I could make them. I think an example of how I would try to avoid bias in the future, if I were to do the same project, would be to try and comment more in the code. I felt that the relative simplicity of the project made comments pretty superfluous, and just reading the code itself would be just as easy as reading any comment made on it.I didn’t really think about the fact that I may only feel that way because I wrote it until after I had already turned it in, and by then I believe the only comment I left in was a pretty much accidental one, that I just used to be a sort-of bookmark for my eyes while I worked on one of the recommendations you gave. I feel like bias along these lines, just assuming that anyone looking at the code would also immediately understand it, is probably the most common that wouldn’t directly produce any errors or anything to that effect, and so keeping it to a minimum would obviously be ideal.

I like to use an example for why keeping ‘technical debt’ to a minimum is important. A video game I played fairly regularly was pretty well-known within its community for being almost entirely from just about the most pure, raw, undiluted spaghetti-code, where absolutely nothing seemed to be even slightly modular, and any small change the developers made always created not just the normal ‘fix one bug, create two more’ that’s sort of a meme with programming, but they would change something as seemingly minor as a character’s audio emission point, and suddenly the servers stopped functioning completely. I want to avoid anything similar to this because, although it is really, really funny when the extreme examples happen, it can obviously ruin what would otherwise be something great when the novelty wears off.