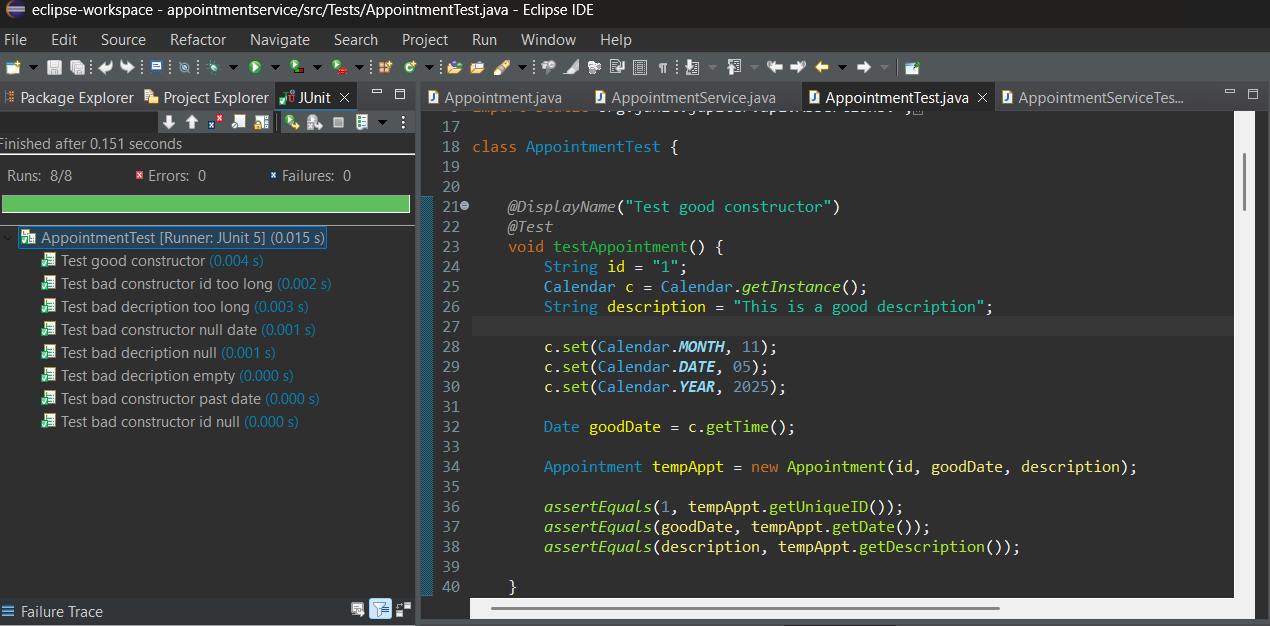
**My unit testing approach for the appointmentservice** feature was to make sure the constructors took appointment ID, appointment date, and an appointment description (reasoning) as the parameters. I then made sure that the program checked if the parameters were null. If the parameters have something within them, then the parameters fields are filled with them. Next, I make sure that appointmentID has a maximum of 10 characters, appointment date to a maximum of 20 characters, and the appointment description having 50 characters. I then set up the getters and setters. I then make sure the program has the right date structure. It is formatted as: Year, Month, Date. Once that is set up, I then make sure that the appointmentlist displays once the program runs. Within the AppointmentService class I create more functions such as a addApointment (adds an appointment), getAppointment (retrieves the appointmentID), deleteAppointment (Deletes the user selected appointment), updateAppointmentDate (Updates the appointment date), and updateAppointmentDesc (Updates the appointment description). Once all of this was set up, I ran my program and checked if my Junit tests were functioning properly and everything was running smoothly. In my program, the Junit tests had a fully completed coverage percentage, meaning everything that I created was working swimmingly. They were effective because the program understood what I asked it to do, based off my functions. **To make sure my code is technically sound**, I ran my program numerous times to make sure everything ran without a hitch. Specifically, I make sure the code in my AppointmentTest.java class is in order. For instance: my testAppointment of 11/05/2025 is used just as an example.   
To make sure my code was efficient, I made an example of an invalid argument. In other words, what the program is to do if the inputted information is invalid. A computer screen shot of a program

AI-generated content may be incorrect.  
As shown in my screenshot, the program even displays “Invalid ID” if there isn’t anything within the appointment ID.   
**Software testing techniques done** on this included functional testing (making sure all the individual classes worked as coded) and non-functional testing (making sure each component was able to load smoothly without any hiccups. This was checked in the Junit window in the screenshots above.) Other software testing techniques that I chose not to do/couldn’t do included Ad-hoc Testing (Since even though there was a deadline for this, I spent plenty of time on it.) and Black Box Testing (Since the code is already known, there isn’t a point tin doing a Black Box Test.)  
  
Functional testing can be applied to all other software projects since in we usually make sure each individual component works as intended, non-functional testing is another no-brainer, making sure everything runs smoothly. Ad-hoc testing can be used to test software right off the bat to see how it runs or if there is something wrong with the software. Black Box Testing can be used to see how a non-developer/the average person would be able to interact with the software.  
  
Whenever I work on any software, I make sure to see if all my code makes sense. I start to be **cautious** whenever I code something that can make or break the software. For instance, making sure that whenever an appointment is scheduled, that users won’t be able to successfully submit appointments with invalid data. It is important to appreciate complexity in coding, since if done well, the code can handle anything the users throw at it. **Bias** can be a cause for concern since if you develop code and someone lets you know of a bug found within it, the knee-jerk reaction would be, “Huh? That doesn’t sound right, I made sure everything was working correctly.” In these situations, I look through everything that can potentially go wrong, even if I’m confident that my code is secure, since not limiting bias can cause software to not be at its best or even worse, not work at all.   
  
**Being disciplined** in software engineering goes a long way since you’ll be able to have strong maintainability in documentation of the software, strong reliability that you’ll follow the best coding standards, as well as reduced technical debt. In other words, since you have strong discipline, you are less likely to make a mistake that is costly, compared to others that aren’t as disciplined. **You don’t want to cut corners** because by doing so, you may end up overlooking something within the code that may cause errors down the line, which can lead to longer development time if the problem is deep within the code, which can then require a complete re-do of the code. I plan to avoid technical debt by making sure I follow the best coding practices, as well as making sure I cover every possible error that may pop up in my software if something goes amiss. For example, rather than having a single function handle every calculation for a system, break the function into smaller parts that handle different things. That way, if a code pops up, it is easier to see where it is coming from compared to just one function not doing a calculation correctly.