

Michael Mathews

08/13/2025

CS320 Project Two

Coverage Tests:

Contact Service-

CoverageContactTest

Element	Class, %	Method, %	Line, %	Branch, %
all	100% (4/4)	100% (29/29)	95% (90/94)	66% (33/50)
ContactService	100% (1/1)	100% (4/4)	100% (17/17)	75% (9/12)
ContactTest	100% (1/1)	100% (6/6)	100% (13/13)	100% (0/0)
ContactServiceTest	100% (1/1)	100% (9/9)	100% (31/31)	100% (0/0)
Contact	100% (1/1)	100% (10/10)	87% (29/33)	63% (24/38)

For the Contact Service all four of my classes are covered by at least one test, so that is 100% coverage in that category. Every method in all of the classes is executed by a test, so that is 100% coverage in that category. In the lines of code ran, my lowest was 29/33 lines, or 87%. My lowest coverage was in my branch statement execution where only two-thirds of the total decisions were covered.

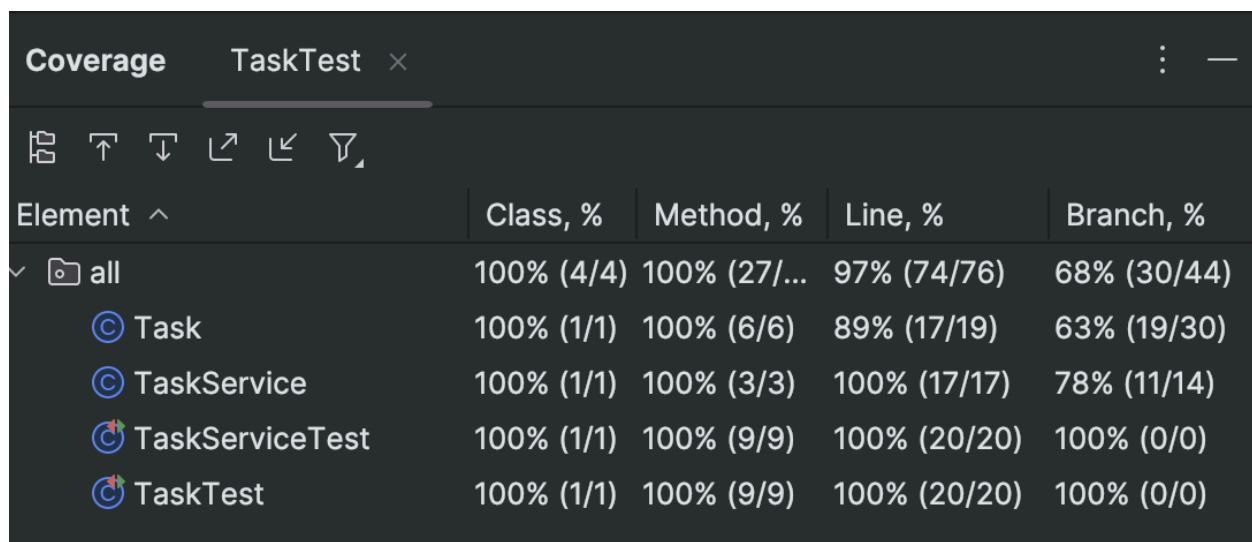
Appointment Service-

CoverageAppointmentTest

Element	Class, %	Method, %	Line, %	Branch, %
all	100% (4/4)	95% (20/21)	87% (61/70)	50% (12/24)
Appointment	100% (1/1)	100% (4/4)	92% (13/14)	71% (10/14)
AppointmentService	100% (1/1)	66% (2/3)	54% (6/11)	20% (2/10)
AppointmentServiceTest	100% (1/1)	100% (4/4)	84% (16/19)	100% (0/0)
AppointmentTest	100% (1/1)	100% (10/10)	100% (26/26)	100% (0/0)

For the Appointment Service all four of my classes are covered by a test, which is 100% coverage. For the method execution three of my classes have 100% execution and the actual AppointmentService class has two-thirds of the methods executed, which could be improved upon. My overall line execution was 61/70 total lines of code run. My decision making branch execution was only 50%, which is definitely something to be explored and added on to get closer to 80-90%.

Task Service-



The screenshot shows a 'Coverage' window for a test named 'TaskTest'. It contains a table with columns for 'Element', 'Class, %', 'Method, %', 'Line, %', and 'Branch, %'. The data is as follows:

Element ^	Class, %	Method, %	Line, %	Branch, %
✓ all	100% (4/4)	100% (27/...)	97% (74/76)	68% (30/44)
Task	100% (1/1)	100% (6/6)	89% (17/19)	63% (19/30)
TaskService	100% (1/1)	100% (3/3)	100% (17/17)	78% (11/14)
TaskServiceTest	100% (1/1)	100% (9/9)	100% (20/20)	100% (0/0)
TaskTest	100% (1/1)	100% (9/9)	100% (20/20)	100% (0/0)

The Task Service has 100% coverage in the class category. Also 100% coverage in the method execution category. For lines run, it has 97% coverage with 74/76 lines of code executed. For the decision branch execution it is at 68% which I would like to see slightly higher with the main culprit being the Task class having only 63% coverage in this category.

Testing Summary-

The requirements set forth were as follows from the instructions-

Contact Class Requirements

- The contact object shall have a required unique contact ID string that cannot be longer than 10 characters. The contact ID shall not be null and shall not be updatable.
- The contact object shall have a required firstName String field that cannot be longer than 10 characters. The firstName field shall not be null.
- The contact object shall have a required lastName String field that cannot be longer than 10 characters. The lastName field shall not be null.
- The contact object shall have a required phone String field that must be exactly 10 digits. The phone field shall not be null.
- The contact object shall have a required address field that must be no longer than 30 characters. The address field shall not be null.

Contact Service Requirements

- The contact service shall be able to add contacts with a unique ID.
- The contact service shall be able to delete contacts per contact ID.
- The contact service shall be able to update contact fields per contact ID. The following fields are updatable:
 - firstName
 - lastName
 - Number
 - Address

Task Class Requirements

- The task object shall have a required unique task ID String that cannot be longer than 10 characters. The task ID shall not be null and shall not be updatable.
- The task object shall have a required name String field that cannot be longer than 20 characters. The name field shall not be null.
- The task object shall have a required description String field that cannot be longer than 50 characters. The description field shall not be null.

Task Service Requirements

- The task service shall be able to add tasks with a unique ID.
- The task service shall be able to delete tasks per task ID.
- The task service shall be able to update task fields per task ID. The following fields are updatable:
 - Name
 - Description

Appointment Class Requirements

- The appointment object shall have a required unique appointment ID string that cannot be longer than 10 characters. The appointment ID shall not be null and shall not be updatable.
- The appointment object shall have a required appointment Date field. The appointment Date field cannot be in the past. The appointment Date field shall not be null.
Note: Use java.util.Date for the appointmentDate field and use before(new Date()) to check if the date is in the past.
- The appointment object shall have a required description String field that cannot be longer than 50 characters. The description field shall not be null.

Appointment Service Requirements

- The appointment service shall be able to add appointments with a unique appointment ID.
- The appointment service shall be able to delete appointments per appointment ID.

Based on these requirements I built tests that covered each of these circumstances for creation, updating, deleting, and improper input. Which specifically aligned with the software requirements as shown above.

I wrote my JUnit tests to cover these specific sets of criteria and they provided almost 100% coverage in most circumstances. The lowest being just over 60% coverage in a category or two.

One of the criteria that I can cite for certain that explains my JUnit test coverage is method percentage. I have almost a 100% method coverage percentage across all projects. This shows that the methods set out in the guidelines were all tested and able to be covered.

Another criteria is line percentage, in which I have 85% or over lines being tested of my code, and in the majority of cases more. For these reasons I can tell that my tests were effective based on coverage percentage.

Technical Soundness-

To make sure that my code was technically sound, one of the main things I did was handle input properly upon initial object creation, and during calls to 'setter' methods for that object if the field was supposed to be updatedable.

For example during the initial object creation I have parameters in place for how the name can be input-

```

public Task(String id, String name, String description){

    if(id != null && id.length() < 10 && !id.isEmpty()) {

        taskID = id;

    } else{

        throw new IllegalArgumentException("Invalid task
ID");

    }

    if(name != null && name.length() < 20 && !name.isEmpty())
{

        taskName = name;

    }else{

        throw new IllegalArgumentException("Invalid task
name");

    }

}

```

When I make a call to the 'setter' I also put those input parameters in place to make sure the input is within the proper boundaries-

```

public void setTaskName(String taskName) {

    if(taskName != null && taskName.length() < 20 &&
!taskName.isEmpty()) {

        this.taskName = taskName;

    }

}

```

```
else {  
    throw new IllegalArgumentException("Invalid task name");  
}  
}
```

I did this for all criteria where applicable in all projects. This created safeguard for when objects were being updated. I also made sure to only allow updates to fields that were supposed to be updateable. Fields that were not updateable only had 'getter' methods, not 'setter' methods.

Another thing that I did to ensure the code would operate as expected was to properly modify the lists. Instead of modifying the collection while iterating over it I used a specified operation in order to properly delete list items, for example-

```
public void deleteAppointment(String id){  
    appointmentList.removeIf(a -> a.getID().equals(id));  
}
```

This ensured no errors occurred within the list.

To make sure the code was efficient I did things like using boolean operators to verify if items were in a list and then only add items if they were in fact in the list, such as-

```
boolean alreadyExists = false;
```

```
for(Appointment a: appointmentList) {  
    if(alreadyExists == false)  
    {  
        if(appointment.getID().equals(a.getID()))  
        {  
            alreadyExists = true;  
        }  
    }  
}  
  
if(!alreadyExists)  
{  
    appointmentList.add(appointment);  
}
```

Reflection:

Techniques-

While writing all of these programs I employed basically the same workflow of building and testing, and then rebuilding. I would write the program, manually test it, correct any errors, manually test it again, and then build the JUnit tests and perform the automated test. After building the initial JUnit test I would construct more edge case tests that I felt would cover the parameters that were set out by the instructions even more thoroughly.

As I mentioned in my last reflection, I programmed 'main' functions and user based input for the first few projects which was helpful for gaining insight into building the software, but counterintuitive when I had to rewrite some of the code in order to fit the automated testing pipeline.

The types of testing that I did not do were functional, end-to-end, acceptance, and performance tests. Functional tests focus on the output of an application. End-to-end tests focus on the function of the application in a completed environment. Acceptance tests focus on verifying that the system satisfies business requirements. Performance testing evaluates the system within the parameters of certain amounts of load

The practical uses of each test are varied but generally speaking each type of testing has its place in some portion of development. Manual testing is great for instant feedback while programming and verifying that everything actually works in the moment, which is great for programmers writing the code directly. Automated tests like JUnit are helpful for creating standards and benchmarks and making sure that no regression is happening during development, like during updates to existing programs. Integration testing will help verify that the software works well in the completed environment such as a newly updated piece of medical equipment that is getting updated software. Acceptance testing is to evaluate if the constructed software performs to user standards and business requirements, such as if it satisfies the user stories. Performance testing is integral to making sure the software can perform within

the environment it is set to run on, such as an embedded application being put on a flip-phone.

Mindset-

When building these projects I employed caution in the realm of making sure that the software met the parameters set forth by the requirements. I wanted what was supposed to be updateable to be able to be updated when necessary and the rest to be left alone. I made sure input was sanitized to the requirements and that updates didn't change that. I used 'getters' and 'setters' to protect input and update class variables correctly.

Any bias that I may have had came from my understanding of what I was supposed to be creating. This built in my mind a sort of testing bias where I was focused on what the outcome was supposed to be. Initially when building the TaskService I created tests that only looked for specific input. After getting feedback I updated the tests to check for null input, improper lengths, and deleting records that do not exist/creating duplicate records.

From a developer's perspective you know how your code works and when testing it you have an inherent understanding of the code's limitations, which you build your tests around. If you were always responsible for testing your own code it limits the diversity in thinking and testing that could take place and variability in input and outcomes.

Performing rigorous testing should include others who are not the developer. One example of this is me writing my set of tests for the TaskService. I wrote tests that included good coverage but didn't account for edge cases on input. Only after getting

outside feedback(from the instructor) did I think to expand the testing outside of the expected input to include more erroneous data.

Being disciplined in your commitment to quality is essential in creating safe and usable products. As a software engineer you interact with the real world through your code, whether that be storing someone's personal data or building software that controls an autonomous vehicle. Everything you create in a professional environment should be built with quality and tested thoroughly in order to not waste time or money, and to keep everyone safe. Planning on the front end can help ensure quality and avoid technical debt. You can plan the tests ahead of time as well in order to create scenarios without having to spend time building software to know what to test. For example if I was building a piece of airline ticket software I wouldn't start by building the software and creating tests for it as we went. We would design the software up front, and create a stage-based testing plan that included the systems it would operate across. Investing time up front reduces technical debt in the future.