Project Two

For all three features my approach was the same. Since I already had some experience with TDD (test driven development), I was able to use the program requirements as a guideline to the tests I needed to implement. From then on I coded the software in a way that passed all the developed tests, which guaranteed a working program in the end.

Since I have used TDD, the approach was directly focused on following the requirements. The most important limitation is in my understanding of what is written as the requirements.

I derived from each requirement presented all the possible business logics that I was able to come up with, that gave me confidence that I was developing great tests.

I used SOLID to ensure quality. The following is an example of that:

```
@BeforeEach
  void setUp() {
    id = "1234567890";
    name = "This is Twenty Chars";
    description = "The task object shall have a required description.";
    tooLongName = "This is way too long to be a task name";
    tooLongDescription =
        "The task object shall have a required description String field that cannot be longer than 50
characters. The description field shall not be null.";
  }
@Test
  void newTaskNameTest() {
    TaskService service = new TaskService();
    service.newTask(name);
    Assertions.assertNotNull(service.getTaskList().get(0).getName());
    Assertions.assertEquals(name, service.getTaskList().get(0).getName());
  }
```

Efficient code must be understandable, and that is something I did. Following the SOLID principles, I was able to ensure that, like in this example:

```
@BeforeEach
  void setUp() {
    id = "1234567890";
    name = "This is Twenty Chars";
    description = "The task object shall have a required description.";
    tooLongName = "This is way too long to be a task name";
    tooLongDescription =
        "The task object shall have a required description String field that cannot be longer than 50 characters. The description field shall not be null.";
  }
```

I have implemented unit testing for the TDD, which is a software testing method by which individual units of source code—sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures—are tested to determine whether they are fit for use.

I have also done some integration testing that is the phase in software testing in which individual software modules are combined and tested as a group. Integration testing is conducted to evaluate the compliance of a system or component with specified functional requirements. It occurs after unit testing and before system testing.

There are many testing techniques that I didn't implement (at least not directly), such as the Black Box Testing, that is a method of software testing that examines the functionality of an application without peering into its internal structures or workings.

Black Box Testing can be applied virtually to every level of software testing: unit, integration, system and acceptance. It is sometimes referred to as specification-based testing.

I was much more aware about the problems that could happen, so I have employed much more caution while developing and testing. That made me better realise how I was able to solve some hard problems in a much better way, such as in how to implement the relationship between tasks and contacts.

Bias is prejudicial, because it fogues the view and judgement of the developer. I, for example, can easily point some errors in a code that I haven't been working for hours, but if it is a code that I spent a lot of time coding, and I am really proud of it, then I will be much less likely of figuring some of the possible errors that the code can generate.

Testing ensures that the final outcome has a quality that was already validated. That saves money and time, and makes software development much cleaner.