CSCI 5308

ADVANCED TOPICS IN SOFTWARE DEVELOPMENT

ASSIGNMENT - 1

Banner ID: B00958098

Original Creator's Github Repository: https://github.com/slabiak/AppointmentScheduler

Forked Github Repository: https://github.com/kapoor98ak/AppointmentScheduler

Table of Contents

Task-1

For this task, we had to choose a Java-based open-source repository that satisfies some conditions. I utilized SEART's [1] github repository search tool to find a repository that would satisfy the following requirements.

The SEART (SoftwarE Analytics Research Team) group is part of the Software Institute at the Università della Svizzera italiana, located in Lugano, Switzerland. They perform research in the area of software engineering, with focus on software analytics, recommender systems for software developers, and empirical software engineering.

Fig 1: SEART's Search Entries to fulfill all the required criteria

Requirements:

> ✅ It must be a maven or gradle-based project
> > ○ The project is a Maven based project (see in Fig 2.)
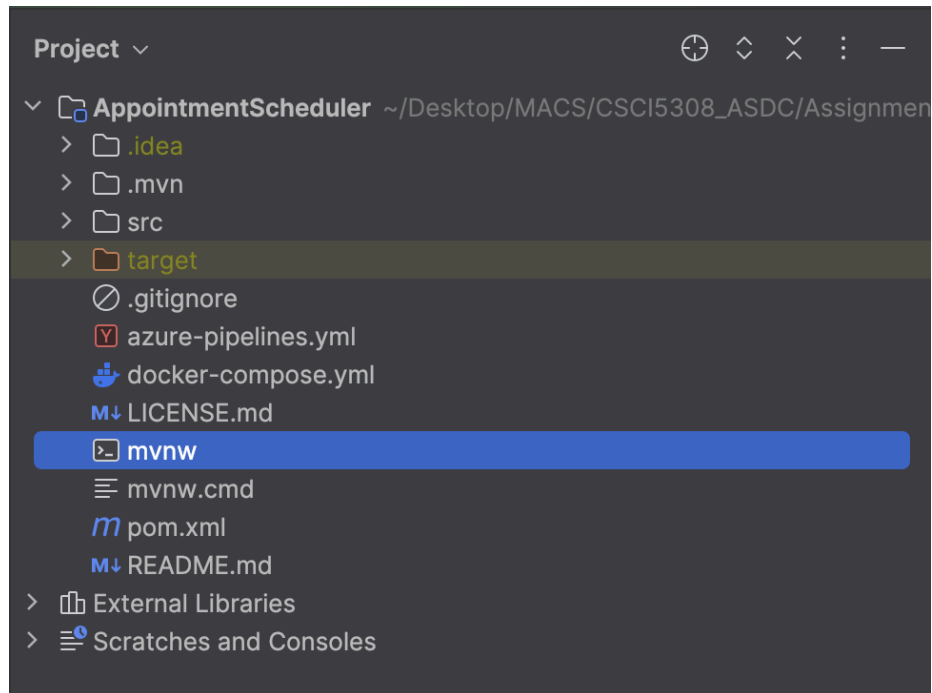
Fig 2: AppointentScheduler is a Maven based project.

✅ It must have at least 10,000 lines of code
  ○ The project has 22,000+ code lines (see in Fig 3.)
✅ It must have at least 50 stars
  ○ The project has 129 stars (see in Fig 3.)
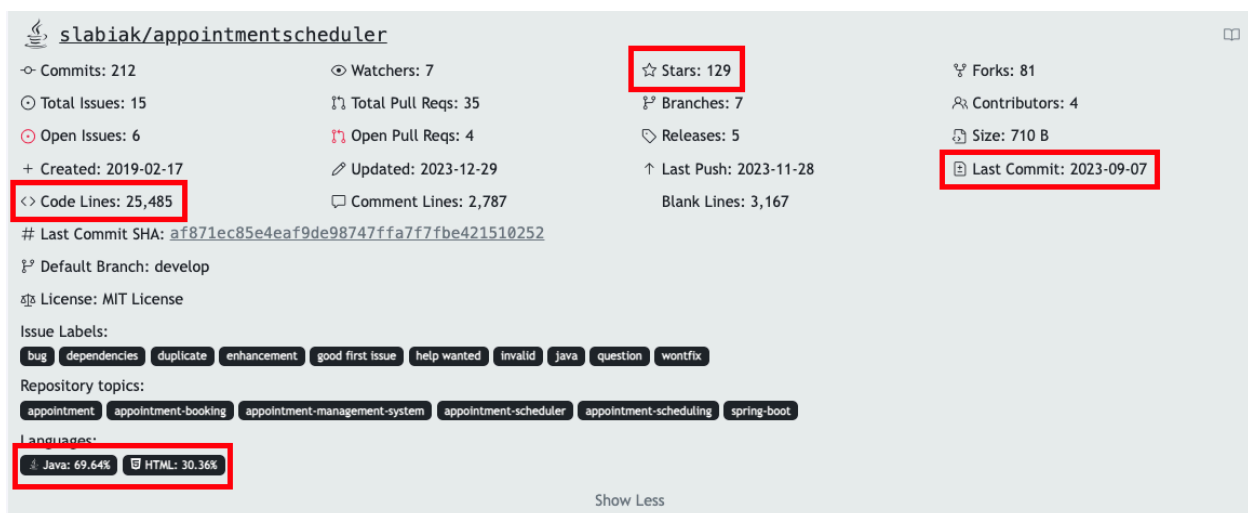
Fig 3: Image showing repository metadata

✅ It must have tests written using the JUnit framework
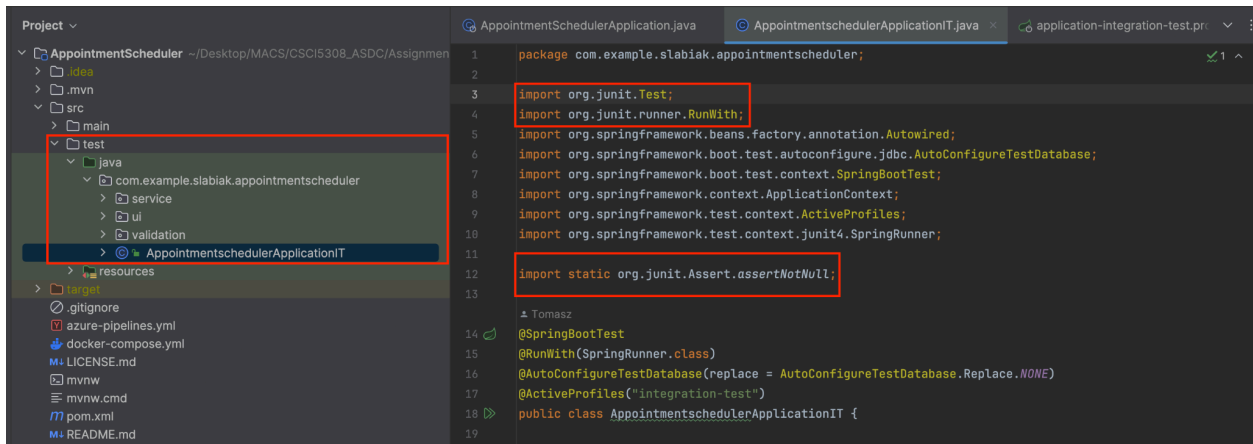
Fig 4: Image showing tests written in JUnit

✅ It must not be a tutorial or example repository
  ○ The Appointment scheduler project is neither an example website or a tutorial.
  ○ **Appointment scheduler:** This is a Spring Boot Web Application to <u>manage and schedule appointments</u> between providers and customers. It has many features such as automatic invoicing, email notifications, appointments cancelation, providers individual working plans with brakes etc.[2]
✅ It must be active (at least one commit in the past one year).
  ○ The project's last commit was in September' 2023 (see in Fig 3.)

Task-2

For this task, we have to provide quantitative measures of test implementation. We can use the total number of automated tests and code coverage (branch) as the quantitative measures.

I referred to this [3] article for developing my understanding on code coverage and I will be citing lines from the same in this task.

Code Coverage

Code coverage describes the percentage of code covered by **automated tests**; it checks which parts of code run during the test suite and which don't.

Just writing Unit Tests is an insufficient metric to count towards the reliability of the program, we need to quantify the usability of the test suite by checking what is the "reach" of our test suite. In other words, *how much of our code base is being reached by our unit tests [3].*

Ways to measure code coverage

1. Statement Coverage

    Statement coverage can be calculated by Number of executed statements / Total number of statements * 100.

    That way of measuring code coverage is able to:
    - verify the do's and don'ts of the written code
    - find dead code and unused statements
    - test different flow paths and checks which ones are not covered

    Statement coverage is typically discussed in the **context of an entire test suite** rather than individual test cases.

2. Branch Coverage

    Branch coverage tells us how many of each decision in a decision-making tree is executed at least once.
    By branches we mean: **conditional statements, loops, switch statements**.

    Branch coverage can be calculated by Number of executed branches / Total number of branches * 100

That way of measuring code coverage is able to:
- verify if the execution of the test suite reaches all branches
- detects possible abnormal behavior of each branch
- can test areas of the source code that other approaches may discount

3. Function Coverage

Function coverage tells us if each function of a program is being called at least once. It is also critical to test functions with a set of input parameters so that the test suites will check if functions behave properly in different scenarios.

Function coverage can be calculated by Number of executed functions / Total number of functions * 100

Function coverage is the broadest metric compared to the rest. High percentage of code coverage doesn't always mean that code is ideal and faultless.

Code Coverage in Appointment Scheduler

Appointment Scheduler's root directory has a test folder.
In there, there are 3 sub-directories.
1. **Service**
   - Here the test classes mainly focus on services for users, work, and appointments.
2. **UI**
   - Here there are tests for the HTML UI.
3. **Validations**
   - Here are the update validations where we check the user information update flow.

Automated Tests

- Initially, 28 automated tests out of 33 automated tests passed and 5 of them failed.
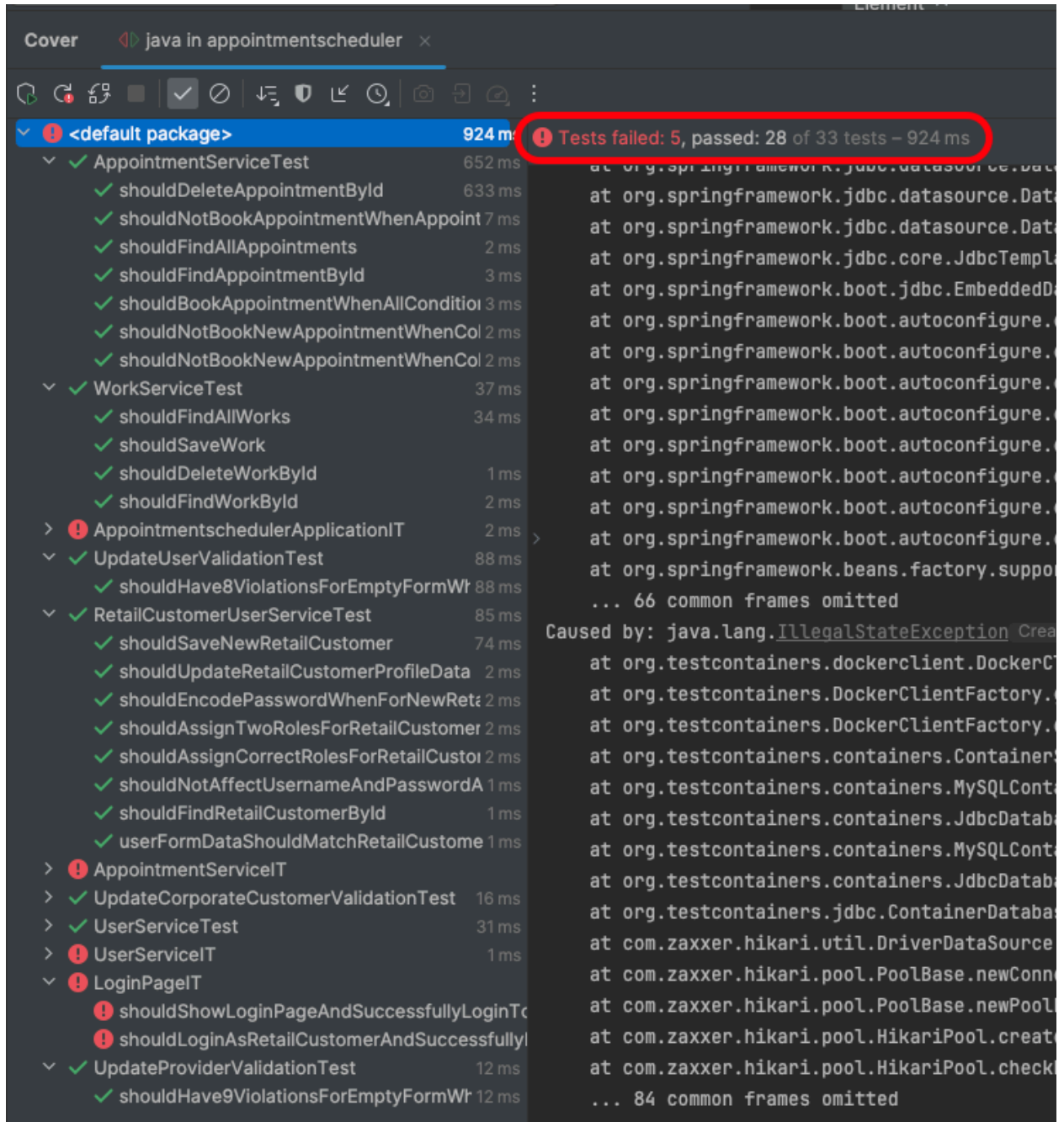- The test cases which were passed and failed are shown in Fig 5.

Fig 5: Image showing 28 out of 33 Automated tests passing

Code Coverage

- For generating the report for the Test Coverage metrics, I used both IntelliJ Coverage plugin[4] and Jacoco dependency[5].
- In Fig 6, we can see the initial test metrics for the Appointment Scheduler.
  We have
    - Only 8.9% Branch Coverage
    - Only 17.8% Line Coverage
    - and only 25% Function or Method Coverage

- Per the reports, the test cases cover
    - 17 classes out of 56 classes.
    - 127 methods out of 500 methods.
    - 252 lines of code out of 1416 lines of code. The
- Per the Fig 6, the Branch coverage can be see as 8%. This report is generated using the IntelliJ's coverage plugin report.

Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Branch, %	Line, %
all classes	30.4% (17/56)	25.4% (127/500)	8.9% (35/394)	17.8% (252/1416)

Coverage Breakdown

Package	Class, %	Method, %	Branch, %	Line, %
com.example.slabiak.appointmentscheduler	0% (0/1)	0% (0/2)		0% (0/2)
com.example.slabiak.appointmentscheduler.config	0% (0/2)	0% (0/4)	0% (0/2)	0% (0/6)
com.example.slabiak.appointmentscheduler.controller	0% (0/9)	0% (0/68)	0% (0/78)	0% (0/291)
com.example.slabiak.appointmentscheduler.entity	50% (5/10)	24.8% (28/113)	11.8% (4/34)	30.1% (50/166)
com.example.slabiak.appointmentscheduler.entity.user	100% (2/2)	84.4% (27/32)	50% (4/8)	83.9% (47/56)
com.example.slabiak.appointmentscheduler.entity.user.customer	66.7% (2/3)	28.6% (4/14)	0% (0/4)	18.2% (4/22)
com.example.slabiak.appointmentscheduler.entity.user.provider	100% (1/1)	23.1% (3/13)	0% (0/16)	9.7% (3/31)
com.example.slabiak.appointmentscheduler.exception	0% (0/3)	0% (0/3)		0% (0/3)
com.example.slabiak.appointmentscheduler.model	66.7% (4/6)	50.7% (38/75)	18.4% (7/38)	39.1% (50/128)
com.example.slabiak.appointmentscheduler.security	0% (0/5)	0% (0/26)	0% (0/12)	0% (0/87)
com.example.slabiak.appointmentscheduler.service.impl	30% (3/10)	19.4% (27/139)	10.9% (20/184)	17.2% (98/569)
com.example.slabiak.appointmentscheduler.util	0% (0/1)	0% (0/2)	0% (0/10)	0% (0/29)
com.example.slabiak.appointmentscheduler.validation	0% (0/3)	0% (0/9)	0% (0/8)	0% (0/26)

generated on 2024-03-03 21:00

Fig 6: Image showing Automated test metrics for Appointment Scheduler

appointmentscheduler

appointmentscheduler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.example.slabiak.appointmentscheduler.service.impl		17%		10%	204	237	526	635	117	144	7	10
com.example.slabiak.appointmentscheduler.controller		0%		0%	108	108	300	300	69	69	9	9
com.example.slabiak.appointmentscheduler.security		0%		0%	34	34	95	95	28	28	5	5
com.example.slabiak.appointmentscheduler.entity		32%		14%	94	121	154	222	84	111	5	10
com.example.slabiak.appointmentscheduler.model		31%		18%	55	94	101	171	37	75	2	6
com.example.slabiak.appointmentscheduler.entity.user.provider		6%		0%	18	21	32	37	10	13	0	1
com.example.slabiak.appointmentscheduler.validation		0%		0%	14	14	27	27	9	9	3	3
com.example.slabiak.appointmentscheduler.util		0%		0%	3	3	29	29	2	2	1	1
com.example.slabiak.appointmentscheduler.entity.user.customer		21%		0%	12	16	24	32	10	14	1	3
com.example.slabiak.appointmentscheduler.entity.user		81%		50%	8	36	12	73	5	32	0	2
com.example.slabiak.appointmentscheduler.config		0%		0%	5	5	9	9	4	4	2	2
com.example.slabiak.appointmentscheduler.exception		0%		n/a	3	3	3	3	3	3	3	3
com.example.slabiak.appointmentscheduler		0%		n/a	2	2	3	3	2	2	1	1
Total	5,849 of 6,980	16%	337 of 370	8%	560	694	1,315	1,636	380	506	39	56

Fig 7: Image showing Jacoco Test Metrics Report for the Project

Task-3

For this task, we have to critique the test implementation in the project and provide at least three strong and weak aspects to support our arguments of the test implementation.

Strong Aspects

With my limited knowledge of the Test Smells and the JUnit Test Best Practices, I could only find one String Aspect for the Test Implementation in the project.

Usage of Mocking

1. **Mocking**
   - Mocking is when we create objects that act like their real counterparts.
   - They are used to isolate the code being tested and we can control the testing environment.
   - The goal of mocking is to mimic the behavior of dependencies without actually invoking them as it makes our test cases as independent as they can be and also, for speed.
   - Appointment Scheduler uses Mockito dependency to create and manage mock objects.
   - It should also be noted that an over-reliance on mocking can lead to tests that don't accurately reflect the real-world behavior of the system. It's crucial to have a balance between isolating code for testing purposes and ensuring that the tests remain representative of actual system behavior.

2. **Mocking used in the Project**
   - The project has multiple instances where mocking is aptly used like shown in the Fig 8.
   - There, we can see that the code uses @Mocks to create mock classes for BCryptPasswordEncoder and defines what should happen when the instance if called.
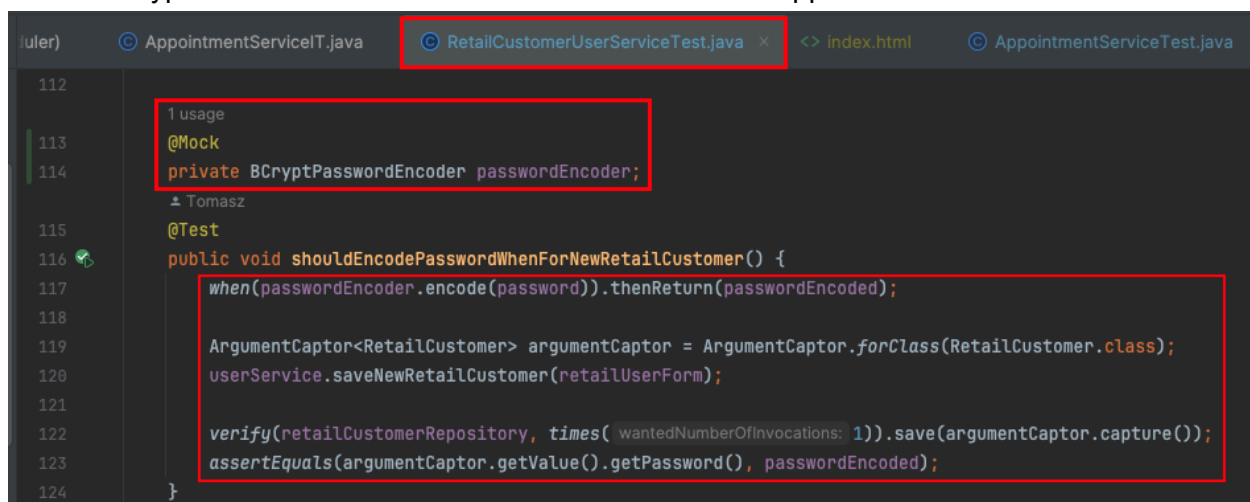

Fig 8: Image showing the use of @Mock and when(...).thenReturn(...) methods

Usage of @Before scripts

1. **@Before notation in JUnit**
   - The @Before annotation is used when different test cases share the same logic[6].
   - The method with the @Before annotation always runs before the execution of each test case.
   - This annotation is commonly used to develop necessary preconditions for each @Test method.
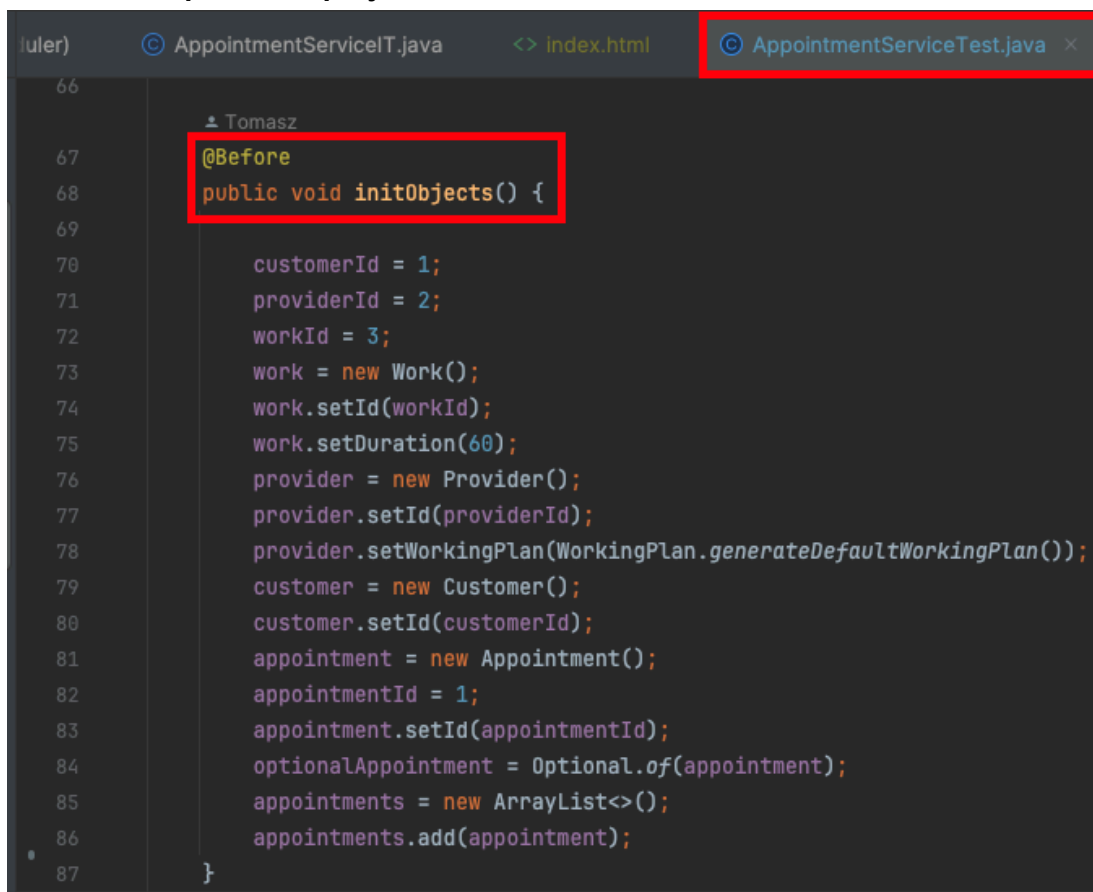   - Some examples of common expensive operations are the creation of a database connection or the startup of a server[7].

2. **Before Scripts in the project**

Fig 9: Image showing the use of @Before script

Weak Aspects

Ambiguous Test Naming

- For enhanced code comprehension, we should ensure that the names of test methods correctly convey the functionality they are verifying.
- This helps easier understanding for other developers regarding the purpose of each test.
- In the project, we can see that there is a test with a misleading name.
- In Fig 10, we can he test method named "shouldSaveNewRetailCustomer" is not the best choice because it suggests that the test is about saving a retail customer, while it's testing the creation of a new appointment
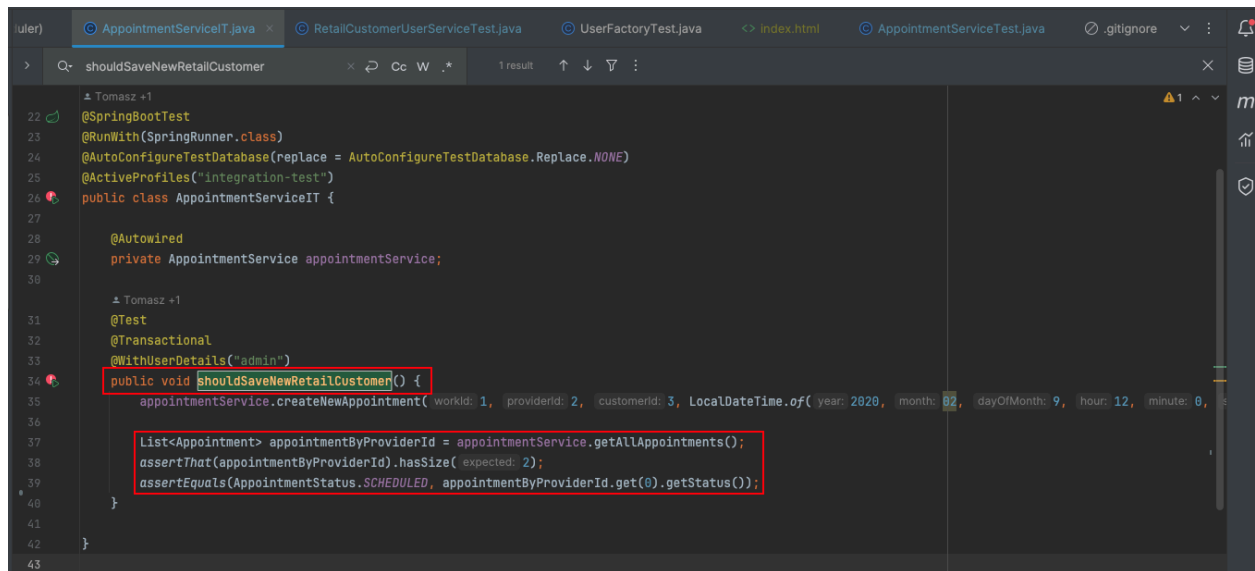
Fig 10: Image showing the min-naming of the shouldSaveNewRetailCustomer() test

Magic Numbers and Exception testing

- Currently, the Test cases majorly use magic numbers as test data, which can hinder understanding for others about the reasoning behind using them (See Fig 11).
- To improve, we can replace the magic numbers with constant data for improved clarity.
- The test folder's unit tests often ignore the scenarios leading to cases where Exceptions can be raised.
- We can add unit tests by considering scenarios where the program may result in runtime exceptions.

```
     ± Tomasz
105  @Test(expected = RuntimeException.class)
106  public void shouldNotBookAppointmentWhenAppointmentStartIsNotWithinProviderWorkingHours() {
107      LocalDateTime startOfNewAppointment = LocalDateTime.of( year: 2019, month: 01, dayOfMonth: 01, hour: 5, minute: 59);
108
109      when(workService.isWorkForCustomer(workId, customerId)).thenReturn( t: true);
110      when(workService.getWorkById(workId)).thenReturn(work);
111      when(userService.getProviderById(providerId)).thenReturn(provider);
112
113      ArgumentCaptor<Appointment> argumentCaptor = ArgumentCaptor.forClass(Appointment.class);
114      appointmentService.createNewAppointment(workId, providerId, customerId, startOfNewAppointment);
115      verify(appointmentRepository, times( wantedNumberOfInvocations: 1)).save(argumentCaptor.capture());
116  }
117  |
```

Fig 11: Image showing use of the Magic-number in the
shouldNotBookAppointmentWhenAppointmentStartIsNotWithinProviderWorkingHours() test

Documentation and Comments

- Currently, the test methods currently lack comments and proper documentation outlining the actual job of the test and verification it performs.
- Including proper comments describing the expected behavior of the test can significantly improve the understandability of the tests for other developers.
- It is a best practice to follow to consistently include comments and documentation within the test case code.
- By using JavaDocs, we can get a structured approach for thorough commenting and style of the test code.
- As seen in Fig 12, the test's objective is currently ambiguous, and the absence of a comment contributes to the lack of clarity. With a descriptive comment, the true purpose of the test would become comprehensible.


```
     ± Tomasz
105  @Test(expected = RuntimeException.class)
106  public void shouldNotBookAppointmentWhenAppointmentStartIsNotWithinProviderWorkingHours() {
107      LocalDateTime startOfNewAppointment = LocalDateTime.of( year: 2019, month: 01, dayOfMonth: 01, hour: 5, minute: 59);
108
109      when(workService.isWorkForCustomer(workId, customerId)).thenReturn( t: true);
110      when(workService.getWorkById(workId)).thenReturn(work);
111      when(userService.getProviderById(providerId)).thenReturn(provider);
112
113      ArgumentCaptor<Appointment> argumentCaptor = ArgumentCaptor.forClass(Appointment.class);
114      appointmentService.createNewAppointment(workId, providerId, customerId, startOfNewAppointment);
115      verify(appointmentRepository, times( wantedNumberOfInvocations: 1)).save(argumentCaptor.capture());
116  }
117  |
```

Fig 12: Lack of proper commenting makes it difficult to understand the test's intent

Live Database Connections

- In the repository test cases, we can notice that it's a standard practice to establish direct connections with databases for various operations.
- When conducting unit tests with JUnit, it is advisable to test one method in isolation at a time. It helps in avoiding dependencies on other classes or databases.

- When a method interfaces with external components such as databases, it can introduce delay and add complexity to the testing process.
- To streamline testing, we generate stubs, fakes, or mock objects for these external dependencies, ensuring the effectiveness of the tests independently of other code.
- In Fig 13 and Fig 14, we can clearly see that the test cases actually try to interact with the database object that go against the best practices.

Fig 13: Image showing the mis-naming of the shouldSaveNewRetailCustomer() test

Fig 14: Image showing the min-naming of the shouldSaveNewRetailCustomer() test

Production Database Connection made in Test Environment
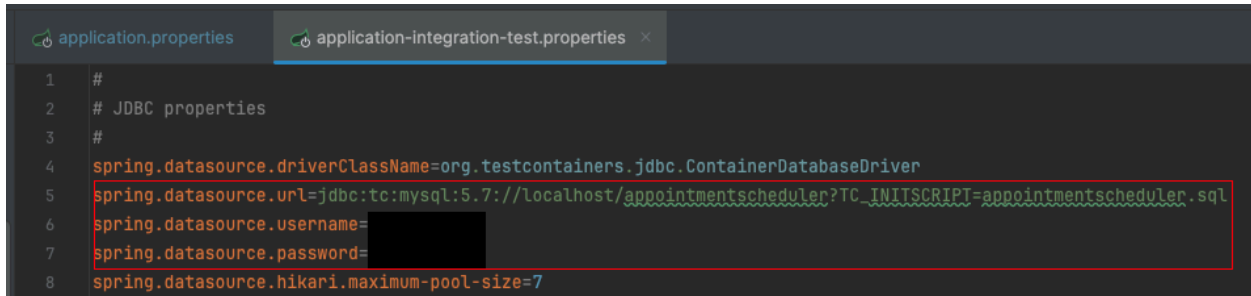
- In the property files for the main src application and the test folder, the property file mentions the database connection string as a different parameter (See Fig 15 for reference)
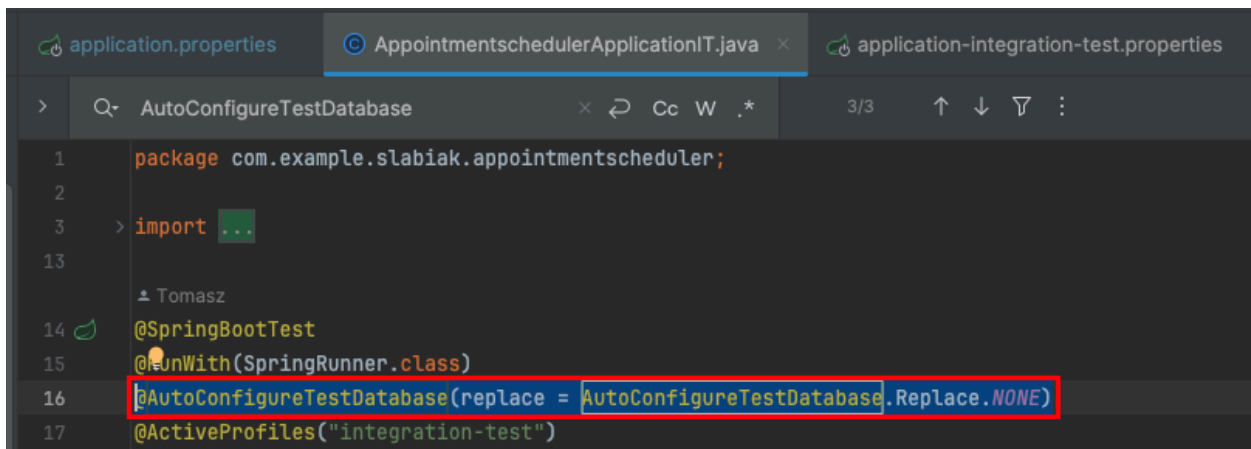
Fig 15: Application property file - Main Application

Fig 16: Application property file - Test sub-directory

- We can notice that this configuration will be making the changes directly in the production database because of the code shown in the Fig 17

Fig 17: Main Test file - Test subdirectory reusing the same database connection

- The test will be using the same database configuration as the main application and it will be affecting the production Data.
- To correct this, we might need a separate testing schema to avoid interfering with the production.

Task-4

For this part, we have to Implement at least **three** new tests for the repository. It could be for new source code elements (new class or method) or for existing code. The newly added tests must not fail due to compilation issues; however, it is fine if they identify a new bug in the project.

The newly added tests must improve the code coverage and not be trivial (for example, creating a new test from an existing test by changing the passed parameters; or, creating a test case for getter/setter method).

I tested the following three controller classes, as the repository did not include test cases for them:

- HomeController
- ExchangeController
- InvoiceController

The test cases are pushed to the repository.
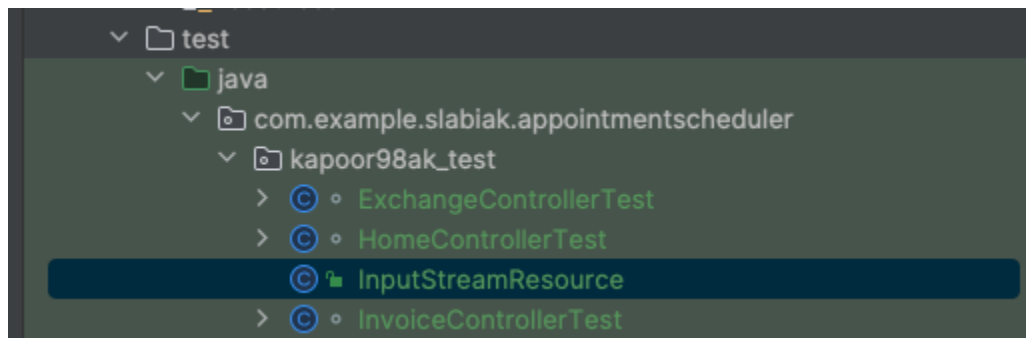
Test cases are listed in Fig 18.

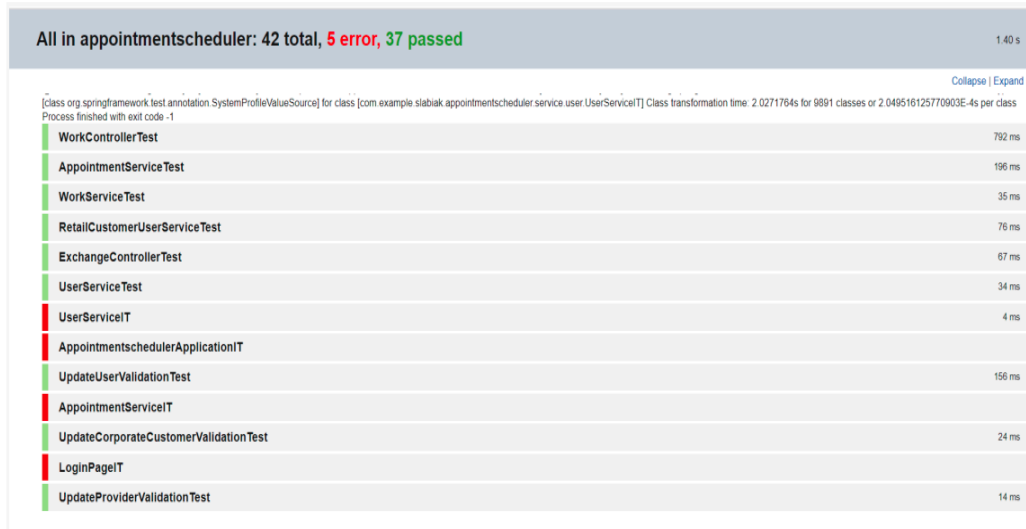Fig 18: Created Test Cases for the Appointment Builder
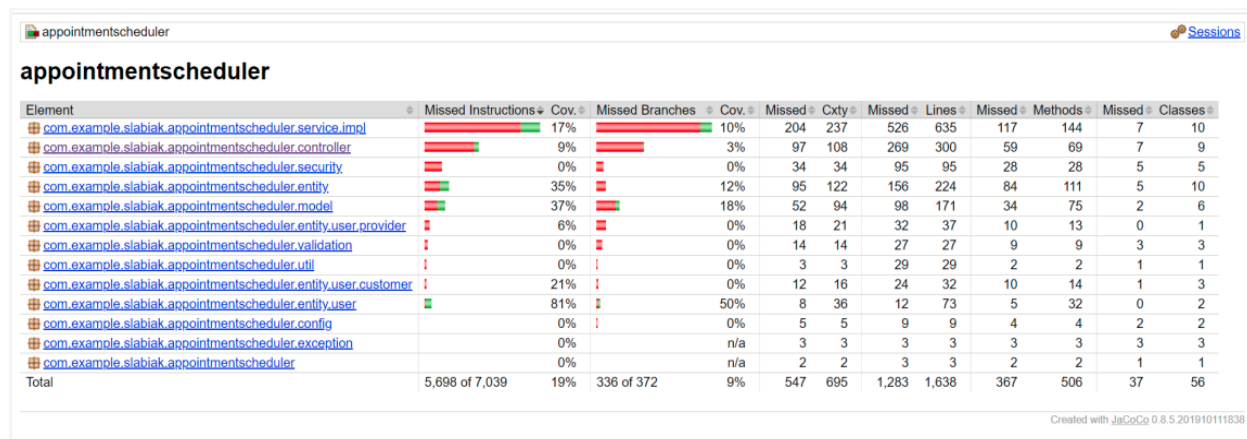
Fig 20: New result of Automated Tests

Fig 21: New result of Code Coverage

References

[1]   SEART - SoftwarE Analytics Research Team. [Online]. Available: https://seart-ghs.si.usi.ch/.

[2]   Appointment scheduler. [Online]. Available: https://github.com/slabiak/AppointmentScheduler/.

[3]   Baeldung. "Code Coverage." Baeldung, January 1, 2024. [Online]. Available:
        https://www.baeldung.com/cs/code-coverage.

[4]   Jacoco. MVN Reporsitory, [Online]. Available:
        https://mvnrepository.com/artifact/org.jacoco/jacoco-maven-plugin

[5]   IntelliJ. "Code Coverage." IntelliJ IDEA, 22 February, 2023. [Online] Available:
        https://www.jetbrains.com/help/idea/code-coverage.html

[6]   Educative. "What is the @Before annotation in JUnit testing?". [Online]. Available:
        https://www.educative.io/answers/what-is-the-before-annotation-in-junit-testing

[7]   Baeldung. "@Before vs @BeforeClass vs @BeforeEach vs @BeforeAll." Baeldung, January 1,
        2024. [Online]. Available:
        https://www.baeldung.com/junit-before-beforeclass-beforeeach-beforeall.

[8]