

Lab Report: Testing

Kevin Barrios

NSHE ID: 2001697903

Professor: Dr. Businge

September 17, 2024

Fork Repo:<https://github.com/imawful/Main>

Introduction

This lab focuses on the process of testing in software development. It explores various tasks that are essential for ensuring software reliability and functionality. The purpose of this report is to provide documentation and analysis for each task assigned in the lab. Throughout this report, tasks related to software testing will be detailed by including objectives, descriptions, and results.

Task 1: JPacman Test Coverage

Objective: Building and Testing Jpacman in IntelliJ.

Description:

To start this task I installed IntelliJ on my machine and cloned the Jpacman repository provided in the lab's assignment. After this I opened the Jpacman project and used gradle to build and run the application. Once I made sure that everything was working on moved on to running the tests by executing Run 'jpacman [test]' with Coverage. Finally, once execution of tests were complete a report was generated and a new window called Coverage showed up in IntelliJ. Below I share my results.

Results:

Coverage Results

Element	Class, %	Method, %	Line, %	Branch, %
nl.tudelft.jpacman	3% (2/55)	1% (5/312)	1% (14/1127)	0% (1/521)
board	20% (2/10)	9% (5/53)	9% (14/141)	1% (1/96)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)	0% (0/8)
game	0% (0/3)	0% (0/14)	0% (0/37)	0% (0/14)
integration	0% (0/1)	0% (0/4)	0% (0/6)	100% (0/0)
level	0% (0/13)	0% (0/78)	0% (0/343)	0% (0/167)
npc	0% (0/10)	0% (0/47)	0% (0/233)	0% (0/116)
points	0% (0/2)	0% (0/7)	0% (0/19)	0% (0/4)
sprite	0% (0/6)	0% (0/45)	0% (0/119)	0% (0/48)
ui	0% (0/6)	0% (0/31)	0% (0/123)	0% (0/60)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)	0% (0/6)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)	0% (0/2)
PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)	100% (0/0)

Conclusion:

The results above show the coverage report generated after running `jpacman [test]`. There are various tabs showing class coverage, method coverage, branch coverage, and line coverage. Is the coverage good enough? After viewing the results we can see that the test coverage is very low in multiple areas of the project. It appears that there is only coverage in `nl.tudelft.jpacman.board` where we test `Direction.java` and `Unit.java`. The current tests for `Direction.java` seem to have an acceptable coverage, whereas `Unit.java` lacks in method, line, and branch coverage. As the tests stand right now, I believe that the coverage is not good enough.

Task 2/2.1: Increasing Coverage on JPacman

Objective: Writing more test to increase the coverage on JPacman.

Description:

In order to increase the coverage on JPacman we simply have to write more tests. Following the directions given in the lab I examine the current test `DirectionTest.java` and use that as an example to write a new test for the player class. The test I wrote for the player class is called `testIsAlive()` and is placed in `PlayerTest.java`. I followed the explanation given on the lab to write the test and the before and after coverage can be seen in the results. After writing `testIsAlive()` I added two additional tests to the player class. I wrote a test for checking that our initial points for pacman is 0 (`testInitialPoints()`). After this, I wrote a test making sure the add points functionality works as expected (`testAddPoints()`) by calling add points with a value of 200 and assert that pacman points is 200. I also decided to expand on the tests written in `OccupantTest.java`. I wrote a test in occupant tests that makes sure that the occupy method for a unit properly occupies a square (`testOccupySquare()`) by asserting that after calling occupy, `unit.hasSquare()` returns true. I also wrote a test to make sure that the leave square method also worked on the unit (`testLeaveSquare()`) by asserting that, after calling occupy() and then leaveSquare(), `unit.hasSquare()` returns false. After writing

my unit tests the coverage for board and level both increased, my results are shown below.

Results:

I share my coverage results starting from the initial coverage to my last written unit test.

Initial Coverage for player

The initial coverage was shared in the results under task 1. The coverage for Player had all fields: class, method, line, and branch at 0%.

Unit test - testIsAlive()

```
public class PlayerTest {  
  
    private Player pac;  
  
    @BeforeEach  
    void setUp()  
    {  
        PlayerFactory pf = new PlayerFactory(new PacManSprites());  
        pac = pf.createPacMan();  
    }  
  
    /* Tests that pacman starts in the alive state. */  
    @Test  
    void testIsAlive()  
    {  
        assertThat(pac.isAlive()).isEqualTo(true);  
    }  
}
```

© Player	100% (1/1)	25% (2/8)	33% (8/24)	0% (0/6)
----------	------------	-----------	------------	----------

The code snippet shows my unit test for the first half of this task where we write a test for Player.IsAlive(). After looking at the other unit tests written in this project I noticed that I can setup a "Before Each" function that will instantiate my player before each unit test. In this unit test I create my player and then I assert that the player starts in the alive state. The coverage for Player increased from 0% 0% 0% 0% to 100% 25% 33% 0%.

Unit test - testInitialPoints()

```
/* Tests that pacman starts with 0 points. */
@Test
void testInitialPoints()
{
    assertThat(pac.getScore()).isEqualTo(0);
}
```

© Player	100% (1/1)	37% (3/8)	37% (9/24)	0% (0/6)
----------	------------	-----------	------------	----------

The code snippet shows my unit test for testing the initial points. Similar to doing the is alive test, I instantiate the player (in the before each function) and then I assert that the player starts with 0 points. The coverage for Player increased from 100% 25% 33% 0% to 100% 37% 37% 0%.

Unit test - testAddPoints()

```
/* Tests that adding points works correctly, when we start pacman should
   have no points, so we add 200 points and make sure the pacman's score
   is 200.
   */
@Test
void testAddPoints()
{
    pac.addPoints(200);
    assertThat(pac.getScore()).isEqualTo(200);
}
```

© Player	100% (1/1)	50% (4/8)	45% (11/24)	0% (0/6)
----------	------------	-----------	-------------	----------

The code snippet shows my unit test for testing the add points function in the Player class. I instantiate the player (in the before each function) and then I call the add points method on the player object with the value of 200. I then assert that the Players get score function returns 200. The coverage for Player increased from 100% 37% 37% 0% to 100% 50% 45% 0%.

Unit tests for Unit

```
/**
 * Resets the unit under test.
 */
@BeforeEach
void setUp() {
    unit = new BasicUnit();
}
```

Unit	100% (1/1)	20% (2/10)	13% (4/29)	2% (1/34)
------	------------	------------	------------	-----------

For my next unit tests I write them for the Unit class in the board package. The image above shows the initial coverage for Unit before writing my tests. I also show the given before each function which just instantiates a basic unit that will be used for testing.

Unit test - testOccupySquare()

```
/* Test that the unit has a square once it calls occupy */
@Test
void testOccupySquare()
{
    BasicSquare bs = new BasicSquare();
    unit.occupy(bs);
    assertTrue(unit.hasSquare());
}
```

Unit	100% (1/1)	50% (5/10)	41% (12/29)	26% (9/34)
------	------------	------------	-------------	------------

The code snippet shows my unit test for testing that a Unit can occupy a square by calling occupy. I first instantiate a basic square, a square object used for testing, and then call occupy on the unit passing in the basic square. After this I assert that unit.hasSquare returns true. The coverage for Unit increased from 100% 20% 13% 2% to 100% 50% 41% 26%.

Unit test - testLeaveSquare()

```

/* Test that the unit can leave a square once occupying one */
@Test
void testLeaveSquare()
{
    BasicSquare bs = new BasicSquare();
    unit.occupy(bs);
    unit.leaveSquare();
    assertThat(unit.hasSquare()).isFalse();
}

```

Unit 100% (1/1) 60% (6/10) 58% (17/29) 41% (14/34)

The code snippet shows my unit test for testing that a Unit can leave a square after occupying one. I first instantiate a basic square and then call occupy on the unit with the basic square as an argument. After this I call the leave square method on the unit and then assert that unit.hasSquare returns false. The coverage for Unit increased from 100% 50% 41% 26% to 100% 60% 58% 41%.

Overall Project Coverage

Coverage jpacman [test] x				
Element	Class, %	Method, %	Line, %	Branch, %
all	16% (9/55)	13% (42/312)	10% (125/1141)	8% (45/539)
> nl.tudelft.jpacman.sprite	66% (4/6)	44% (20/45)	51% (66/128)	31% (21/66)
> nl.tudelft.jpacman.board	30% (3/10)	28% (15/53)	30% (43/141)	25% (24/96)
> nl.tudelft.jpacman.level	15% (2/13)	8% (7/78)	4% (16/348)	0% (0/167)
> nl.tudelft.jpacman.npc.ghost	0% (0/9)	0% (0/43)	0% (0/225)	0% (0/116)
> nl.tudelft.jpacman.ui	0% (0/6)	0% (0/31)	0% (0/123)	0% (0/60)
> nl.tudelft.jpacman	0% (0/3)	0% (0/27)	0% (0/74)	0% (0/8)
> nl.tudelft.jpacman.game	0% (0/3)	0% (0/14)	0% (0/37)	0% (0/14)
> nl.tudelft.jpacman.points	0% (0/2)	0% (0/7)	0% (0/19)	0% (0/4)
> nl.tudelft.jpacman.fuzzer	0% (0/1)	0% (0/6)	0% (0/32)	0% (0/8)
> nl.tudelft.jpacman.integration	0% (0/1)	0% (0/4)	0% (0/6)	100% (0/0)
> nl.tudelft.jpacman.npc	0% (0/1)	0% (0/4)	0% (0/8)	100% (0/0)

After writing all my unit tests, including testIsAlive, the overall coverage increased from the initial values of 3% 1% 1% 0% to 16% 13% 10% 8%.

Conclusion:

The results above show the coverage as I write each unit test. The main focus when writing these tests were to increase the coverage for the Player class and the Unit class. If I were to keep writing unit tests for different classes we would see the overall coverage keep increasing. As it stands now the overall coverage for JPacman still needs some work but after writing a few unit tests it makes a significant difference.

Task 3: JaCoCo Report on JPacman

Objective: Analyzing the report generated from the JaCoCo tool and Comparisson with IntelliJ.

Description:

In this task I follwed the instructions in the lab and viewed the JaCoCo report that was generated in the build/reports/jacoco/test/html directory. After viewing this report I see that the results are fairly different to that of which I see in IntelliJ. The report generated from the JaCoCo tool shows more details about the coverage and allows you to see the code snippets with highlights wherever code branches are missed. The coverage percentages in the JaCoCo report also show higher percentages to that of IntelliJ. Below I share my results of the generated report.

Results:



















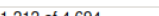

Comparison between coverage reports.

IntelliJ Coverage

Element	Class, %	Method, %	Line, %	Branch, %
all	16% (9/55)	13% (42/312)	10% (125/1141)	8% (45/539)
nl.tudelft.jpacman.sprite	66% (4/6)	44% (20/45)	51% (66/128)	31% (21/66)
nl.tudelft.jpacman.board	30% (3/10)	28% (15/53)	30% (43/141)	25% (24/96)
nl.tudelft.jpacman.level	15% (2/13)	8% (7/78)	4% (16/348)	0% (0/167)
nl.tudelft.jpacman.npc.ghost	0% (0/9)	0% (0/43)	0% (0/225)	0% (0/116)
nl.tudelft.jpacman.ui	0% (0/6)	0% (0/31)	0% (0/123)	0% (0/60)
nl.tudelft.jpacman	0% (0/3)	0% (0/27)	0% (0/74)	0% (0/8)
nl.tudelft.jpacman.game	0% (0/3)	0% (0/14)	0% (0/37)	0% (0/14)
nl.tudelft.jpacman.points	0% (0/2)	0% (0/7)	0% (0/19)	0% (0/4)
nl.tudelft.jpacman.fuzzer	0% (0/1)	0% (0/6)	0% (0/32)	0% (0/8)
nl.tudelft.jpacman.integration	0% (0/1)	0% (0/4)	0% (0/6)	100% (0/0)
nl.tudelft.jpacman.npc	0% (0/1)	0% (0/4)	0% (0/8)	100% (0/0)

JaCoCo Report

jpacman

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.tudelft.jpacman.level		67%		57%	74	155	104	344	21	69	4	12
nl.tudelft.jpacman.npc.ghost		71%		55%	56	105	43	181	5	34	0	8
nl.tudelft.jpacman.ui		77%		47%	54	86	21	144	7	31	0	6
default		0%		0%	12	12	21	21	5	5	1	1
nl.tudelft.jpacman.board		86%		59%	43	93	2	110	0	40	0	7
nl.tudelft.jpacman.sprite		86%		59%	30	70	11	113	5	38	0	5
nl.tudelft.jpacman		69%		25%	12	30	18	52	6	24	1	2
nl.tudelft.jpacman.points		60%		75%	1	11	5	21	0	9	0	2
nl.tudelft.jpacman.game		87%		60%	10	24	4	45	2	14	0	3
nl.tudelft.jpacman.npc		100%		n/a	0	4	0	8	0	4	0	1
Total	1,212 of 4,694	74%	292 of 637	54%	292	590	229	1,039	51	268	6	47

Conclusion:

Are the coverage results from JaCoCo similar to the ones you got from IntelliJ in the last task?

No, the results from these two reports are different. I believe that JaCoCo is either running more tests that might be related to the gradle building process or that IntelliJ is showing the coverage on a certain subset of the overall tests. Either way, JaCoCo report will show the code that is being covered and the code that is not being covered with color highlighting providing an easy way to see where changes should be made. I also had a suspicion that the JaCoCo report could be showing percentages of missed coverage while IntelliJ shows percentages of covered code however I dismissed the suspicion after looking into the details of the JaCoCo report such as examining code pieces.

Did you find helpful the source code visualization from JaCoCo on uncovered branches?

Yes, looking at the source code through the JaCoCo report provides helpful highlighting on uncovered branches. This allows us to easily see which parts of the code are being tested and what is being left out. With this visualization we can come to certain conclusions about the class and the related tests allowing for efficient improvements.

Which visualization did you prefer and why? IntelliJ's coverage window or JaCoCo's report?

Personally I prefer IntelliJ's coverage window for visualizing the unit tests. I found that the JaCoCo report showed too much information in the columns and didn't give a clear direction of where to improve without examining further. When looking at IntelliJ's coverage window it was clear which areas are lacking in unit tests and it seemed to be displaying results for exactly what I was working on. After writing unit tests and reexamining the coverage window I can see the improvements that I was making and can judge if the coverage was sufficient. Although, JaCoCo did provide extra details including the highlighting which might be useful for more robust tests.

Task 4: Working with Python Test Coverage

Objective:

Practice improving test coverage in Python. Generate pytest report to identify test cases to cover.

Description:

I started this task by first cloning the python testing lab and installing the dependencies. Before writing any code I ran pytest to generate an initial report so that I could identify if test cases are passing and see any missing lines that need to be covered. The initial report generated a coverage of 72% and we are tasked to increase the coverage to 100%. Following the instructions from the lab, and the pytest report, we see that line 26 in account.py is missing coverage. I wrote a new test case in test_account.py where I test the `__repr__` method of Account. Running pytest again we see that the coverage increases from 72% to 74% and the missing line number 26 is omitted. Continuing on with the lab we notice line 30 in the missing tab on the report. So we write a test for `account.to_dict()` where we increase the overall coverage to 77%. I continued on with this approach of analyzing the missing lines

columns in pytest and coming up with a test case in test_account.py to cover it. Below I show the test cases made as I attempted to reach 100% coverage.

Results:

Missing: Line 26

```
===== test session starts =====
platform linux -- Python 3.12.6, pytest-8.3.3, pluggy-1.5.0 -- /usr/bin/python
cachedir: .pytest_cache
rootdir: /home/kusa/School/472/assignments/labs/testing/test_coverage
configfile: pytest.ini
plugins: anyio-4.4.0, cov-5.0.0
collected 2 items

tests/test_account.py::test_create_all_accounts PASSED [ 50%]
tests/test_account.py::test_create_an_account PASSED [100%]

----- coverage: platform linux, python 3.12.6-final-0 -----
Name                Stmts   Miss  Cover   Missing
-----
models/__init__.py      7      0   100%
models/account.py     40     13    68%   26, 30, 34-35, 45-48, 52-54, 74-75
TOTAL                  47     13    72%
```

===== 2 passed in 2.31s =====

After cloning the python testing lab repository and running pytest we get our initial report which shows us that we have 2 passing test cases. We analyze the missing column to see which lines need coverage. Looking at line 26 we see that it corresponds to the accounts `__repr__` method.

```
def test_repr():
    ''' Test Representation of an Account '''
    account = Account()
    account.name = "Foo"
    assert str(account) == "<Account 'Foo'>"
```

We add in a new test case to test_account.py where we write the test case given in the lab which makes sure that the repr method returns what we expect.

```

===== test session starts =====
platform linux -- Python 3.12.6, pytest-8.3.3, pluggy-1.5.0 -- /usr/bin/python
cachedir: .pytest_cache
rootdir: /home/kusa/School/472/assignments/labs/testing/test_coverage
configfile: pytest.ini
plugins: anyio-4.4.0, cov-5.0.0
collected 3 items

tests/test_account.py::test_create_all_accounts PASSED [ 33%]
tests/test_account.py::test_create_an_account PASSED [ 66%]
tests/test_account.py::test_repr PASSED [100%]

----- coverage: platform linux, python 3.12.6-final-0 -----
Name                Stmts   Miss  Cover   Missing
-----
models/__init__.py      7      0   100%
models/account.py     40     12    70%   30, 34-35, 45-48, 52-54, 74-75
-----
TOTAL                  47     12    74%

===== 3 passed in 2.38s =====
[kusa@awfulc740 test_coverage]$

```

After running pytest again we see that we now have 3 passing test cases, line 26 is eliminated from the missing column, and the overall coverage increases from 72% to 74%. We target line 30 next.

Missing: Line 30

```

def test_to_dict():
    ''' Test Account to Dict '''
    rand = randrange(0, len(ACCOUNT_DATA))
    data = ACCOUNT_DATA[rand]
    account = Account(**data)
    result = account.to_dict()

    assert account.name == result["name"]
    assert account.email == result["email"]
    assert account.phone_number == result["phone_number"]
    assert account.disabled == result["disabled"]
    assert account.date_joined == result["date_joined"]

```

We write the test.to_dict method from the lab to target the missing line 30. Here, we make sure that the account's to dict function works as expected.

```
===== test session starts =====
platform linux -- Python 3.12.6, pytest-8.3.3, pluggy-1.5.0 -- /usr/bin/python
cachedir: .pytest_cache
rootdir: /home/kusa/School/472/assignments/labs/testing/test_coverage
configfile: pytest.ini
plugins: anyio-4.4.0, cov-5.0.0
collected 4 items

tests/test_account.py::test_create_all_accounts PASSED [ 25%]
tests/test_account.py::test_create_an_account PASSED [ 50%]
tests/test_account.py::test_repr PASSED [ 75%]
tests/test_account.py::test_to_dict PASSED [100%]

----- coverage: platform linux, python 3.12.6-final-0 -----
Name                               Stmts  Miss  Cover   Missing
-----
models/__init__.py                   7      0   100%
models/account.py                   40     11    72%   34-35, 45-48, 52-54, 74-75
TOTAL                               47     11    77%
```

===== 4 passed in 2.33s =====

After running pytest again we increase the coverage 74% to 77% and notice from the missing column that we must now look at lines 34-35.

Missing: Lines 34-35

```
def test_from_dict():  
    ''' Test Account from Dict '''  
  
    # create dict for sample data  
    sample_data = {  
        "id" : 2001697903,  
        "name" : "sample_name",  
        "email" : "sample@email.com",  
        "phone_number" : "7027448658",  
        "disabled" : False,  
        "date_joined" : "09/15/2024"  
    }  
  
    #create new account and call from_dict with sample data  
    account = Account()  
    account.from_dict(sample_data)  
  
    # assert that account fields match sample data dictionary  
    # to ensure that the from_dict method works correctly.  
    assert account.id == sample_data["id"]  
    assert account.name == sample_data["name"]  
    assert account.email == sample_data["email"]  
    assert account.phone_number == sample_data["phone_number"]  
    assert account.disabled == sample_data["disabled"]  
    assert account.date_joined == sample_data["date_joined"]
```

Looking at lines 34-35 in `account.py` we see that we need to write a test case for testing the account's `from_dict` method. To write this test case I simply created a dictionary that contained fields that correspond to the values of an account, then created an account and called the `from_dict` method on the created account with my dictionary. Afterwards I used asserts to make sure that the account's fields were properly set and matched the values of my dictionary.

```
===== test session starts =====
platform linux -- Python 3.12.6, pytest-8.3.3, pluggy-1.5.0 -- /usr/bin/python
cachedir: .pytest_cache
rootdir: /home/kusa/School/472/assignments/labs/testing/test_coverage
configfile: pytest.ini
plugins: anyio-4.4.0, cov-5.0.0
collected 5 items

tests/test_account.py::test_create_all_accounts PASSED [ 20%]
tests/test_account.py::test_create_an_account PASSED [ 40%]
tests/test_account.py::test_repr PASSED [ 60%]
tests/test_account.py::test_to_dict PASSED [ 80%]
tests/test_account.py::test_from_dict PASSED [100%]

----- coverage: platform linux, python 3.12.6-final-0 -----
Name                               Stmts   Miss  Cover   Missing
-----
models/__init__.py                   7        0   100%
models/account.py                   40        9    78%   45-48, 52-54, 74-75
TOTAL                               47        9    81%
```

```
===== 5 passed in 2.49s =====
```

Running pytest again and looking at the report we see that we cover the missing lines and the coverage increases from 77% to 81% we now target lines 45-48.

Missing: Lines 45-48

```
def test_update():
    ''' Test Account update '''
    #generate random account
    rand = randrange(0, len(ACCOUNT_DATA))
    data = ACCOUNT_DATA[rand] # get a random account
    account = Account(**data)
    account.create() #create the account

    # grab a field.
    oldname = account.to_dict()["name"]
    # change a field.
    account.name = "TESTNAME"
    # call update
    account.update()

    failed = False
    try:
        account.update()
    except DataValidationError as e:
        failed = str(e) == "Update called with empty ID field"

    assert not failed
```

```
def test_failed_update():
    ''' Test a failed update '''
    rand = randrange(0, len(ACCOUNT_DATA))
    data = ACCOUNT_DATA[rand]
    account = Account(**data)
    account.create()

    account.id = None

    failed = False
    try:
        account.update()
    except DataValidationError as e:
        failed = str(e) == "Update called with empty ID field"

    assert failed
```

Missing lines 45-48 correspond to the update function in account.py. We write two test cases test_update and test_failed_update to cover these lines. In the update test I run the account.update() function under a try block so that I can catch the potential error that the function will yield. If the error message we are expecting from a failure never arises we know that the update succeeded. In the failed update test I change the account's id field to a non

integer value and run `account.update()` under the same conditions and assert the failure went as expected.

```
===== test session starts =====
platform linux -- Python 3.12.6, pytest-8.3.3, pluggy-1.5.0 -- /usr/bin/python
cachedir: .pytest_cache
rootdir: /home/kusa/School/472/assignments/labs/testing/test_coverage
configfile: pytest.ini
plugins: anyio-4.4.0, cov-5.0.0
collected 7 items

tests/test_account.py::test_create_all_accounts PASSED [ 14%]
tests/test_account.py::test_create_an_account PASSED [ 28%]
tests/test_account.py::test_repr PASSED [ 42%]
tests/test_account.py::test_to_dict PASSED [ 57%]
tests/test_account.py::test_from_dict PASSED [ 71%]
tests/test_account.py::test_update PASSED [ 85%]
tests/test_account.py::test_failed_update PASSED [100%]

----- coverage: platform linux, python 3.12.6-final-0 -----
Name                               Stmts  Miss  Cover   Missing
-----
models/__init__.py                   7      0   100%
models/account.py                   40      5    88%   52-54, 74-75
-----
TOTAL                               47      5    89%
```

===== 7 passed in 2.32s =====

Again, we run pytest and see we increase the coverage from 81% to 89% and target lines 52-54 next.

Missing: Lines 52-54

```
def test_delete():
    ''' Test account delete '''
    rand = randrange(0, len(ACCOUNT_DATA))
    data = ACCOUNT_DATA[rand]
    account = Account(**data)
    account.create()

    assert account in Account.all()

    account.delete()

    assert account not in Account.all()
```

Missing lines 52-54 correspond to the delete function in account.py. I write a simple test case that first creates an account and assert it exists and then call delete and assert it no longer exists.

```
===== test session starts =====
platform linux -- Python 3.12.6, pytest-8.3.3, pluggy-1.5.0 -- /usr/bin/python
cachedir: .pytest_cache
rootdir: /home/kusa/School/472/assignments/labs/testing/test_coverage
configfile: pytest.ini
plugins: anyio-4.4.0, cov-5.0.0
collected 8 items

tests/test_account.py::test_create_all_accounts PASSED [ 12%]
tests/test_account.py::test_create_an_account PASSED [ 25%]
tests/test_account.py::test_repr PASSED [ 37%]
tests/test_account.py::test_to_dict PASSED [ 50%]
tests/test_account.py::test_from_dict PASSED [ 62%]
tests/test_account.py::test_update PASSED [ 75%]
tests/test_account.py::test_failed_update PASSED [ 87%]
tests/test_account.py::test_delete PASSED [100%]

----- coverage: platform linux, python 3.12.6-final-0 -----
Name                               Stmts  Miss  Cover   Missing
-----
models/__init__.py                   7      0   100%
models/account.py                   40      2    95%   74-75
TOTAL                               47      2    96%

===== 8 passed in 2.40s =====
```


We increase the coverage from 89% to 96% and look at lines 74-75 next.

Missing: Lines 74-75

```
def test_find():  
    ''' Test the class method find from account class '''  
    #generate random account  
    rand = randrange(0, len(ACCOUNT_DATA))  
    data = ACCOUNT_DATA[rand] # get a random account  
    account = Account(**data)  
    account.create() #create the account  
  
    assert account == Account.find(account.id)
```

Lines 74-75 correspond to the class method find in account.py. I follow the same approach of the other test cases to generate an account and then assert that the find method returns the same account given the account's id.

```

===== test session starts =====
platform linux -- Python 3.12.6, pytest-8.3.3, pluggy-1.5.0 -- /usr/bin/python
cachedir: .pytest_cache
rootdir: /home/kusa/School/472/assignments/labs/testing/test_coverage
configfile: pytest.ini
plugins: anyio-4.4.0, cov-5.0.0
collected 9 items

tests/test_account.py::test_create_all_accounts PASSED [ 11%]
tests/test_account.py::test_create_an_account PASSED [ 22%]
tests/test_account.py::test_repr PASSED [ 33%]
tests/test_account.py::test_to_dict PASSED [ 44%]
tests/test_account.py::test_from_dict PASSED [ 55%]
tests/test_account.py::test_update PASSED [ 66%]
tests/test_account.py::test_failed_update PASSED [ 77%]
tests/test_account.py::test_delete PASSED [ 88%]
tests/test_account.py::test_find PASSED [100%]

===== warnings summary =====
tests/test_account.py::test_find
  /home/kusa/School/472/assignments/labs/testing/test_coverage/models/account.py:75: LegacyAPIWarning: The
  e Query.get() method is considered legacy as of the 1.x series of SQLAlchemy and becomes a legacy construc
  t in 2.0. The method is now available as Session.get() (deprecated since: 2.0) (Background on SQLAlchemy 2
  .0 at: https://sqlalche.me/e/b8d9)
    return cls.query.get(account_id)

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html

----- coverage: platform linux, python 3.12.6-final-0 -----
Name                Stmts  Miss  Cover   Missing
-----
models/__init__.py      7      0   100%
models/account.py     40      0   100%
-----
TOTAL                  47      0   100%

===== 9 passed, 1 warning in 2.43s =====

```

When running pytest I did get a warning from sql alchemy saying that the `query.get()` method is now considered legacy, however the test cases still ran and we reached full coverage of 100%.

Conclusion:

Following the approach introduced to us in the lab where we look at the missing line column from the pytest report, and then look at the lines in `account.py` that match these numbers, we can come up with test cases to write that ensure our code has full coverage.

Task 5: TDD

Objective:

In this Task, we write test cases based on the requirements given, and then write the code to make the test cases pass. This is the Test Driven Development approach.

Description:

To start this lab I cloned the python testing repository given in the instructions. In this repo we will be working with pytest for our test cases and work with HTTP methods and REST guidelines. We start off by editing the tests/test_counter.py to write our first test case called test_create_a_counter(). In this test case we write what we expect to happen when a counter is created and check for it. In this case we make sure that a status code of 201_CREATED is returned from our post request that creates a counter. I follow along with the rest of the given lab instructions and write another test called test_duplicate_a_counter where we make sure there is a conflict response code if we try to make the same counter twice. The lab also helps refactor our code by creating pytest fixtures to make our test cases more clear. It's important to note that as I'm following along these Instructions I am writing the test cases first and then running pytest making sure that our test case fails, and then we go to our counter.py to write the code that makes our tests turn green. We currently have 2 test cases test_create_a_counter and test_duplicate_a_counter that are passing from following TDD, now we must implement the test_update_a_counter and test_read_a_counter and write the code to pass these test cases after we get red tests from pytest. Below I share my results from creating both new test cases.

Results:

Test create counter and duplicate counter.

```
def test_create_a_counter(self, client):
    ''' It should create a counter '''
    result = client.post('/counters/foo')
    assert result.status_code == status.HTTP_201_CREATED

def test_duplicate_a_counter(self, client):
    ''' It should return error for duplicates '''
    result = client.post('/counters/bar')
    assert result.status_code == status.HTTP_201_CREATED
    result = client.post('/counters/bar')
    assert result.status_code == status.HTTP_409_CONFLICT
```

```
===== test session starts =====
platform linux -- Python 3.12.6, pytest-8.3.3, pluggy-1.5.0 -- /usr/bin/python
cachedir: .pytest_cache
rootdir: /home/kusa/School/472/assignments/labs/testing/tdd
configfile: pytest.ini
plugins: anyio-4.4.0, cov-5.0.0
collected 2 items

tests/test_counter.py::TestCounterEndpoints::test_create_a_counter PASSED [ 50%]
tests/test_counter.py::TestCounterEndpoints::test_duplicate_a_counter PASSED [100%]

----- coverage: platform linux, python 3.12.6-final-0 -----
Name                Stmts  Miss  Cover   Missing
-----
src/__init__.py         0      0  100%
src/counter.py         11      0  100%
src/status.py           6      0  100%
-----
TOTAL                  17      0  100%

===== 2 passed in 1.49s =====
```

These two test were given in the lab and we learned how to do TDD while writing them. We now have passing tests in pytest with 100% coverage. We now focus on creating update_a_counter.

Test update a counter.

```
def test_update_a_counter(self, client):
    # create a counter
    result = client.post('/counters/my_counter')
    assert result.status_code == status.HTTP_201_CREATED

    # save the base of our counter and make sure we start at 0
    data = result.json
    baseline = data['my_counter']
    assert baseline == 0

    # call the update counter method
    result2 = client.put(f"/counters/my_counter")

    #make sure that our update was successful.
    assert result2.status_code == status.HTTP_200_OK

    #make sure that are counters data is now one more than previous
    # baseline.
    data2 = result2.json
    curr_value = data2['my_counter']
    assert curr_value == baseline + 1
```

To implement the update a counter method into our app we first write the test case for it. I first create a counter and then call the update method on it and verify I get expected response code. I then make sure that my new value is updated from the previous value. I use a put request for the update method since following the REST guidelines it seems this request is used for updating existing values.

```

===== FAILURES =====
----- TestCounterEndpoints.test_update_a_counter -----

self = <tests.test_counter.TestCounterEndpoints object at 0x7fb3af682210>
client = <FlaskClient <Flask 'src.counter'>>

def test_update_a_counter(self, client):
    # create a counter
    result = client.post('/counters/my_counter')
    assert result.status_code == status.HTTP_201_CREATED

    # save the base of our counter and make sure we start at 0
    data = result.json
    baseline = data['my_counter']
    assert baseline == 0

    # call the update counter method
    result2 = client.put(f"/counters/my_counter")

    #make sure that our update was successful.
    assert result2.status_code == status.HTTP_200_OK
E       assert 405 == 200
E         + where 405 = <WrapperTestResponse streamed [405 METHOD NOT ALLOWED]>.status_code
E         + and   200 = status.HTTP_200_OK

tests/test_counter.py:51: AssertionError

----- coverage: platform linux, python 3.12.6-final-0 -----
Name                Stmts   Miss  Cover   Missing
-----
src/__init__.py         0      0   100%
src/counter.py         12      0   100%
src/status.py           6      0   100%
-----
TOTAL                   18      0   100%

===== short test summary info =====
FAILED tests/test_counter.py::TestCounterEndpoints::test_update_a_counter - assert 405 == 200
===== 1 failed, 2 passed in 1.59s =====

```

We make sure that the test cases runs and fails how we expect it to, we can now write our code in counter.py to make our test case pass.

```

@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    ''' Update a counter '''
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS
    COUNTERS[name] += 1
    return {name: COUNTERS[name]}, status.HTTP_200_OK

```

In counter.py I create a new method called update_a_counter and implement the necessary code that will make sure the test case passes.

```

===== test session starts =====
platform linux -- Python 3.12.6, pytest-8.3.3, pluggy-1.5.0 -- /usr/bin/python
cachedir: .pytest_cache
rootdir: /home/kusa/School/472/assignments/labs/testing/tdd
configfile: pytest.ini
plugins: anyio-4.4.0, cov-5.0.0
collected 3 items

tests/test_counter.py::TestCounterEndpoints::test_create_a_counter PASSED [ 33%]
tests/test_counter.py::TestCounterEndpoints::test_duplicate_a_counter PASSED [ 66%]
tests/test_counter.py::TestCounterEndpoints::test_update_a_counter PASSED [100%]

----- coverage: platform linux, python 3.12.6-final-0 -----
Name                Stmts   Miss  Cover   Missing
-----
src/__init__.py         0      0  100%
src/counter.py        16      0  100%
src/status.py          6      0  100%
-----
TOTAL                  22      0  100%

===== 3 passed in 1.84s =====

```

We run pytest after implementing the code and see that we are now passing our previous test case.

Test read a counter.

```

def test_read_a_counter(self, client):
    #create a counter
    result = client.post('/counters/some_counter')
    assert result.status_code == status.HTTP_201_CREATED

    #hold counter
    counter = result.json['some_counter']

    #read from counter
    counter_name = "some_counter"
    result2 = client.get(f'/counters/{counter_name}')
    assert result2.status_code == status.HTTP_200_OK

    #compare the counter we are reading with the original counter
    read_counter = result2.json[counter_name]
    assert counter == read_counter

```

To implement the read a counter method I first wrote the test case you can see above. I first created a counter and then stored the counter so I can compare with it later. I then call the read method and expect a 200 OK response. Afterwards I assert that the new counter I'm reading matches the counter we initially created.


```

tests/test_counter.py::TestCounterEndpoints::test_read_a_counter FAILED [100%]

===== FAILURES =====
----- TestCounterEndpoints.test_read_a_counter -----

self = <tests.test_counter.TestCounterEndpoints object at 0x7953366394f0>
client = <FlaskClient <Flask 'src.counter'>>

    def test_read_a_counter(self, client):
        #create a counter
        result = client.post('/counters/some_counter')
        assert result.status_code == status.HTTP_201_CREATED

        #hold counter
        counter = result.json['some_counter']

        #read from counter
        counter_name = "some_counter"
        result2 = client.get(f'/counters/{counter_name}')
        assert result2.status_code == status.HTTP_200_OK
>
E       assert 405 == 200
E         + where 405 = <WrapperTestResponse streamed [405 METHOD NOT ALLOWED]>.status_code
E         + and   200 = status.HTTP_200_OK

tests/test_counter.py:71: AssertionError

----- coverage: platform linux, python 3.12.6-final-0 -----
Name                Stmts  Miss  Cover   Missing
-----
src/__init__.py        0      0   100%
src/counter.py        17      0   100%
src/status.py          6      0   100%
-----
TOTAL                  23      0   100%

===== short test summary info =====
FAILED tests/test_counter.py::TestCounterEndpoints::test_read_a_counter - assert 405 == 200
===== 1 failed, 3 passed in 1.41s =====

```

We run pytest after creating this test case and see that we are failing the test as expected.

```

@app.route('/counters/<name>', methods=['GET'])
def read_counter(name):
    ''' Read from counter '''
    app.logger.info(f"Request to read from counter: {name}")
    global COUNTERS
    return {name: COUNTERS[name]}, status.HTTP_200_OK

```

After I was able to get red test result I worked on making this method in counter.py. I basically just read a counter by it's name that was passed and returned it back with the proper response.


```

===== test session starts =====
platform linux -- Python 3.12.6, pytest-8.3.3, pluggy-1.5.0 -- /usr/bin/python
cachedir: .pytest_cache
rootdir: /home/kusa/School/472/assignments/labs/testing/tdd
configfile: pytest.ini
plugins: anyio-4.4.0, cov-5.0.0
collected 4 items

tests/test_counter.py::TestCounterEndpoints::test_create_a_counter PASSED [ 25%]
tests/test_counter.py::TestCounterEndpoints::test_duplicate_a_counter PASSED [ 50%]
tests/test_counter.py::TestCounterEndpoints::test_update_a_counter PASSED [ 75%]
tests/test_counter.py::TestCounterEndpoints::test_read_a_counter PASSED [100%]

----- coverage: platform linux, python 3.12.6-final-0 -----
Name                Stmts  Miss  Cover   Missing
-----
src/__init__.py       0      0   100%
src/counter.py       20      0   100%
src/status.py         6      0   100%
-----
TOTAL                 26      0   100%

===== 4 passed in 1.35s =====

```

We run pytest after implement the code in account.py and see that we are now passing the previous test case we wrote.

Refactoring.

```

@app.route('/counters/<name>', methods=['PUT'])
def update_counter(name):
    ''' Update a counter '''
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS
    if name not in COUNTERS:
        return {"Message": f"Counter {name} does not exist"}, status.HTTP_404_NOT_FOUND
    COUNTERS[name] += 1
    return {name: COUNTERS[name]}, status.HTTP_200_OK

@app.route('/counters/<name>', methods=['GET'])
def read_counter(name):
    ''' Read from counter '''
    app.logger.info(f"Request to read from counter: {name}")
    global COUNTERS
    if name not in COUNTERS:
        return {"Message": f"Counter {name} does not exist"}, status.HTTP_404_NOT_FOUND
    return {name: COUNTERS[name]}, status.HTTP_200_OK

```

After getting green for our new tests that we wrote we can now refactor our code. I edited the counter.py file to add error checks in our new methods we created. I simply check if the name we are using doesn't exist and then return a 404 error. After running pytest again I noticed that there was missing lines numbers for the code I just added.

```
def test_fail_update_a_counter(self, client):
    result = client.put('/counters/identexist')
    assert result.status_code == status.HTTP_404_NOT_FOUND

def test_fail_read_a_counter(self, client):
    result = client.get('/counters/identexist')
    assert result.status_code == status.HTTP_404_NOT_FOUND
```

I added two new test cases in test_counter.py that target these new missing lines by making sure that I test for failed updates and reads.

```
===== test session starts =====
platform linux -- Python 3.12.6, pytest-8.3.3, pluggy-1.5.0 -- /usr/bin/python
cachedir: .pytest_cache
rootdir: /home/kusa/School/472/assignments/labs/testing/tdd
configfile: pytest.ini
plugins: anyio-4.4.0, cov-5.0.0
collected 6 items

tests/test_counter.py::TestCounterEndpoints::test_create_a_counter PASSED [ 16%]
tests/test_counter.py::TestCounterEndpoints::test_duplicate_a_counter PASSED [ 33%]
tests/test_counter.py::TestCounterEndpoints::test_update_a_counter PASSED [ 50%]
tests/test_counter.py::TestCounterEndpoints::test_read_a_counter PASSED [ 66%]
tests/test_counter.py::TestCounterEndpoints::test_fail_update_a_counter PASSED [ 83%]
tests/test_counter.py::TestCounterEndpoints::test_fail_read_a_counter PASSED [100%]

----- coverage: platform linux, python 3.12.6-final-0 -----
Name                Stmts   Miss  Cover   Missing
-----
src/__init__.py         0      0   100%
src/counter.py        24      0   100%
src/status.py          6      0   100%
-----
TOTAL                  30      0   100%
```

This image shows my final pytest results after running my refactored code.

Conclusion:

The results above showcase my code snippets for each test case written in this task. Overall this task of our lab was very informative and really showed the power of test driven development. I wrote test cases and then from those test cases built the necessary methods in my code. Using this approach we can quickly come up with code that meets our requirements and refactor to keep improving the code.

Conclusion

This lab report documented various tasks focused on software testing, providing detailed descriptions and results for each task. Testing is a critical component of the software development process, ensuring code functionality and reliability. Through the completion of these tasks, a deeper understanding of testing in software development was achieved.