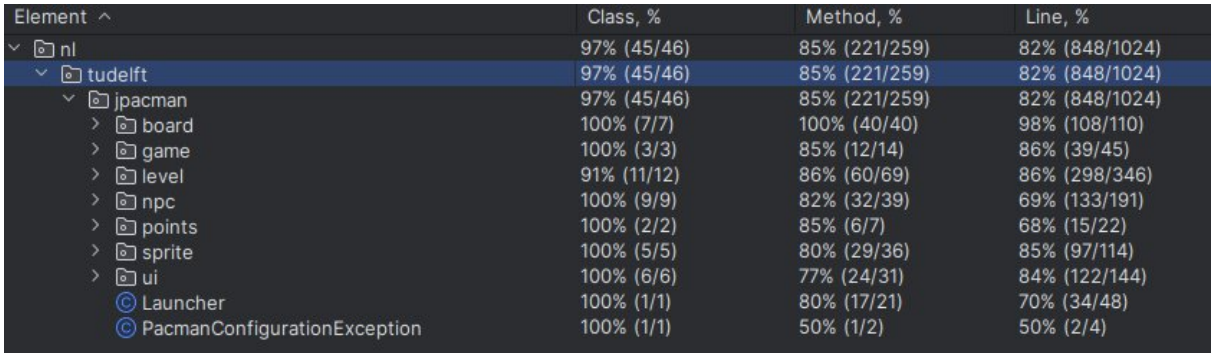
Dr Businge, Project Coordinator
Garrett Prentice, Project Manager
RE: Code Coverage & Evaluation (JaCoCo v IntelliJ)

I believe that the coverage results from IntelliJ are better for uses that prefer integrated environments. When I use IntelliJ, I use the built in code coverage as it provides me with all that I need and allows me to jump directly to editable source code. JaCoCo is a great substitute for users that do not use IntelliJ; VSCode, or Neovim.

Similarity
I have similar results between both JaCoCo and IntelliJ's coverage reporting values. However, IntelliJ gives me a line based breakdown of the code. JaCoCo gives me functional breakdown of the code. For example, in Figure 1 and 2, IntelliJ reports by line, method, and class. These are things that only require touching to change. But JaCoCo coverage is more focused on code branching and combination of input. For this reason, JaCoCo might be better for quality of test.

Figure 1: IntelliJ coverage report for JPacman repo



Element	Class, %	Method, %	Line, %
nl	97% (45/46)	85% (221/259)	82% (848/1024)
tudelft	97% (45/46)	85% (221/259)	82% (848/1024)
jpacman	97% (45/46)	85% (221/259)	82% (848/1024)
board	100% (7/7)	100% (40/40)	98% (108/110)
game	100% (3/3)	85% (12/14)	86% (39/45)
level	91% (11/12)	86% (60/69)	86% (298/346)
npc	100% (9/9)	82% (32/39)	69% (133/191)
points	100% (2/2)	85% (6/7)	68% (15/22)
sprite	100% (5/5)	80% (29/36)	85% (97/114)
ui	100% (6/6)	77% (24/31)	84% (122/144)
Launcher	100% (1/1)	80% (17/21)	70% (34/48)
PacmanConfigurationException	100% (1/1)	50% (1/2)	50% (2/4)

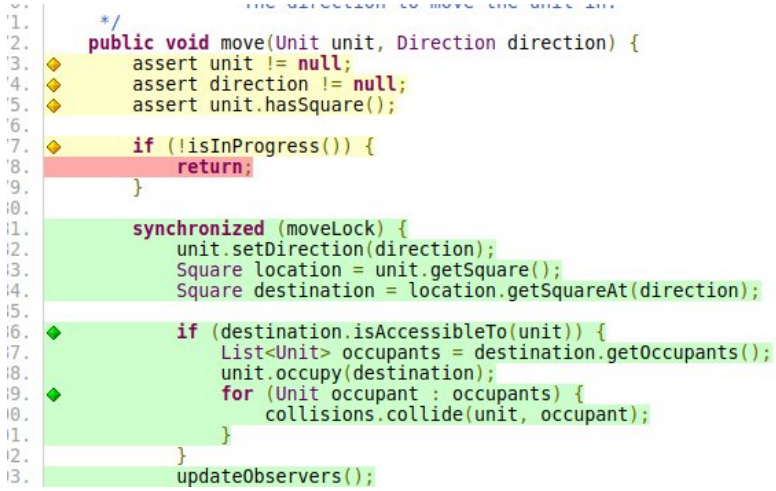
Figure 2: JaCoCo coverage report for JPacman repo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.tudelft.jpacman.npc.ghost		71%		55%	56	105	43	181	5	34	0	8
nl.tudelft.jpacman.level		85%		69%	51	155	37	344	7	69	1	12
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%		58%	44	93	2	110	0	40	0	7
nl.tudelft.jpacman.sprite		86%		59%	30	70	11	113	5	38	0	5
nl.tudelft.jpacman		71%		25%	11	30	16	52	5	24	0	2
nl.tudelft.jpacman.points		60%		75%	1	11	5	21	0	9	0	2
nl.tudelft.jpacman.game		89%		60%	9	24	2	45	1	14	0	3
nl.tudelft.jpacman.npc		100%		n/a	0	4	0	8	0	4	0	1
Total	963 of 4,694	79%	273 of 637	57%	268	590	158	1,039	35	268	2	47

As shown above, JaCoCo reports a lower code coverage than IntelliJ. In JaCoCo, it reports only 79% for missed instructions, which is closest to line coverage in IntelliJ, which is reported as 82%. For a full reference to the changes that were made to the JPacman repository, see my fork of the github repository. (https://github.com/S1robe/jpacman)

Usefulness
I believe that seeing the source code in the browser is the most useful feature of JaCoCo. I can match this directly to my test cases without opening the project. For example, in Figure 3, I can navigate to a specific section of code that was not covered and then write test cases to cover it.

Figure 3: JaCoCo coverage of Level.java#move()



```
    */
    public void move(Unit unit, Direction direction) {
        assert unit != null;
        assert direction != null;
        assert unit.hasSquare();

        if (!isInProgress()) {
            return;
        }

        synchronized (moveLock) {
            unit.setDirection(direction);
            Square location = unit.getSquare();
            Square destination = location.getSquareAt(direction);

            if (destination.isAccessibleTo(unit)) {
                List<Unit> occupants = destination.getOccupants();
                unit.occupy(destination);
                for (Unit occupant : occupants) {
                    collisions.collide(unit, occupant);
                }
            }
            updateObservers();
```

As shown above, the code in yellow is assertions and branches that were checked. The section shown in red is branch that has not yet been covered, and the green is what has been covered.

Preference

I have taken this course at a different school, we used JaCoCo, however, when paired with IntelliJ it is not as powerful. But, when developing a report to senior developer or even as the project lead, it can make my life easy. Receiving a zip file with these test case reports is sometimes easier than going through the process of opening GitHub or IntelliJ to view test results.

Now that I use Neovim, a text editor IDE, it is much lighter weight, but test cases are harder to view. JaCoCo bundled with Gradle would be a solution to this problem without needing to integrate with a plugin or write any code. From a developers standing, I prefer IntelliJ. From a project manager perspective, I prefer JaCoCo.

JPacman Test Coverage

The following code snippets are from the JPacman repository. It is a pacman clone that is aimed at teaching how to unit test various parts of a program. Below is an acceptable test of the movement, map parser, and collision detection.

Figure 4: Test of Movement in JPacman

```
    @Test
    void testPlayerMove(){
        Player pacman = launcher.getGame().getPlayers().get(0);
        Square pacmanStartSpot = pacman.getSquare();
        launcher.getGame().getLevel().move(pacman, Direction.NORTH);
// Never moved cause upward collision
        Assertions.assertThat(pacmanStartSpot).isEqualTo(pacman.getSquare());
        launcher.getGame().getLevel().move(pacman, Direction.SOUTH);
// Never moved cause downward collision
        Assertions.assertThat(pacmanStartSpot).isEqualTo(pacman.getSquare());

        launcher.getGame().getLevel().move(pacman, Direction.EAST);
        launcher.getGame().getLevel().move(pacman, Direction.WEST);
// mvoed right so left should be same
        Assertions.assertThat(pacmanStartSpot).isEqualTo(pacman.getSquare());
        launcher.getGame().getLevel().move(pacman, Direction.WEST);
        launcher.getGame().getLevel().move(pacman, Direction.EAST);
// moved right back to start spot should be same as start.
        Assertions.assertThat(pacmanStartSpot).isEqualTo(pacman.getSquare());
    }
```

As shown above in figure 4, the players movement is being test on the standard map. In this configuration there are walls directly above and below, but not the sides. Since player exact position is not exposed, but the square that the player is on is. As a result we can compare the square the player starts on *'pacmanStartSpot'* and compare it to the players current position '*pacman.getSquare()*'. Because there are walls directly above and below the player moving up (*north*) or down (*south*) should not change the players position. Similarly, moving left (*west*) or right (*east*) will change the players position.

Figure 5: Test of Map Parser in JPacman

```
//Load simple map, should load and throw no error
@Test
void testLoadSimpleMap() {
    Assertions.assertThatCode(() -> mp.parseMap("/simplemap.txt"))
        .doesNotThrowAnyException();
}

//Empty maps are not allowd, should throw error
@Test
void testLoadEmptyMap(){
    Assertions.assertThatThrownBy(() -> mp.parseMap("/emptyMap.txt"))
        .isInstanceOf(PacmanConfigurationException.class);
}

//A different non-std map, should not error
@Test
void testLoadCertainDeathMap() {
    Assertions.assertThatCode(() -> mp.parseMap("/certainDeathMap.txt"))
        .doesNotThrowAnyException();
}

//Map doesnt exist, should error.
@Test
void testNonExistentMap(){
    Assertions.assertThatThrownBy(() -> mp.parseMap("/IdontExistMap.txt"))
        .isInstanceOf(PacmanConfigurationException.class);
}

//Map doesnt have a player should not error.
@Test
void testMapMissingPlayer(){
    Assertions.assertThatCode(() -> mp.parseMap("/missingPlayer.txt"))
        .doesNotThrowAnyException();
}
```

Above, in figure 5, is a test of the map parser used to load and set the position of the player and npc's (ghosts). The map if empty should produce an exception, as it should if the map does not exist. Test cases 2 and 4, in figure 5, show how this can be tested for using JUnit. The remaining tests account for situations where the map is non-standard, and is missing a player. Test cases 1, 3, and 5, are intentional behavior.

Figure 6: Test of Collision with walls, coins, and a ghost

```
    @Test
    void collideCoinThenWallThenGhost(){
 // Should collide with coin
        customLevel.move(pacman, Direction.WEST);
// coins add 10 to score, since this is the first one, score should be 10
        Assertions.assertThat(pacman.getScore()).isEqualTo(10);

        Square wall = pacman.squaresAheadOf(1);
        customLevel.move(pacman, Direction.WEST);
 // Square should be the same, it is unchanged.
        Assertions.assertThat(pacman.squaresAheadOf(1)).isEqualTo(wall);

//continue marching right, to the ghost.
        ScheduledExecutorService timer = Executors.newScheduledThreadPool(1);
        timer.scheduleAtFixedRate(() -> {
            if (pacman.isAlive()){
                customLevel.move(pacman, Direction.EAST);
            } else {
                timer.shutdown();
            }
        }, 0, 100, TimeUnit.MILLISECONDS);

// This will fail if the task is interrupted.
        Assertions.assertThatCode(() -> {
            if (timer.awaitTermination(1, TimeUnit.SECONDS))
                timer.shutdown();
            }).doesNotThrowAnyException();

// End of game should be caused by interaction with ghost.
        Assertions.assertThat(pacman.getKiller()).isInstanceOf(Ghost.class);
    }
```

The final test was of the collisions between the player (pacman) and walls, coins, and ghosts. First the player moves left, which causes the player in this map to pick up a coin. Coins add 10 to the players score, so the score should be 10 if a coin was picked up. The player is also against the left wall, and so attempting to move left should be prevented by the wall. Then the player is moved right-ward until eventually bumping into a ghost causing the game to end, which verifies that the collision occurred.

Python Test Coverage

This segment of the report is regarding the ‘test_coverage’ repository provided to teach testing practices in python using ‘nosetests’. In this example, a model account is provided which will be tested. Below is snippets of each various test cases from the project. Included is a test of all the non-class methods provided within models/account.py: to_dict, from_dict, create, update, delete. For a full reference to the changes that were made to the repository, see my fork of the github repository. (https://github.com/S1robe/CS472-test_coverage)

Figure 7: Test of Account#create()

```
def test_create_an_account(self):
    """ Test Account creation using known data """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    account.create()
    self.assertEqual(len(Account.all()), 1)
```

The method Account#create is responsible for creating an account. By default the Account database is empty, as shown in figure 7. After this method is executed there should be exactly one (1) account in it.

Figure 8: Test of Account#__repr__()

```
def test_repr(self):
    """Test the representation of an account"""
    account = Account()
    account.name = "Foo"
    self.assertEqual(str(account), "<Account 'Foo'>")
```

The method Account#__repr__ is responsible for generating a representation of an account. In this case, shown in figure 8, the proper response when called, is the type (Account), and its name, which as tested above in figure 8, should return “<Account ‘Foo’>”.

Figure 9: Test of Account#to_dict()

```
def test_to_dict(self):
    """ Test account to dict """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    result = account.to_dict()
    self.assertEqual(account.name, result["name"])
    self.assertEqual(account.email, result["email"])
    self.assertEqual(account.phone_number, result["phone_number"])
    self.assertEqual(account.disabled, result["disabled"])
    self.assertEqual(account.date_joined, result["date_joined"])
```

The method Account#to_dict is responsible for turning an account into a dictionary. By comparing the results of the conversion to the original we are able to find they are the same.

Figure 10: Test of Account#from_dict()

```
def test_from_dict(self):
    """Test account from dict """
    data = ACCOUNT_DATA[self.rand] # get a random account
    account = Account(**data)
    result = account.to_dict()
    resultAccount = Account()
    resultAccount.from_dict(result)
    self.assertEqual(account.name, resultAccount.name)
    self.assertEqual(account.email, resultAccount.email)
    self.assertEqual(account.phone_number, resultAccount.phone_number)
    self.assertEqual(account.disabled, resultAccount.disabled)
    self.assertEqual(account.date_joined, resultAccount.date_joined)
```

The method Account#from_dict() is responsible for turning an account stored as a dictionary back into an Account object. By comparing an account that was made from this method with another that was created prior we can prove that the two accounts are equal.

Figure 11: Test of Account#update()

```
    # Modify some fields
    result.disabled = not account.disabled
    result.name = "Test" + account.name
    newName = result.name
    result.id = randnum
    # Committ the changes
    result.update()
    # Reload the account
    result = Account.query.get(randnum)
    # Check for changes.
    self.assertEqual(result.name, newName)
    self.assertEqual(result.name, account.name)
    self.assertNotEqual(oldname, newName)
    self.assertEqual(account.disabled, result.disabled)
```

Figure 12: Test of Account#update() fail

```
ef test_update_fail(self):
    """Test account update DataValidationError"""
    account = Account(**ACCOUNT_DATA[self.rand])
    account.name = "NewName"
    with self.assertRaises(DataValidationError):
        account.update()
```

The method Account#update is responsible for committing changes to an account. It will fail if the id is not set. Figures 11 and 12 display the successful test and the fail test. In the first test the id used to retrieve the account is saved so that it can be updated later.

Figure 13: Test of Account#delete()

```
    randnum = self.rand
    account = Account.query.get(randnum)
    account.delete()
    self.assertIsNone(Account.query.get(randnum))
```

The method Account#delete will delete the account it is called on by removing it from the database and deleting its reference. As shown in figure 13, the account must not be in the database after it is deleted.

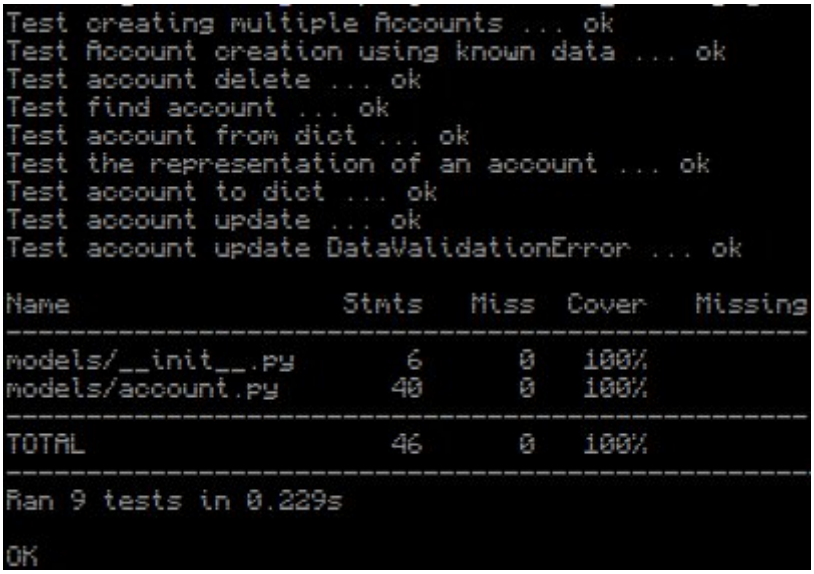
Figure 14: Test of Account#find()

```
randNum = self.rand
account = Account.find(randNum)
result = ACCOUNT_DATA[(randNum-1)]

self.assertEqual(account.name, result["name"])
self.assertEqual(account.email, result["email"])
self.assertEqual(account.phone_number, result["phone_number"])
self.assertEqual(account.disabled, result["disabled"])
```

The method Account#find will attempt to find the account specified. It is important to note that the dictionary that the accounts are loaded from initially starts at 0 and so the reference numbers are off by 1. To adjust for this we subtract 1 from the reference number ‘randnum’ to use the correct reference number.

Figure 15: Test coverage report



Shown above in figure 15 are the results of executing the tests shown previously in figures 7 through 14. In this case 100% test coverage was achieved. This does not necessarily mean that the code is complete, but does indicate the this code is well tested.

Test Driven Development

This section of the report contains how the test driven development process could go. Test driven development begins by making tests that then define the behavior of the program. By writing tests and then writing code to fulfill those tests the program is tested during development which improves code quality. Below is each method and then how it was tested. For a full reference to the changes that were made to the repository, see my fork of the github repository. (https://github.com/S1robe/CS472-tdd)

Figure 16: Flask app method create_counter

```
@app.route('/counters/<name>', methods=['POST'])
def create_counter(name):
    """Create a counter"""
    app.logger.info(f"Request to create counter: {name}")
    global COUNTERS
    if name in COUNTERS:
        return {"Message": f"Counter {name} already exists"}, status.HTTP_409_CONFLICT
    COUNTERS[name] = 0
    return {name: COUNTERS[name]}, status.HTTP_201_CREATED
```

The method create_counter as shown in figure 16 should create a counter when invoked. It requires that the full name be given in the form “/counters/<name>”. The tests that this method were based on are as follows in figure 17.

Figure 17: Flash app method create_counter tests

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

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

As shown in Figure 17, the development of the method create_counter is provided by the result of the ‘assert’ statements. Each of these statements represents a return statement in Figure 16. The method can be built backwards from the return statements to produce a method that is quality and fulfills the requirements of the project. A similar process is employed to develop the following methods in Figures 18 and 20.

Figure 18: Flask app method update_counter

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

The method update_counter should update the counter that is provided in “counters/<name>”. This method returns a ‘not found’ status if the counter requested is not found and responds ‘OK’ otherwise.

Figure 19: Flask app method update_counter test

```
def test_update_a_counter(self):
    """It should update a counter"""
    name = "cntr"
    fqdn = "/counters/" + name
    result = self.client.post(fqdn)  # Create
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    baseline = result.get_json()[name]      # Result
    result = self.client.put(fqdn)  # Update
    self.assertEqual(result.status_code, status.HTTP_200_OK)  # 200
    self.assertEqual(baseline+1, result.get_json()[name])

def test_update_a_counter_fail(self):
    """It should return 404, update nonexist-counter"""
    result = self.client.put("/counters/IDontExist")  # Update
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
```

These tests shown in figure 19 cover all possible inputs for the ‘update_counter’ method. The first method definition covers the successful invocation. The tested application returns a ‘created’ response because the counter must first be created, and then updated. The second method returns a ‘not found’ because the counter does not exist yet.

Figure 20: Flask app method get_counter

```
@app.route('/counters/<name>', methods=['GET'])
def get_counter(name: str):
    """Get a counter value"""
    app.logger.info(f"Request counter value for: {name}")
    if name in COUNTERS:
        return {name: COUNTERS[name]}, status.HTTP_200_OK
    return {"Message": f"Counter {name} does not exist"}, status.HTTP_404_NOT_FOUND
```

The method get_counter should retrieve the value of the counter if it exists, otherwise it should report that the counter is not found.

Figure 21: Flask app method get_counter tests

```
def test_get_a_counter(self):
    """It should read the counter"""
    result = self.client.post("/counters/cnt")
    self.assertEqual(result.status_code, status.HTTP_201_CREATED)
    result = self.client.get("/counters/cnt")
    self.assertEqual(result.status_code, status.HTTP_200_OK)
    self.assertEqual(0, result.get_json()["cnt"])

def test_get_a_counter_fail(self):
    """It should 404 when reading non-existent counter"""
    result = self.client.get("/counters/IDontExist")
    self.assertEqual(result.status_code, status.HTTP_404_NOT_FOUND)
```

The tests shown in figure 21, demonstrate both the successful retrieval of the counter's value, after it has been created with a create_counter. The second method "test_get_a_counter_fail" shows how the method should respond if the counter does not exist.

Figure 22: Test coverage from test driven development

```
It should create a counter ... ok
It should return an error for duplicates ... ok
It should read the counter ... ok
It should 404 when reading non-existent counter ... ok
It should update a counter ... ok
It should return 404, update nonexist-counter ... ok

Name              Stmts   Miss  Cover   Missing
-----------------------------------------------
src/counter.py       24      0   100%
src/status.py         6      0   100%
-----------------------------------------------
TOTAL                30      0   100%
-----------------------------------------------------------
Ran 6 tests in 0.102s

OK
```

Finally, figure 22 shows 100% coverage of test driven development. This again indicates that the code is well tested, but not necessarily complete. If you have any questions regarding these methods, their reliability, or completeness, please reach out.