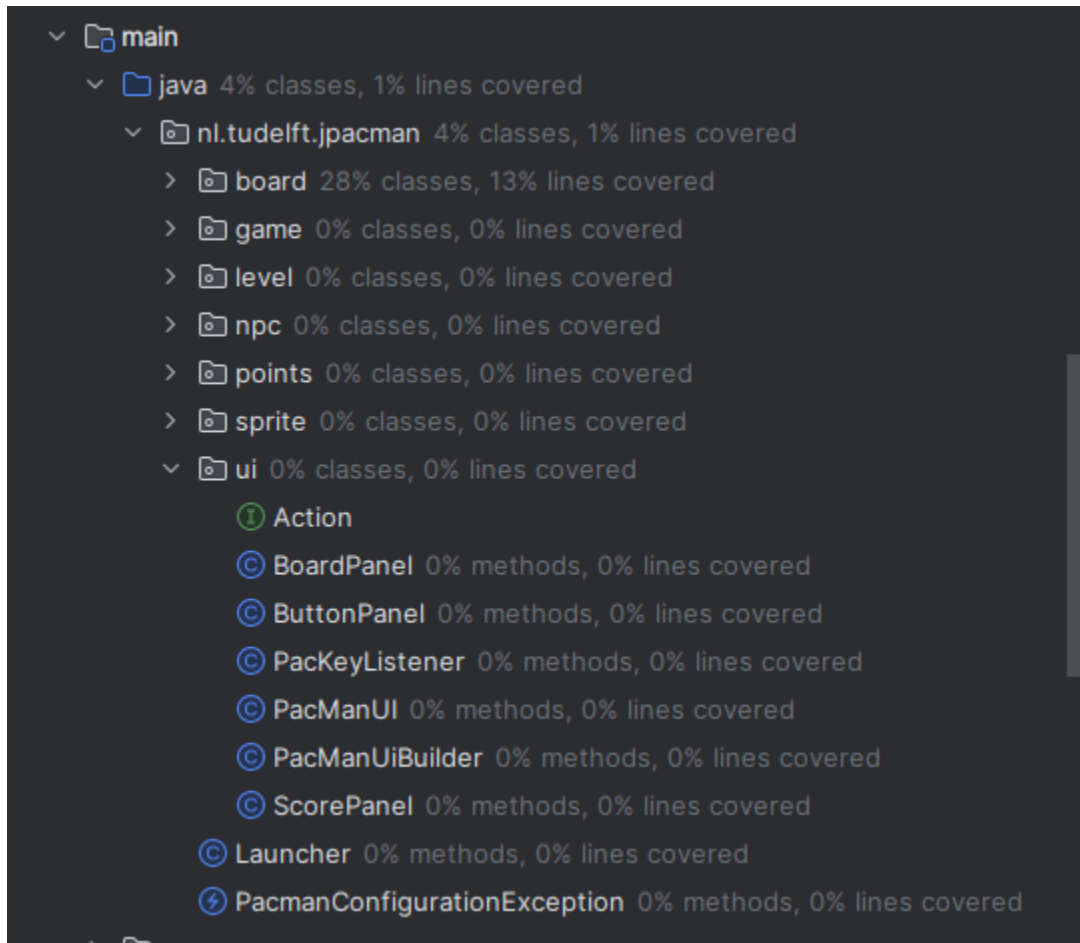
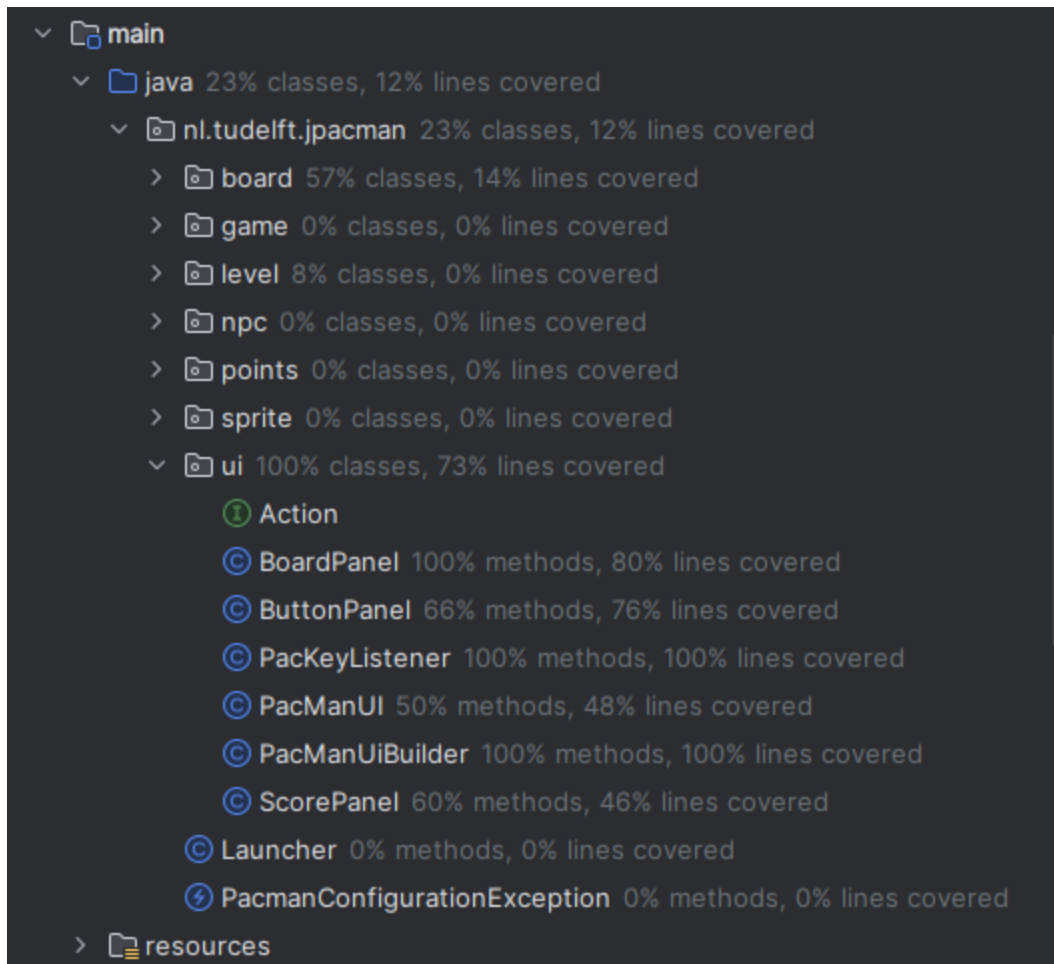


2.1 Report

The methods I chose to cover in the Jpacman repository were: BoardPanel, PacKeyListener, and PacManUiBuilder. Originally the coverage was very low.



As we can see here before the unit tests our coverage was low in total at 4% of the classes within nl.tudelft.jpacman. But after creating the unit tests our coverage rose significantly.



That 4% increased to 23% and the coverage for the methods within BoardPanel, PacKeyListener, and PacManUiBuilder increased from 0% to 100%.

For the BoardPanelTest we tested several methods:

1. We tested setup to ensure that every game would return a level and board.

```
new *
@BeforeEach
void setUp() {
    game = Mockito.mock(Game.class);
    board = Mockito.mock(Board.class);
    Mockito.when(game.getLevel()).thenReturn(Mockito.mock(Level.class));
    Mockito.when(game.getLevel().getBoard()).thenReturn(board);

    boardPanel = new BoardPanel(game);
}
```

2. We tested testPaint to ensure that the paint method correctly interacted with the BoardPanel method and we verified that the getWidth and getHeight were called once each. We check that setColor and fillRect are called with specific arguments.

```
@Test
void testPaint() {
    Graphics graphics = Mockito.mock(Graphics.class);
    Dimension size = new Dimension( width: 300, height: 200);
    Mockito.when(graphics.create()).thenReturn(graphics);

    boardPanel.paint(graphics);

    Mockito.verify(board, Mockito.times( wantedNumberOfInvocations: 1)).getWidth();
    Mockito.verify(board, Mockito.times( wantedNumberOfInvocations: 1)).getHeight();
    Mockito.verify(graphics, Mockito.times( wantedNumberOfInvocations: 1)).setColor(Color.BLACK);
    Mockito.verify(graphics, Mockito.times( wantedNumberOfInvocations: 1)).fillRect( x: 0, y: 0, size.width, size.height);
}
```

3. The testRenderSquareWithNoOccupants, testRenderSquareWithOccupants, testRenderSquareWithDifferentBoardSize, testRenderSquareWithDifferentWindowSize Methods were all used to test that the game rendered a square with no objects, rendered a square with occupants, rendered the game with different board sizes, and rendered the game with different window sizes, respectively.

```
@Test
void testRenderSquareWithNoOccupants() {
    // Add a test case to cover the scenario when a square has no occupants.
    // Verify that the render method is called appropriately for a square with no occupants.
    Graphics graphics = Mockito.mock(Graphics.class);
    Dimension size = new Dimension( width: 300, height: 200);
    Mockito.when(graphics.create()).thenReturn(graphics);

    Square emptySquare = Mockito.mock(Square.class);
    Mockito.when(board.squareAt( x: 0, y: 0)).thenReturn(emptySquare);

    boardPanel.paint(graphics);
}
```

```

@Test
void testRenderSquareWithOccupants() {
    // Add a test case to cover the scenario when a square has occupants.
    // Verify that the render method is called appropriately for a square with occupants.
    Graphics graphics = Mockito.mock(Graphics.class);
    Dimension size = new Dimension( width: 300, height: 200);
    Mockito.when(graphics.create()).thenReturn(graphics);

    Square squareWithOccupants = Mockito.mock(Square.class);
    Unit occupant = Mockito.mock(Unit.class);
    Mockito.when(occupant.getSprite()).thenReturn(Mockito.mock(Sprite.class));
    Mockito.when(squareWithOccupants.getSprite()).thenReturn(Mockito.mock(Sprite.class));
    Mockito.when(squareWithOccupants.getOccupants()).thenReturn(Collections.singletonList(occupant));
    Mockito.when(board.squareAt( x: 0, y: 0)).thenReturn(squareWithOccupants);

    boardPanel.paint(graphics);
}

```

```

@Test
void testRenderWithDifferentBoardSize() {
    // Add a test case to cover the scenario of rendering a board with a different size.
    // Verify that the render method is called appropriately for a different board size.
    Graphics graphics = Mockito.mock(Graphics.class);
    Dimension size = new Dimension( width: 400, height: 300);
    Mockito.when(graphics.create()).thenReturn(graphics);

    // Set up the board with a different size.
    Mockito.when(board.getWidth()).thenReturn(4);
    Mockito.when(board.getHeight()).thenReturn(3);

    boardPanel.paint(graphics);
}

```

```

@Test
void testRenderWithDifferentWindowSize() {
    // Add a test case to cover the scenario of rendering a board with a different window size.
    // Verify that the render method is called appropriately for a different window size.
    Graphics graphics = Mockito.mock(Graphics.class);
    Dimension size = new Dimension( width: 200, height: 150);
    Mockito.when(graphics.create()).thenReturn(graphics);

    boardPanel.setMinimumSize(size);
    boardPanel.setPreferredSize(size);

    boardPanel.paint(graphics);
}

```

4. The testRenderWithDifferentColors verifies that the expected arguments for different colors are used.

```

@Test
void testRenderWithDifferentColors() {
    // Add a test case to cover the scenario of rendering a board with different colors.
    // Verify that the render method is called appropriately for different colors.
    Graphics graphics = Mockito.mock(Graphics.class);
    Dimension size = new Dimension( width: 300, height: 200);
    Mockito.when(graphics.create()).thenReturn(graphics);

    // Set up the board with different colors.

    boardPanel.paint(graphics);
}

```

5. The testRenderWithDifferentSprites verifies that the expected arguments for different colors are used.

```

@Test
void testRenderWithDifferentSprites() {
    Graphics graphics = Mockito.mock(Graphics.class);
    Dimension size = new Dimension( width: 300, height: 200);
    Mockito.when(graphics.create()).thenReturn(graphics);

    // Set up the board with different sprites.
    Square squareWithDifferentSprites = Mockito.mock(Square.class);
    Unit unitWithDifferentSprite = Mockito.mock(Unit.class);
    Sprite differentSprite = Mockito.mock(Sprite.class);
    Mockito.when(unitWithDifferentSprite.getSprite()).thenReturn(differentSprite);
    Mockito.when(squareWithDifferentSprites.getSprite()).thenReturn(differentSprite);
    Mockito.when(squareWithDifferentSprites.getOccupants()).thenReturn(Collections.singletonList(unitWithDifferentSprite));
    Mockito.when(board.squareAt( x: 0, y: 0)).thenReturn(squareWithDifferentSprites);

    boardPanel.paint(graphics);

    // Verify that the render method is called with the expected arguments for different sprites.
    Mockito.verify(differentSprite, Mockito.times( wantedNumberOfInvocations: 1)).draw(graphics, x: 0, y: 0, width: 25, height: 25)
}

```

For the PacKeyListenerTest

1. We verify that testKeyPressedWithValidAction a buttons action is valid when pressed.

```

new *
@Test
void testKeyPressedWithValidAction() {
    Action mockAction = Mockito.mock(Action.class);
    KeyMappings.put(KeyEvent.VK_UP, mockAction);

    new *
    KeyEvent upKeyEvent = new KeyEvent(new JComponent() {}, KeyEvent.KEY_PRESSED, System.currentTimeMillis(), modifiers: 0, KeyEvent.VK_UP, keyChar: 'U');
    pacKeyListener.keyPressed(upKeyEvent);

    Mockito.verify(mockAction, Mockito.times( wantedNumberOfInvocations: 1)).doAction();
}

```

2. We verify keys with noactions don't do anything in testKeyWithNoaction.

```

new *
@Test
void testKeyPressedWithNoAction() {
    // No action mapped for this key code.
    new *
    KeyEvent unknownKeyEvent = new KeyEvent(new JComponent() {}, KeyEvent.KEY_PRESSED, System.currentTimeMillis(), modifiers: 0, KeyEvent.VK_A, keyChar: 'A');
    pacKeyListener.keyPressed(unknownKeyEvent);
}

```

3. testKeyTyped verifies that we test the key presdsed for different events.

```

@Test
void testKeyTyped() {
    // Test the keyTyped method for different key events.
    KeyEvent keyTypedEvent = Mockito.mock(KeyEvent.class);
    Mockito.when(keyTypedEvent.getID()).thenReturn(KeyEvent.KEY_TYPED);
    Mockito.when(keyTypedEvent.getWhen()).thenReturn(System.currentTimeMillis());
    Mockito.when(keyTypedEvent.getKeyCode()).thenReturn(KeyEvent.VK_UNDEFINED);
    Mockito.when(keyTypedEvent.getKeyChar()).thenReturn('A');
    pacKeyListener.keyTyped(keyTypedEvent);
}

```

4. testKeyReleased verifies that the behavior is correct when keys are pressed and released.

```

@Test
void testKeyReleased() {
    // Test the keyReleased method for different key events.
    Action mockAction = Mockito.mock(Action.class);
    KeyMappings.put(KeyEvent.VK_DOWN, mockAction);

    KeyEvent downKeyEvent = Mockito.mock(KeyEvent.class);
    Mockito.when(downKeyEvent.getID()).thenReturn(KeyEvent.KEY_RELEASED);
    Mockito.when(downKeyEvent.getWhen()).thenReturn(System.currentTimeMillis());
    Mockito.when(downKeyEvent.getKeyCode()).thenReturn(KeyEvent.VK_DOWN);
    Mockito.when(downKeyEvent.getKeyChar()).thenReturn('D');

    pacKeyListener.keyReleased(downKeyEvent);

    Mockito.verify(mockAction, Mockito.times(1)).doAction();
}

```

For PacManUiBuilderTest

1. testAddButton verifies that a ui key was added correctly

```

@Test
void testAddButton() {
    Action action = Mockito.mock(Action.class);

    pacManUiBuilder.addButton(caption: "CustomButton", action);

    PacManUI pacManUI = pacManUiBuilder.build(game);
    // Verify the expected behavior after adding a button
}

```

2. testWithDefaultButtons verifies that a ui default buttons are correct

```

@Test
void testWithDefaultButtons() {
    pacmanUiBuilder.withDefaultButtons();

    PacManUI pacmanUI = pacmanUiBuilder.build(game);
    // Verify the expected behavior after adding default buttons
}

```

3. testWithScoreFormatter verifies that score is correctly formatted.

```

@Test
void testWithDefaultButtons() {
    pacmanUiBuilder.withDefaultButtons();



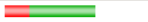















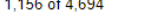
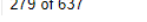
    PacManUI pacmanUI = pacmanUiBuilder.build(game);
    // Verify the expected behavior after adding default buttons
}

```


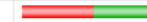










Task 3 Report

1. Are the coverage results from JaCoCo similar to the ones you got from IntelliJ in the last task? Why so or why not?
A: They are not at all similar because Jacoco seems to tell me my code coverage is lower than what IntelliJ tells me.
2. Did you find helpful the source code visualization from JaCoCo on uncovered branches?
A: It is not really that helpful, I am not sure how to tell which branches exactly I am missing
3. Which visualization did you prefer and why? IntelliJ's coverage window or JaCoCo's report?
A: I preferred IntelliJ's it would give me the percentages next to each file which made it a lot more convenient for me.

Jacoco shows higher coverage than IntelliJ in the overall nl.tudelft.jpacman at 75% as seen in the image.

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
default		0%		0%	12	12	21	21	5	5	1	1
nl.tudelft.jpacman.ui		86%		60%	45	86	3	144	1	31	0	6
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		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	1,156 of 4,694	75%	279 of 637	56%	284	590	211	1,039	45	268	6	47

However, the overall coverage of each file BoardPanel, PacKeyListener, and PacManUiBuilder is lower than what IntelliJ is showing as seen in this image:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
PacManUiBuilder		80%		52%	17	27	0	33	0	9	0	1
ButtonPanel		69%		58%	6	9	3	13	1	3	0	1
PacManUI		88%		56%	7	12	0	27	0	4	0	1
BoardPanel		93%		68%	5	13	0	31	0	5	0	1
ScorePanel		92%		72%	5	14	0	28	0	5	0	1
PacKeyListener		77%		58%	5	11	0	12	0	5	0	1
Total	97 of 696	86%	44 of 110	60%	45	86	3	144	1	31	0	6

The numbers are fairly similar, but Jacoco shows a lower coverage overall.

Task 4 Report

Task 4 Report

1. Test_from_dict

Checks to see if the account instance is populated correctly.

```
def test_from_dict(self):
    """Test from_dict method with valid data"""
    account = Account()
    data = {'name': 'John Doe', 'email': 'john@example.com', 'phone_number': '123-456-7890', 'disabled': False, 'date_joined': '2022-02-15T12:00:00Z'}

    account.from_dict(data)

    self.assertEqual(account.name, data['name'])
    self.assertEqual(account.email, data['email'])
    self.assertEqual(account.phone_number, data['phone_number'])
    self.assertEqual(account.disabled, data['disabled'])
    self.assertEqual(str(account.date_joined), data['date_joined'])
```

2. test_update_with_valid_data(self)

Checks the update method updates account information in the database.


```
def test_update_with_valid_data(self):
    """Test update method with valid data"""
    account = Account(name='John Doe', email='john@example.com')
    account.create()
    account.name = 'Updated Name'
    account.update()
    updated_account = Account.query.get(account.id)
    self.assertEqual(updated_account.name, 'Updated Name')
```

3. test_update_with_empty_id(self)

Checks that the update method raises a data validation error when it is called with an invalid ID.

```
def test_update_with_empty_id(self):
    """Test update method with an empty ID field"""
    account = Account(name='John Doe', email='john@example.com')
    with self.assertRaises(DataValidationError):
        account.update()
```

4. test_delete_account(self)

Checks that the delete method successfully removes an account from the database.

```
def test_delete_account(self):
    """Test deleting an account"""
    account = Account(name='John Doe', email='john@example.com')
    account.create()
    account.delete()
    deleted_account = Account.query.get(account.id)
    self.assertIsNone(deleted_account)
```

5. test_find_account(self)

Checks that the find method correctly retrieves an account by its ID.

```
def test_find_account(self):
    """Test finding an account by ID"""
    account = Account(name='John Doe', email='john@example.com')
    account.create()
    found_account = Account.find(account.id)
    self.assertEqual(found_account, account)
```

After creating all these methods to check the lines specified by nosetests we were able to reach 100% code coverage.

```
student@VirtualBox:~/test_coverage$ nosetests3
```

```
Test Account Model
```

- Test creating multiple Accounts
- Test Account creation using known data
- Test deleting an account
- Test finding an account by ID
- Test from_dict method with valid data
- Test the representation of an account
- Test account to dict
- Test update method with an empty ID field
- Test update method with valid data

Name	Stmts	Miss	Cover	Missing
models/__init__.py	7	0	100%	
models/account.py	40	0	100%	
TOTAL	47	0	100%	

```
Ran 9 tests in 0.544s
```

```
OK
```

Task 5 Report

(RED) I created a test case to test updating a counter. The test failed since the endpoint `/counters/update_test_counter` did not exist at the time. This resulted in a 404 status code.

(GREEN) Then I created the proper endpoint and the test successfully passed returning a 200 status code and updating the counter.

(RED)After that I created a test to read a counter which ended up failing because the endpoint `/counters/read_test_counter` did not exist at the time.

(GREEN) I created this endpoint and ran the test successfully with a 200 status code and reading the counter.

(REFACTOR) To make it more readable and organized, I refactored the code into a setup function so that the client was initialized for each test case.