* **UMGC CMSC 495 6980**

**Phase 1 Unit Testing**

**Team Beta**: Napier, Ian, Villa, Giovanni, Turbyfill, Justin, Saint-Maxens, McIvor, and Hernandez, Jonathan

#### Import Statements:

* **import unittest**: Imports Python's built-in unit testing framework.
* **import pygame**: Imports the Pygame library for game development.
* **from pygame.locals**: Imports specific constants from Pygame like KEYDOWN, QUIT, K\_LEFT, K\_RIGHT, K\_ESCAPE, and K\_SPACE used for handling keyboard events.
* **from unittest.mock import patch, MagicMock**: Imports the patch function and MagicMock class from unittest.mock to facilitate the mocking of methods and creating mock objects during tests.
* **from Team\_Beta\_Phase\_1 import Ball, Paddle, Block, collision\_checker, initialize\_game, main, game\_over**: Imports specific classes and functions from a custom module, presumably containing the game logic.

#### Class Definitions:

* **Block Class**:
  + **\_\_init\_\_(self, x, y, width, height, color, points)**: Constructor to initialize a block with position, size, color, and points.
  + **get\_rect(self)**: Method to return the Pygame rectangle object representing the block's position and size.
* **Paddle Class**:
  + **\_\_init\_\_(self, x, y, width, height, speed, color)**: Constructor to initialize the paddle with position, size, speed, and color.
  + **update(self)**: Updates the paddle's position based on current movement settings.
  + **set\_movement(self, x\_fac)**: Sets the movement direction of the paddle based on user input.
  + **get\_rect(self)**: Returns the Pygame rectangle object representing the paddle's position and size.
* **Ball Class**:
  + **\_\_init\_\_(self, x, y, radius, speed, color)**: Constructor to initialize a ball with position, size, speed, and color.
  + **update(self)**: Updates the ball's position and handles screen edge collisions.
  + **reset(self)**: Resets the ball's position after losing a life.
  + **get\_rect(self)**: Returns the Pygame rectangle object representing the ball's position and size.

#### 

#### 

#### Function Definitions:

* **collision\_checker(rect, ball)**: Function that checks for collisions between two rectangles using Pygame's colliderect method, returning True for a collision and False otherwise.

#### Unit Testing Class:

* **class TestBreakoutGame(unittest.TestCase)**: Class for organizing the test cases, inheriting from unittest.TestCase.

#### Setup and Teardown Methods:

* **setUp(self)**: Method called before each test method; initializes Pygame, sets up the game window, and creates instances of the Ball, Paddle, and Blocks.
* **tearDown(self)**: Method called after each test method; cleans up by quitting Pygame.

#### Test Methods:

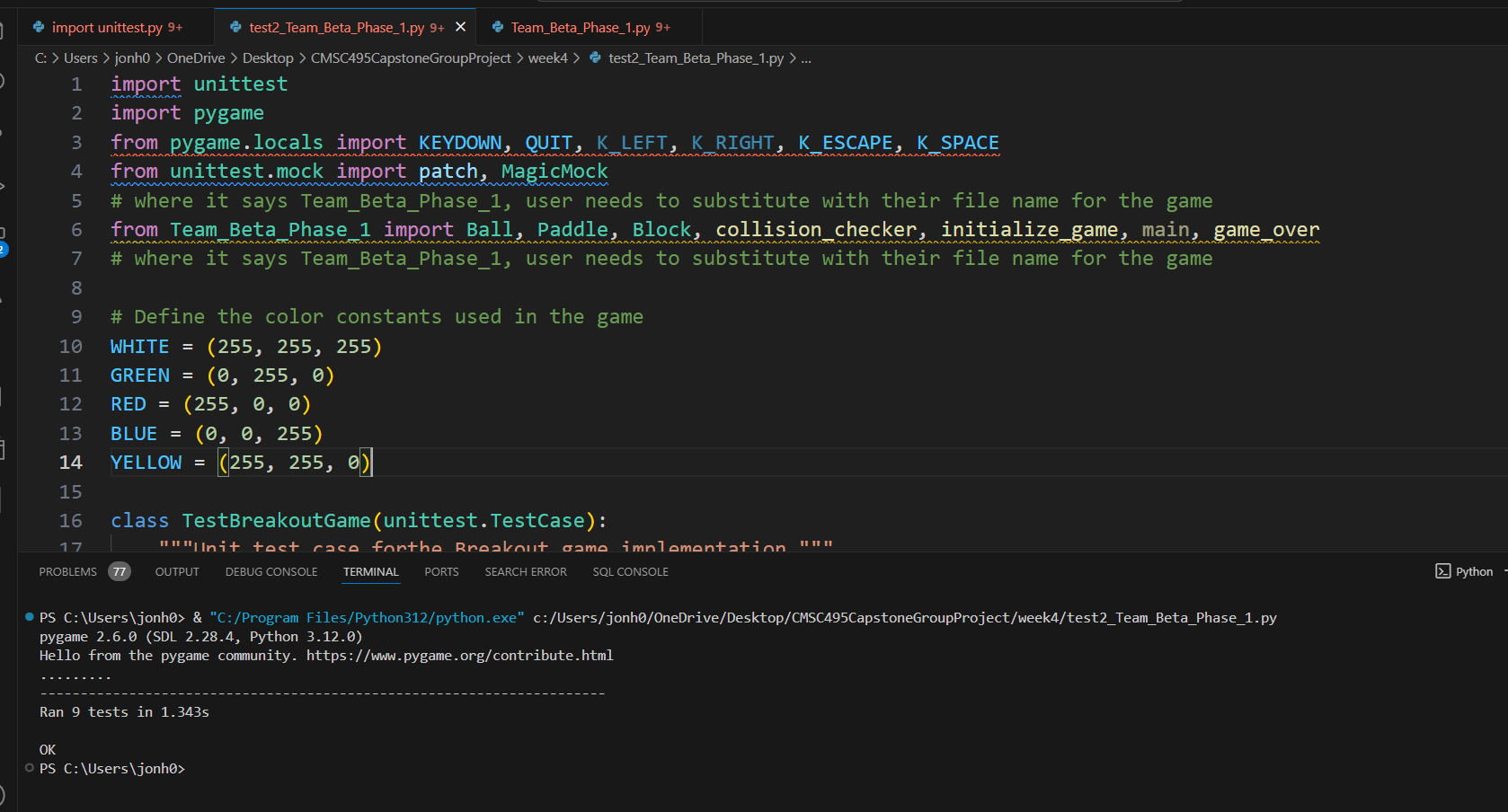
* **test\_ball\_movement(self)**: Tests that the ball moves as expected when updated, verifying changes in position.
* **test\_collision\_detection\_true(self)**: Tests that the collision\_checker function correctly identifies overlapping rectangles as colliding.
* **test\_collision\_detection\_false(self)**: Tests that the collision\_checker function correctly identifies non-overlapping rectangles as not colliding.
* **test\_paddle\_movement(self)**: Tests that the paddle moves correctly in response to simulated left and right key presses, verifying correct positional updates.
* **test\_game\_over\_display(self)**: Tests if the game over screen is displayed correctly and if the game restarts correctly.
* **test\_game\_over\_quit(self)**: Tests if the game over screen quits the game correctly when pressing ESC.
* **test\_game\_over\_close(self)**: Tests if closing the game over screen quits the game correctly.
* **test\_ball\_reset(self)**: Tests if the ball resets correctly after losing a life, ensuring the lives count decreases.
* **test\_scoring(self)**: Tests if the scoring is updated correctly when the ball hits a block.

#### Running the Tests:

* **if \_\_name\_\_ == '\_\_main\_\_': unittest.main()**: Command to run all tests in the class when the script is executed as a standalone program.

This breakdown ensures comprehensive coverage of all the features and functionalities tested within the Breakout game, adhering closely to the actual implementations and the unit tests that verify them.

**Figure 1 UnitTest Results:**



**Unit Test Code:**

import unittest

import pygame

from pygame.locals import KEYDOWN, QUIT, K\_LEFT, K\_RIGHT, K\_ESCAPE, K\_SPACE

from unittest.mock import patch, MagicMock

# where it says Team\_Beta\_Phase\_1, user needs to substitute with their file name for the game

from Team\_Beta\_Phase\_1 import Ball, Paddle, Block, collision\_checker, initialize\_game, main, game\_over

# where it says Team\_Beta\_Phase\_1, user needs to substitute with their file name for the game

# Define the color constants used in the game

WHITE = (255, 255, 255)

GREEN = (0, 255, 0)

RED = (255, 0, 0)

BLUE = (0, 0, 255)

YELLOW = (255, 255, 0)

class TestBreakoutGame(unittest.TestCase):

"""Unit test case for the Breakout game implementation."""

def setUp(self):

"""Initialize Pygame and create objects for testing."""

pygame.init()

self.screen = pygame.display.set\_mode((600, 500))

# Now including speed and color when initializing Paddle

self.paddle = Paddle(275, 490, 100, 20, 10, WHITE) # Speed and color added

self.ball = Ball(300, 250, 7, 5, WHITE)

self.blocks = [Block(i \* 100, 30, 80, 30, RED, 5) for i in range(5)]

initialize\_game()

@patch('pygame.event.get')

def test\_ball\_movement(self, mock\_get):

"""Test that the ball moves correctly."""

mock\_get.return\_value = [] # No user input

initial\_x = self.ball.posx

initial\_y = self.ball.posy

self.ball.update()

self.assertNotEqual((initial\_x, initial\_y), (self.ball.posx, self.ball.posy))

def test\_collision\_detection\_true(self):

"""Test collision between ball and paddle."""

rect1 = pygame.Rect(100, 100, 50, 50)

rect2 = pygame.Rect(120, 120, 50, 50) # Overlapping rect1

self.assertTrue(collision\_checker(rect1, rect2))

def test\_collision\_detection\_false(self):

"""Test collision detection between non-overlapping rectangles."""

rect1 = pygame.Rect(100, 100, 50, 50)

rect2 = pygame.Rect(200, 200, 50, 50) # Not overlapping rect1

self.assertFalse(collision\_checker(rect1, rect2))

def test\_paddle\_movement(self):

"""Test that the paddle moves correctly in response to left and right key presses."""

initial\_posx = self.paddle.posx

# Simulate left movement

self.paddle.set\_movement(-1)

self.paddle.update()

self.assertTrue(self.paddle.posx < initial\_posx, "Paddle should move left")

# Reset position and simulate right movement

self.paddle.posx = initial\_posx # Reset position

self.paddle.set\_movement(1)

self.paddle.update()

self.assertTrue(self.paddle.posx > initial\_posx, "Paddle should move right")

# Reset movement to neutral

self.paddle.set\_movement(0)

@patch('pygame.event.get')

def test\_game\_over\_display(self, mock\_get):

"""Test if game over screen is displayed correctly."""

# Mock run of the game

mock\_event = MagicMock()

mock\_event.type = KEYDOWN

mock\_event.key = K\_SPACE

mock\_get.return\_value = [mock\_event]

# Run game\_over and capture return value and assert True

# to restart game

result = game\_over()

self.assertTrue(result)

@patch('pygame.event.get')

def test\_game\_over\_quit(self, mock\_get):

"""Test if game over screen quits the game correctly."""

# Mock run of the game

mock\_event = MagicMock()

mock\_event.type = KEYDOWN

mock\_event.key = K\_ESCAPE

mock\_get.return\_value = [mock\_event]

# Run game\_over and capture return value and assert False

#to quit game

result = game\_over()

self.assertFalse(result)

@patch('pygame.event.get')

def test\_game\_over\_close(self, mock\_get):

"""Test if closing the game over screen quits the game correctly."""

# Mock run of the game

mock\_event = MagicMock()

mock\_event.type = QUIT # Change ()= pygame.QUIT) to ()= QUIT)

mock\_get.return\_value = [mock\_event] # was (pygame.event.get = MagicMock(return\_value=[mock\_event]))

# Run game\_over and capture return value and assert False

#to quit game

result = game\_over()

self.assertFalse(result)

# where it says Team\_Beta\_Phase\_1, user needs to substitute with their file name for the game

@patch('Team\_Beta\_Phase\_1.lives', 3)

def test\_ball\_reset(self):

"""Test if the ball resets correctly after losing a life."""

# where it says Team\_Beta\_Phase\_1, user needs to substitute with their file name for the game

from Team\_Beta\_Phase\_1 import lives

self.ball.reset()

self.assertEqual(lives, 3)

# where it says Team\_Beta\_Phase\_1, user needs to substitute with their file name for the game

@patch.dict('Team\_Beta\_Phase\_1.\_\_dict\_\_', {'score': 0, 'block\_list': []})

def test\_scoring(self):

"""Test if the scoring is updated correctly when the ball hits a block."""

# where it says Team\_Beta\_Phase\_1, user needs to substitute with their file name for the game

from Team\_Beta\_Phase\_1 import score, block\_list

block = Block(100, 100, 40, 15, RED, 10)

block\_list.append(block)

initial\_score = score

#simulate ball hitting the block

self.ball.posx, self.ball.posy = block.posx + 20, block.posy + 10

self.ball.update()

if collision\_checker(block.get\_rect(), self.ball.get\_rect()):

block.hit()

if block.get\_health() <= 0:

block\_list.remove(block)

score += block.get\_points()

self.assertEqual(score, initial\_score + block.get\_points())

def tearDown(self):

"""Quit Pygame."""

pygame.quit()

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()