INFORME TEST I QUALITAT DEL SOFTWARE

Maties Lopez Alarcon 1599296
Esther Ginfarré Blázquez 1494785

Introducció

El nostre projecte és el famós videojoc anomenat Tetris, en aquest cas es tracta d'una versió simplificada que es visualitza des de la terminal, no té música ni efectes d'animació en guanyar o perdre. De totes maneres tot el joc és completament funcional.

Enllaç directe al nostre Github per poder fer un pull del codi o revisar tots els comits que hem anat fent.

https://github.com/peepoclownn/Tetris-Game

Ha sigut programat en el llenguatge de programació "Python" i s'han utilitzat diverses llibreries i mòduls per facilitar tant el software com la part de testing que s'ha realitzat.

Els més importants a mencionar són "unittest" per crear d'una forma més còmoda els testos, concretament els mock objects per exemple.
El "ontab" ens ha ajudat al fet que si no hi ha cap interacció en uns segons per part del jugador, la peça continuï caient pel taulell fins a arribar al final i o xoqui amb un altre. També a què no faci falta prémer la tecla "Enter" cada vegada que es fa un moviment. Només farà falta al primer moviment, ja que no volem que el joc comenci just després de posar el nostre nom.

Com funciona per terminal tot el joc, la forma que hem pensat per actualitzar el taulell és amb un clear, el mòdul "os" ens facilita aquesta tasca amb la funció "os.system('clear')".

Instruccions d'execució

Després de fer un Git Pull del codi del videojoc hi ha diverses a coses a realitzar. L'obrirem amb el nostre IDE, es recomana el "pycharm".
Seleccionarem l'apartat de "Run/Debug configuration" on seleccionem l'arxiu que executarem per inicial el joc.
Afegirem l'arxiu /src/Vista/v_main.py, utilitzarem el Python3 (Nosaltres l'hem instal·lat amb l'Anaconda3) i a Environament variables afegirem si no està escrit "PYTHONNUNBUFFERED=1".
Seguidament, si mirem l'apartat "Modify options" ens sortirà una llista, en ella hem de marcar les opcions:

Open run/debug tool window when started
Add content roots to PYTHONPATH
Add source roots to PYTHONPATH
Emulate terminal in output console

Finalment, anirem a l'arxiu "/src/Controlador/c_main.py", si revisem la funció cls() a la línia 31, veiem que hi ha una funció comentada i l'altre no. Si el sistema operatiu en el qual s'executarà el codi és Windows, s'ha de deixar la línia 33, és a dir, os.system('cls').
En canvi, si el sistema operatiu és unix com Linux i Macosx hauràs de comentar la línia 33 i descomentar la línia 32, és a dir, os.system('clear').

Després de tots aquests preparatius, si executes el fitxer v_main.py com abans hem mencionat ja podràs jugar al joc. Posa la terminal la pantalla completa i veuràs el menú d'opcions on podràs jugar, veure els punts dels 3 millors jugadors i sortir del joc.

Implementació del codi per TDD

La implementació per codi "Test Driven Development" consisteix a desenvolupar software a partir de tests de funcions que ha de realitzar el codi.

S'ha implementat aquesta tècnica a bona part del codi, però a causa de problemes de teoria i explicació i implementació del test no s'ha pogut aplicar a tot el codi. Un exemple del TDD aplicat podria ser la funció "tablero_lleno()", el que ha de fer aquesta funció és saber si s'ha completat el taulell (hi ha una peça a l'última fila) i el test és introduir una peça l'última fila per veure si acaba el joc o no.

Test ubicat a ***test/model/m_tablero_test.py***:

```
def test_tablero_lleno(): #Se introduce un 1 en la fila 0 para ver si la función detecta correctamente que has perdido.
    tablero = Tablero(5, 10)
    for i in tablero.tablero:
        i[0] = 1
    assert(tablero.tablero_lleno(True) == True), "Fin de partida"
```

Veiem que el test crea un taulell 5x10 i introdueix una peça a la fila 0 que és la més alta, com la funció no acaba i no retorna True podem afirmar que falta una funció que aturi el joc, allà és quan creem la funció tablero_lleno (colocada)

Funció ubicada a ***src/model/m_tablero.py***:

```
def tablero_lleno(self, colocada):
    if colocada:
        for i in self.tablero[0:][0]:
            if i == 1:
                return True
    return False
```

Després, en inicialitzacions de les classes de Tauler i Puntuació, gràcies al TDD hem fet modificacions de les classes per passar el test.
Un dels exemples seria com el "programador" inicialitza el tauler i la puntuació màxima assolir. En comptes de llançar una excepció hem modificat les classes perquè s'inicialitzen amb el valor passat per paràmetre de forma absoluta.

Proves de caixa negra

En realitzar aquest tipus de proves ens hem trobat amb dos grans dificultats:

1. No saber on aplicar-ho: Com que el nostre codi està molt enfocat en què l'usuari només disposa d'accions mínimes, hem hagut de focalitzar molt les proves als errors del programador en utilitzar el nostre joc.
2. No disposar de funcions amb 3 paràmetres: Per realitzar el pairawrse testing no disposem de funcions en el codi que compleixin la condició.

A causa de les casuístiques indicades ha sigut dificultós aplicar aquestes proves, però on més s'han pogut aplicar ha sigut en el test de la classe taulell.

TESTING PARTICIONS EQUIVALENTS

A la classe Tauler, en fer la inicialització de l'objecte, s'ha dut a terme que sigui el que sigui el paràmetre que introdueixi el programador (negatiu o positiu) faci un absolut d'aquest i no salti una excepció. De la mateixa manera que, sí, és un número inferior a 5 inicialitzi amb 5 després d'informar per paràmetre.

La partició equivalent, per tant, seria un valor fora del 5 (2 i 3) i un a dins del permès (10 i 15). El test seria en conseqüència el següent, ubicat a ***test/model/m_tablero_test.py***:

```
# Test particio equivalent 1
tablero = Tablero( filas: 2,  columnas: 3)
assert (tablero.filas == 5 and tablero.columnas == 5), "Pone los valores por defecto"
# Test particio equivalent 2
tablero = Tablero( filas: 10,  columnas: 15)
assert (tablero.filas == 10 and tablero.columnas == 15), "Pone los valores indicados"
```

La funció a testejar de la classe taulell, ubicat a ***src/model/m_tablero.py***:

```
def __init__(self, filas, columnas):
    if abs(filas) >= 5 and abs(columnas) >= 5:
        self.filas = abs(filas)
        self.columnas = abs(columnas)
        self.tablero = [[0 for x in range(self.filas)] for y in range(self.columnas)]
    else:
        print("Valores erroneos, se inicializa con los minimos (5x5)")
        self.filas = 5
        self.columnas = 5
        self.tablero = [[0 for x in range(self.filas)] for y in range(self.columnas)]
```

TESTING VALORS FRONTERA I LÍMIT

A la mateixa funció de la classe taulell del test anterior s'ha realitzat els casos de test per als valors frontera i límit a la inicialització de les files i les columnes.

Els casos de prova respecta aquestes situacions:

```
#Test frontera
tablero = Tablero( filas: 5,  columnas: 5)
assert (tablero.filas == 5 and tablero.columnas == 5), "Se incializa con los valores dados."

# Test limit 1
tablero = Tablero( filas: 6,  columnas: 6)
assert (tablero.filas == 6 and tablero.columnas == 6), "Se incializa con los valores dados."
# Test limit 2
tablero = Tablero( filas: 4,  columnas: 4)
assert (tablero.filas == 5 and tablero.columnas == 5), "Se incializa con los valores dados."
```

Com es pot observar, només hi ha una frontera en aquest cas que seria el 5, ja que no s'ha posat cap restricció respecte a la mida del taulell.

TESTING PAIRWISE TESTING

Com a Pairwise Testing hem hagut de crear una nova funció no implementada. Aplicant TDD hem creat primer els nostres casos de prova per després implementar la funció. S'ha imaginat com un possible tipus per fer de forma aleatòria segons cada cop que es faci una línia la suma dels punts.

En aquest cas en la nostra nova funció s'ubica a **src/models/m_puntuación.py**:

```
    '''
    Función creada para realizar test de pairwise.
    Seria utilizada para variar los puntos a sumar cuando se haga linea.
    '''
    def puntuacion_pairwise_testing(self, r1, r2, r3, p):
        if (r1 == 1 or r1 == 2) and (r2 == 1 or r2 == 2) and (r3 == 1 or r3 == 2):
            self.puntos += (p+r1*r2)-r3
        else:
            self.puntos += 20
        return self.puntos
```

Com es pot veure, s'ha decidit que els paràmetres r1, r2, r3 només accepti 1 o 2. Si és diferent d'aquests valors només se sumarà els punts corresponents (p) que decideixi el programador.

El total de combinacions serien: 2 x 2 x 2 = 2^3.

Gràcies, el pairwise testing aquest nombre se'n redueix a 4 casos de prova:

Caso de Prueba	r1	r2	r3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

Per fer el test, ho hem realitzat en aquesta funció que es troba ubicada a ***test/models/m_puntuacion_test.py***. Per no ocupar massa espai aquesta sería la implementació del primer cas de pairwise testing:

```
def test_puntuacion_pairwise_testing():
    random1 = 1
    random2 = 1
    random3 = 1
    puntos = Punts(200)
    p = 20
    puntos.puntuacion_pairwise_testing(random1, random2, random3, p)
    assert (puntos.get_puntos() == 20)
```

Proves de caixa blanca

TESTING STATMENT COVERAGE

No disposem d'un test que "simuli" una partida amb totes les casuístiques possibles per fer el statment coverage. Disposem de parts de codi on complim que s'ha executat tot de la funció.

Podem dir que no hem fet un statment coverage del 100% al test a causa de no fer la simulació de joc, ja que hi ha funcions de moviment (dreta, esquerra, a baix) que no les hem testejat, pel fet que depenen d'una altra funció per portar a cap el moviment que és overlap_check.

Així i tot, el statmenent coverage encara que no disposem d'una eina que ens indiqui el percentatge total podem assumir que estarà entre un 75% i 80%.

TESTING DECISION COVERAGE

Per fer decision coverage, s'ha dut a terme que els diferents tests comprovessin passar per la condició i no passar-hi.
Un cas on complim seria el cas de prova de les fitxes. En el test ubicat a test/Model/m_fichas_test.py hi trobem dues funcions:

```python
# Si le damos una pieza erronea la función la reconoce i no la rotará.
1494785
def test_rotar_pieza_fail():
    pieza = [[1, 0],
             [1, 0],
             [1, 1]]
    assert(fichas.rotar_pieza(pieza) != False), "Pieza erronea detectada"

1494785 +1
def test_rotar_pieza_pass():
    pieza = [[1, 1],
             [1, 0],
             [1, 1]]
    assert(fichas.rotar_pieza(pieza) != False), "Pieza erronea detectada"
```

En aquestes funcions estem comprovant la condició de la funció de **src/Model/m_fichas.py**:

```python
# Función que rota la pieza correspondiente
4 usages  Maties Lopez
def rotar_pieza(pieza):
    if(pieza in piezas_completas):
        copiaPieza = deepcopy(pieza)
        reversa = copiaPieza[::-1]
        piezafinal = []
        for element in zip(*reversa):
            piezafinal.append(list(element))
        return piezafinal
    else:
        return False
```

Com es pot veure amb el test estem mirant si la peça donada, és una de les tan possibles peces que pot haver-hi de girar o no, i el cas en què és una peça no possible. En dur a terme aquest test ens assegurem que aquesta funció passa pels dos estats possibles: True o False en la condició. Per tant, hem fet decision coverage i en aquest cas en particular, també hem fet statment coverage.

TESTING CONDITION COVERAGE

La tècnica Condition coverage s'ha aplicat amb diversos tests que asseguren que les subcondicions de cada "if-else" hagin passat per tots els estats possibles (True o False).

Un exemple pot ser al test anomenat test_inicializacion() que es troba al fitxer **/test/Model/m_tablero_test.py.**
Ja que amb diversos casos es prova el resultat de la funció quan les files i les columnes són majors o iguals a 5, que vol dir que totes dues són True. En l'altre cas, s'introdueixen inputs menors que 5, per tant, en aquest cas les dues subcondicions resultaran False i saltaran al "else" directament.

TESTING PATH COVERAGE

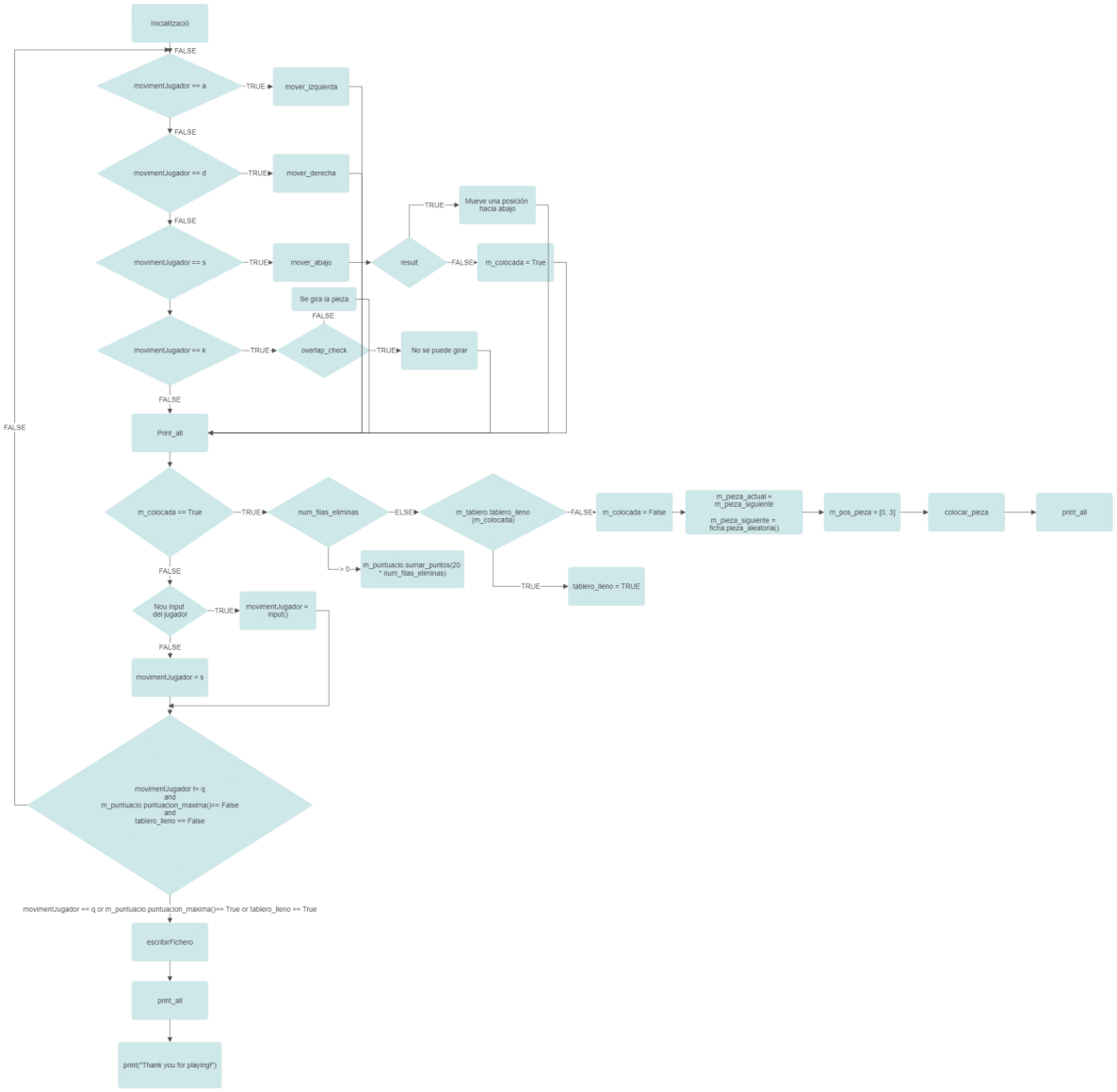
La funció principal del nostre codi s'anomena playGame, és on els inputs del jugador es veuen reflectits, si el jugador utilitza les tecles "a","s", "d" o "k" veurem com cada un d'ells té un camí a recórrer. També si prem "q", has arribat als punts màxims o has completat el taulell hi haurà una altra direcció, en aquest cas serà finalitzar la partida. En canvi, si el jugador no introdueix cap input a part del primer per iniciar el joc, el sistema introduirà ell mateix la tecla "s" per a continuar baixant fins que arribi al final del taulell.

Un dels camins que podem obtenir pot ser en la inicialització l'usuari introdueix com a input "k". En aquest cas va passant per totes les condicions fins a arribar a moviment Jugador == k que és True, en aquell moment es revisa amb la funció overlap_check si es pot girar la peça o no. En cas que es pugui es gira (com és el cas) i cridem el print_all, si no es deixa la peça com a l'inici i cridem el print_all de totes formes. En aquest cas com és el primer moviment no ha arribat a tocar cap altre peça ni al final del taulell així que m_colocada = False. De nou, s'espera el següent input del jugador, en aquest cas no introduïm cap així que el mateix sistema després d'uns milisegons s'assigna l'input "s" per continuar baixant. Arribem a la condició del bucle general on com l'input no és "q", no s'ha arribat a la puntuació màxima, ja que no hem fet cap línia i el taulell no està ple perquè és l'única peça tornem a iniciar tota la funció on el següent input serà l'anteriorment mencionat "s".
El test que es realitza pel Path Coverage està a l'arxiu **/test/Controlador/c_main_test.py**

Aquí el podem veure:

```
    '''Test del Path Coverage que comprueba que si apretamos la tecla "k", la pieza gire correctamente'''
    @patch( target: 'builtins.input', side_effect=['k'])
    @patch('builtins.print')
    def test_path_coverage(self, mock_print, mock_input):
        controlador.playGame("tst")
        mock_print.assert_any_call('Pieza girada correctamente')
```

Aquest és el diagrama de flux complet de tots els camins possibles a la funció playGame():



TESTING LOOPING TESTING

En el nostre codi només disposem de 3 bucles que es recorrin diversos cops. Altres “bucles” que disposem, en arribar a una condició en concret fa un retorn del necessari.
Degut això només podíem fer loop testing de limitades situacions: col·locar i esborrar peça.
En la funció original una és la còpia de l’altre, simplement que modifica la posició indicada.
Si s’esborra s'omple de 0, si es col·loca d’1.
Les funcions testejades són les següents:

```python
def colocar_pieza(self, pieza, posicion):
    curr_piece_x = len(pieza)
    curr_piece_y = len(pieza[0])
    for i in range(curr_piece_x):
        for j in range(curr_piece_y):
            if self.tablero[posicion[0]+i][posicion[1]+j] == 0:
                if pieza[i][j] == 1:
                    self.tablero[posicion[0]+i][posicion[1]+j] = pieza[i][j]

7 usages  Maties Lopez +1
def borrar_pieza(self, pieza, posicion):
    curr_piece_x = len(pieza)
    curr_piece_y = len(pieza[0])
    for i in range(curr_piece_x):
        for j in range(curr_piece_y):
            if self.tablero[posicion[0] + i][posicion[1] + j] == 1:
                # CHEQUEAR QUE NO SE VAYA DEL TABLERO.
                if pieza[i][j] == 1:
                    self.tablero[posicion[0] + i][posicion[1] + j] = 0
```

Les funcions test de loop testing es troben a la ruta ***test/Model/m_tablero_test.py*** i són les següents:

1— Test en el cas no realitzem cap modificació, ja que el valor de la peça és el mateix que el del tauler:

```python
def test_looptesting_1():
    tablero = Tablero( filas: 5, columnas: 5)
    respuesta = [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    pieza = [[0]]
    posicion = [0, 0]
    tablero.borrar_pieza(pieza,posicion)
    assert(tablero.tablero == respuesta)
```

2— Test en el cas no realitzem la modificació, pel fet que el valor de la peça és diferent que el del tauler. En aquest cas utilitzem també la funció colocar_pieza()

```python
def test_looptesting_2():
    # Caso en el que el bucle se ejecuta una vez
    tablero = Tablero( filas: 5, columnas: 5)
    respuesta = [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    respuesta2 = [[0, 1, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    pieza = [[1]]
    posicion = [0, 1]
    tablero.colocar_pieza(pieza, posicion)
    assert (tablero.tablero == respuesta2)
    tablero.borrar_pieza(pieza, posicion)
    assert(tablero.tablero == respuesta)
```

3— Test en què es recorre tota la peça i es col·loca en la posició indicada al tauler i s'elimina correctament:

```python
def test_looptesting_3():
    # Caso en el que el bucle se ejecuta una vez
    tablero = Tablero( filas: 5, columnas: 5)
    respuesta = [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    respuesta2 = [[1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    pieza = [[1, 1, 1, 1, 1], [1, 1, 1, 1, 1]]
    posicion = [0, 0]
    tablero.colocar_pieza(pieza, posicion)
    assert (tablero.tablero == respuesta2)
    tablero.borrar_pieza(pieza, posicion)
    assert(tablero.tablero == respuesta)
```

Mock objects

Sigui per arribar a situacions desitjades o controlar casos impossibles de controlar, hem fet ús dels mock objects. S'ha utilitzat la llibreria d'unittest.mock per fer les proves.

Un dels usos ha sigut a la nostra funció principal "playGame()" que és el nucli del joc. Hem testejat amb casos senzills o condicions invàlides que els outputs siguin correctes. La funció a testejar està a la ubicació ***src/controlador/c_main.py***.

Per fer un ús correcte de la llibreria, i en aquest cas poder indicar els inputs que es volen fer servir, hem de crear una classe de test. Aquesta classe es troba a la ubicació ***test/controlador/c_main_test.py***:

```python
class TestPlayGame():
    new *
    @patch( target: 'builtins.input', side_effect=['s', 'q'])
    @patch('builtins.print')
    def test_basic_flow(self, mock_print, mock_input):
        controlador.playGame("tst")
        mock_print.assert_any_call('----------------------------------TETRIS GAME----------------------------------')

    new *
    @patch( target: 'builtins.input', side_effect=['x', 'q'])
    @patch('builtins.print')
    def test_invalid_input(self, mock_print, mock_input):
        controlador.playGame("tst")
        mock_print.assert_any_call('Invalid input')

    new *
    @patch( target: 'builtins.input', side_effect=['s', 'k', 'x', 's', 's', 's', 's', 's', 's', 's', 's', 's'])
    @patch('builtins.print')
    def test_gameover(self, mock_print, mock_input):
        controlador.playGame("tst")
        mock_print.assert_any_call('Thank you for playing!')
```

En el test bàsic, fem simplement una tecla, en prémer aquesta tecla ha de sortir un dels prints que surten a la consola per continuar.

En el test invàlid, cliquem com a primera instància una tecla diferent de les permeses de moviment. En fer això apareix un print d'Invàlid input, esperant una tecla permesa.

Finalment, fer un cas per forçar el game_over, en aquest cas només fem clic la tecla "s" per omplir de forma vertical el tauler i arribi al màxim d'aquest.

Conclusions

En resum, en aquest projecte hem pogut veure tots els conceptes que engloba el sector del Testing. El nostre software és el Tetris, sembla un joc senzill de programar i testejar, però en realitat requereix un gran esforç per part dels enginyers que han de saber aplicar correctament les tècniques que han après. Començant per la implementació de Test Driven Development on mitjançant tests hem pogut desenvolupar el codi i provar que no hi hagi errors en ell.
En l'entorn de proves de caixa negra on la forma de fer testing és tenint en compte els inputs i outputs de les funcions, amb les particions equivalents hem pogut veure el cas de què el valor de la fila o la columna sigui menor que 5, en aquest cas el mateix codi s'assignava 5 com a mínim. Els valors frontera i límit s'han testejat amb les dimensions del taulell, en aquest cas el valor és 5 així que el valor frontera és 6 i el valor límit és 4.
Per al pairwise es va fer una nova funció que rebia tres valors que podien ser 1 o 2, quan el jugador completava una fila els 20 punts que obté s'hauran d'aplicar a una fórmula juntament amb els tres valors aleatoris. La puntuació podrà quedar igual, menor que 20 o major que 20 depenent de l'aleatorietat del moment.

Dins de les proves de caixa blanca hem revisat el Statement coverage que significa que totes les línies del codi s'hagin executat almenys una vegada. També el condition i decision 7coverage que fan referència al fet que s'han testejat l'if-else de totes les condicions i tots els casos de les subcondicions de cada if respectivament.

El Path coverage en aquest cas s'ha fet de la funció principal anomenada playGame(), on s'ha creat un diagrama de tots els possibles camins, hem realitzat i documentat una jugada per veure quin camí seguiría en un cas concret.

En el Looping testing on fem proves de recorregut de bucles per verificar que és recorrent correctament, com per exemple a la funció colocar_pieza() o borrar_pieza().

Finalment, amb els Mock objects, hem pogut veure la seva importància dintre del testing per situacions concretes de la partida. Un esdeveniment on una peça arriba al final del taulell i es col·loca, o entra en contacte amb un altre peça i es col·loca automàticament.
També poden ser situacions on no podem controlar certes situacions per fer test com els random, o per "jugar" de forma automàtica introduint els inputs com mock objects.

Amb aquesta pràctica ens hem adonat de la importància del testing en un software. Gràcies a aplicar-ho hem pogut corregir errors i anar millorant el nostre software per aconseguir que passaren el test. Hem passat el software en situacions no esperades, millorant així la reacció del nostre.