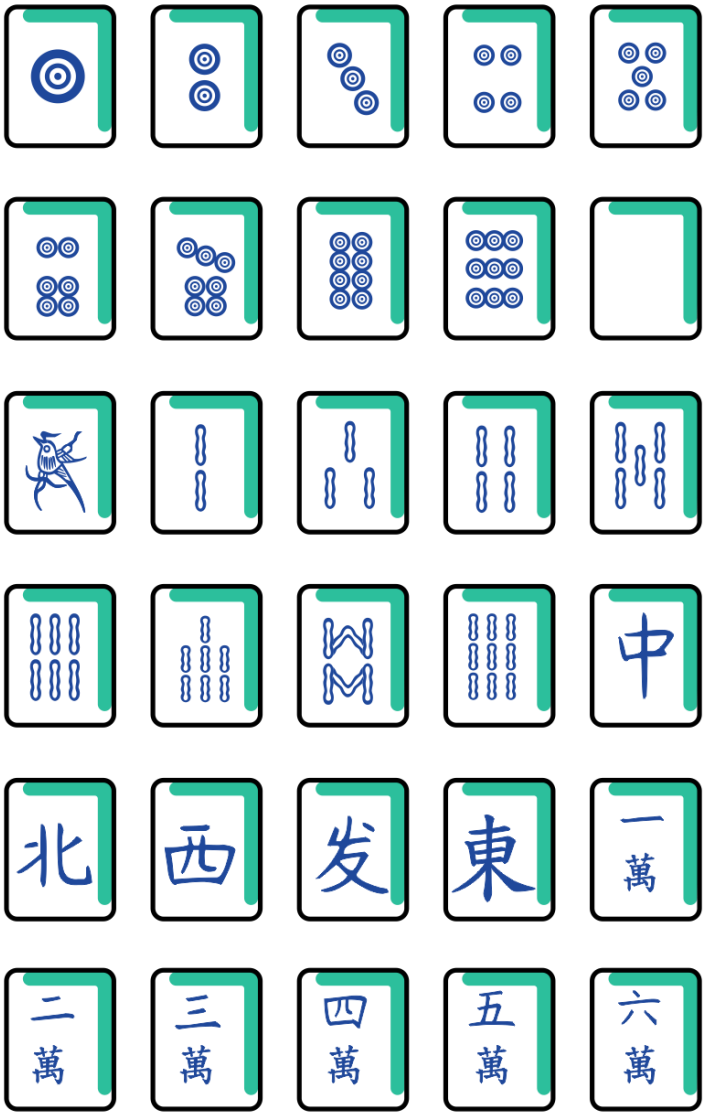


TILEDOM

PRÀCTICA 1: APLICACIÓ DES DE ZERO

LUCÍA TORRESCUSA RUBIO - 1633302

TEST I QUALITAT DEL SOFTWARE

2025-26

Índex

1. Introducció 2
2. Normes establertes 2
3. Arquitectura MVC 2
3.1 Model 2
3.2 Vista 3
3.3 Controlador 3
4. Funcionalitats 4
4.1 Inicialització del taulell segons dificultat 4
4.2 Verificació de que una parella és correcta 6
4.3 Verificació de l'estat de la partida 7
4.4 Generació de peces de manera equilibrada 9
4.5 GameController 11
4.6 Diagrames de path coverage 15
5. Data-driven testing 17
6. Integració continua i desplegament continu 18

Informe de Pràctiques – Tiledom

1. Introducció

El projecte Tiledom per l'assignatura Test i Qualitat del Software implementa un joc basat en el joc de Tiledom amb l'estètica del Mahjong, però quintant-li la dificultat de les diverses capes.

A nivell de software, s'ha implementat seguint l'arquitectura Model-Vista-Controlador, i s'han aplicat diverses tècniques avançades de testing, incloent *test-driven development*, caixa negra, caixa blanca, automatització amb la tècnica de *data-driven*, *mock objects* i CI/CD.

2. Normes establertes

Com que per reduir la dificultat no s'ha fet una còpia exacta del joc Tiledom, s'han definit les següents regles pel seu funcionament:

1. Hi ha tres nivells de dificultat: fàcil, mitjà i difícil.
2. Cada dificultat té una mida de taulell, un número de peces i una quantitat de tipus predeterminada:
    a. El nivell senzill és un taulell 8x8, amb 40 peces de 5 tipus diferents.
    b. El nivell mitjà és un taulell de 10x10, amb 60 peces de 7 tipus diferents.
    c. El nivell difícil és un taulell de 12x12, amb 90 peces de 10 tipus diferents.
3. Hi ha d'haver un nombre de peces parell, i cada tipus de peça també ha d'aparèixer en parelles.
4. Totes les peces han de quedar pegades al centre del taulell.
5. Per poder eliminar una parella, cada peça ha de ser del mateix tipus i ha de tenir un costat (dreta o esquerre) lliure.
6. Serà victòria si el taulell queda buit.
7. Serà derrota si el taulell no està buit i no queden parelles eliminables.

3. Arquitectura MVC

El projecte està dividit en les tres capes de Model-Vista-Controlador:

3.1 Model

El Model està format per:

- Board
- GameSession
- RandomTileGenerator

Aquesta capa del codi s'encarrega de representar el taulell i gestionar l'estat del joc. Inclou tota la lògica de manipular fitxes, els càlculs, les validacions i l'actualització dels estats.

3.2 Vista

La vista inclou:

- GameWindow
- BoardPanel
- UIManager
- AudioManager i ImplementedAudioManager

Aquesta capa s'encarrega de la gestió de finestres i renderització, la interacció visual amb l'usuari com els clics, i la reproducció d'àudio i la resposta visual.

3.3 Controlador

Té GameController i els següents estats:

- InitState
- PlayState
- PauseState
- ConfigureState

El controlador rep els esdeveniments de la Vista i actualitza el Model, de manera que decideix el flux del joc i l'estat actiu.

Per cada capa de codi desenvolupat, s'ha creat una capa de test, que tractarem als següents punts.

4. Funcionalitats

Totes les funcions s'han implementat seguint el mètode TDD: un primer test fallit, una primera implementació, i les correccions conseqüents. De totes formes, i per arribar a assolir tots els tests demanats, s'han continuat afegint tests un cop desenvolupades les funcions.

4.1 Inicialització del taulell segons dificultat

Creació d'un taulell de joc amb mida variable (8, 10 o 13), nombre de tipus de peces i nombre de peces inicials segons la dificultat seleccionada. Inclou validació d'arguments i inicialització del generador de peces amb RandomTileGenerator.

```java
public class Board {

    // extraiem el generador per poder aplicar el mock als tests
    private RandomTileGenerator genRandom;

    //mida segons nivell de dificultat -> generarà un taulell tiles[size][size]
    private int size;
    //valor 0 -> no hi ha peça
    //valors 1-10 -> diferents tipus de peces
    private int tiles[][];

    // -------------------- Funcions inicialització taulell ----------------------------
    public Board() {
        throw new IllegalArgumentException(s: "Falten variables per inicialitzar el taulell");
    }
    // funció que inicialitzi el taulell d'una mida variable
    // segons la dificultat i les peces col·locades aleatòriament
    public Board(int dificultat, RandomTileGenerator gen) {
        if (dificultat < 1 || dificultat > 3)
            throw new IllegalArgumentException(s: "Dificultat fora de rang (1-3)");

        this.genRandom = gen;

        int numTipus = 0, numPeces = 0;
        // mida i peces segons dificultat
        switch (dificultat) {
            case 1 -> {size = 8; numTipus = 5; numPeces = 40;}
            case 2 -> {size = 10; numTipus = 7; numPeces = 60;}
            case 3 -> {size = 12; numTipus = 10; numPeces = 90;}
        }
        tiles = new int[size][size];
        setTiles(size, numTipus, numPeces);
    }
```

Aquesta funció es troba dins l'arxiu board.java. Per poder testejar-la abans d'haver implementat RandomTileGenerator, es va haver de fer servir un mockObject d'aquesta classe.

Se li han realitzat els següents tests de caixa negra:

- testBoardSizeIsCorrect() amb particions equivalents de dificultats vàlides, valors límits i fora de rang.

- testBoardConnected()
- testBoardHasEvenTiles()
- testBoardConstructor_UsesRandomGenerator() per comprovar que, quan es feia servir el mock, realment l'estava cridant.

Després dels tests de caixa negra, aquesta funció tenia el següent coverage:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
hasAvailableMoves()		100 %		90 %	2	12	0	11	0	1
setTiles(int, int, int)		100 %		85 %	3	11	0	12	0	1
tryMatch(int, int, int, int)		100 %		78 %	3	8	0	9	0	1
Board(int, RandomTileGenerator)		100 %		75 %	2	6	0	11	0	1
isSideFree(int, int)		100 %		100 %	0	7	0	3	0	1

Totes les instruccions de la funció estaven cobertes, però hi havia algunes branques per les que no passava mai.

```
    // --------------------- Funcions inicialització taulell ----------------------------
    public Board() { throw new IllegalArgumentException("Falten variables per inicialitzar el taulell"); }
    // funció que inicialitzi el taulell d'una mida variable segons la dificultat i les peces col·locades aleatòriament
    public Board(int dificultat, RandomTileGenerator gen) {
        if (dificultat < 1 || dificultat > 3) throw new IllegalArgumentException("Dificultat fora de rang (1-3)");

        this.genRandom = gen;

        int numTipus = 0, numPeces = 0;
        // mida i peces segons dificultat
        switch (dificultat) {
            case 1 -> {size = 8; numTipus = 5; numPeces = 40;}
            case 2 -> {size = 10; numTipus = 7; numPeces = 60;}
            case 3 -> {size = 12; numTipus = 10; numPeces = 90;}
        }
        tiles = new int[size][size];
        setTiles(size, numTipus, numPeces);


    }
```

Per solucionar això, es va crear el següent test de caixa blanca:

- testBoardConstructor_InvalidDifficulty_WhiteBox()

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
hasAvailableMoves()		100 %		100 %	0	12	0	11	0	1
setTiles(int, int, int)		100 %		85 %	3	11	0	12	0	1
tryMatch(int, int, int, int)		100 %		100 %	0	8	0	9	0	1
Board(int, RandomTileGenerator)		100 %		87 %	1	6	0	12	0	1
isSideFree(int, int)		100 %		100 %	0	7	0	3	0	1
isEmpty()		100 %		100 %	0	4	0	4	0	1

D'aquesta manera, es va arribar al màxim coverage de branques per aquesta funció, ja que al switch mai arribaria un valor que no fossin les dificultats acceptades, i mai es compliria el cas default. Així, entre tots els tests que s'han fet sobre aquesta funció, en té statement coverage, decision coverage sobre el switch i condition coverage amb la dificultat dins i fora dels límits.

A més, aquesta funció crida a setTiles(), funció que també es testeja

```
public void setTiles(int size, int numTipus, int numPeces){
    // costat mínim que pot contenir totes les peces
    int side = (int) Math.ceil(Math.sqrt(numPeces));

    // Calculem l'offset per centrar el bloc
    int start = (size - side) / 2;
    int end = start + side;

    // Omplim les peces dins del bloc central
    int placed = 0;
    for (int i = 0; i < size && placed < numPeces; i++) {
        for (int j = 0; j < size && placed < numPeces; j++) {
            // només col·loca peces si la posició està dins del bloc central
            // i una petita probabilitat extra si és al voltant
            boolean insideCenter = i >= start && i < end && j >= start && j < end;
            if (insideCenter || (Math.random() % 5 == 0)) {
                int value = genRandom.genera();
                tiles[i][j] = value;
                placed++;
            }

        }
    }
}
```

amb els mateixos tests i que inclou un doble bucle for aniuat. A aquesta funció se lo fa un loop testing amb les funcions testBoardConnected() i isConnected(), a més d'un path coverage sobre el flux. De totes maneres, aquesta funció tampoc pot aconseguir un branch coverage del 100% degut a que fa servir Math.random().

La funció setTiles() fa una col·locació inicial de peces al taulell dins d'un bloc central calculat a partir del nombre de peces. Es fa servir el generador genRandom.genera() per assignar el tipus de peça. En acabar d'executar el constructor, tenim el taulell preparat per començar a jugar.

4.2 Verificació de que una parella és correcta

A la funció tryMatch() es fa la comprovació de que dues peces es poden eliminar del taulell. Aquesta valida que les peces no siguin buides (representades com a 0 en el taulell), que siguin del mateix tipus, que no siguin la mateixa casella i que totes dues tinguin un costat lliure. Finalment, en cas afirmatiu, s'eliminen les peces.

```java
// Comprova si les peces seleccionades tenen al menys un lateral lliure
// que no siguin posicions lliures
// i si les dues són del mateix tipus
public boolean tryMatch(int x1, int y1, int x2, int y2) {
    // son buides
    if (tiles[x1][y1] == 0 || tiles[x2][y2] == 0) return false;
    // son de diferent tipus
    if (tiles[x1][y1] != tiles[x2][y2]) return false;
    // és la mateixa peça
    if (x1 == x2 && y1 == y2) return false;

    // tenen al menys un costat buit
    boolean firstFree = isSideFree(x1, y1);
    boolean secondFree = isSideFree(x2, y2);
    if (!(firstFree && secondFree)) return false;

    tiles[x1][y1] = 0;
    tiles[x2][y2] = 0;
    return true;
}

// Funció auxiliar per comprovar si tenen un costat buit
private boolean isSideFree(int i, int j) {
    // si és fora del taulell també és lliure
    boolean leftFree  = (j - 1 < 0) || tiles[i][j - 1] == 0;
    boolean rightFree = (j + 1 >= tiles[i].length) || tiles[i][j + 1] == 0;
    return leftFree || rightFree;
}
```

Aquesta funció també es troba dins de board.java. Compta amb els següents tests de caixa negra:

- testTryMatch() fa les comprovacions de diferents tipus de parelles que es poden seleccionar: sense cap costat lliure, al límit del taulell, amb peces buides als costats, dues peces lliures, una peça lliure i una plena, etc.
- testTryMatchFailsWhenSameTileSelected()
- testTryMatchPairwise() disseny de test pairwise fent servir el tipus de la peça, i quin costat tenen lliure.

Després d'aquests tests, es va rebre aquest resultat de coverage:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
hasAvailableMoves()		100 %		90 %	2	12	0	11	0	1
setTiles(int, int, int)		100 %		85 %	3	11	0	12	0	1
tryMatch(int, int, int, int)		100 %		78 %	3	8	0	9	0	1
Board(int, RandomTileGenerator)		100 %		75 %	2	6	0	11	0	1
isSideFree(int, int)		100 %		100 %	0	7	0	3	0	1
isEmpty()		100 %		100 %	0	4	0	4	0	1

```
public boolean tryMatch(int x1, int y1, int x2, int y2) {
    // son buides
    if (tiles[x1][y1] == 0 || tiles[x2][y2] == 0) return false;
    // son de diferent tipus
    if (tiles[x1][y1] != tiles[x2][y2]) return false;
    // és la mateixa peça
    if (x1 == x2 && y1 == y2) return false;

    // tenen al menys un costat buit
    boolean firstFree = isSideFree(x1, y1);
    boolean secondFree = isSideFree(x2, y2);
    if (!(firstFree && secondFree)) return false;

    tiles[x1][y1] = 0;
    tiles[x2][y2] = 0;
    return true;
}
```

Faltaven algunes branches per recórrer. Es van crear els següents tests de caixa blanca per solucionar-lo:

- testTryMatch_FirstIf()
- testTryMatch_ThirdIf()
- testTryMatch_FourthIf()

De manera que es tractaven totes les possibilitats de les variables, aconseguint el 100% de coverage:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
hasAvailableMoves()		100 %		100 %	0	12	0	11	0	1
setTiles(int, int, int)		100 %		85 %	3	11	0	12	0	1
tryMatch(int, int, int, int)		100 %		100 %	0	8	0	9	0	1
Board(int, RandomTileGenerator)		100 %		87 %	1	6	0	12	0	1

Així doncs, amb aquesta funció cobrim el statement coverage, decision coverage i condition coverage.

4.3 Verificació de l'estat de la partida

Després de cada jugada feta amb tryMatch(), es fa una crida a isEmpty() i a hasAvailableMoves() per comprovar si el taulell s'ha quedat buit (doble for verificant si hi ha alguna peça que no sigui 0), o si queda alguna parella buida amb costat(s) lliures que es pugui eliminar.

```
// Comprova si queden peces al taulell
public boolean isEmpty() {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (tiles[i][j] != 0) return false;
        }
    }
    return true;
}
```

```java
// Comprova si queden moviments disponibles al taulell
public boolean hasAvailableMoves() {

    //recorrem tot el taulell
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            int current = tiles[i][j];
            //ignorem les buides
            if (current == 0) continue;

            //seleccionem les peces amb costats lliures
            if (isSideFree(i, j)) {
                //busquem si hi ha una del mateix tipus amb costat lliure
                for (int x = 0; x < size; x++) {
                    for (int y = 0; y < size; y++) {
                        if ((x == i && y == j) || tiles[x][y] == 0) continue;
                        if (tiles[x][y] == current && isSideFree(x, y)) {
                            return true; //hi ha al menys una jugada disponible
                        }
                    }
                }
            }
        }
    }
    return false;
}
```

Els tests de caixa negra fets a aquestes funcions són:

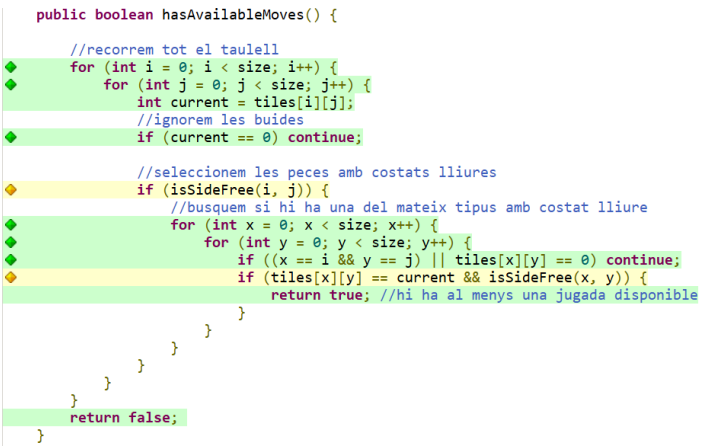
- testIsEmpty()
- testHasAvailableMoves()

Amb aquests tests es va aconseguir el següent coverage:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
hasAvailableMoves()		100 %		90 %	2	12	0	11	0	1
setTiles(int, int, int)		100 %		85 %	3	11	0	12	0	1
tryMatch(int, int, int, int)		100 %		78 %	3	8	0	9	0	1
Board(int, RandomTileGenerator)		100 %		75 %	2	6	0	11	0	1
isSideFree(int, int)		100 %		100 %	0	7	0	3	0	1
isEmpty()		100 %		100 %	0	4	0	4	0	1
setTiles(int[][])		100 %		n/a	0	1	0	3	0	1

Per isEmpty() ja estava tot cobert, però vaig afegir els següents tests de caixa blanca per hasAvailableMoves():

- testHasAvailableMoves_isSideFree()
- testHasAvailableMoves_InnerIf()



Tractant aquestes dues instruccions que no passaven per totes les branques, es va assolir un 100% de coverage a la funció:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
hasAvailableMoves()		100 %		100 %	0	12	0	11	0	1
setTiles(int, int, int)		100 %		85 %	3	11	0	12	0	1
tryMatch(int, int, int, int)		100 %		100 %	0	8	0	9	0	1
Board(int, RandomTileGenerator)		100 %		87 %	1	6	0	12	0	1

En aquestes funcions fem statement coverage, decision coverage, condition coverage i loop testing, en aquest cas de quatre fors aniuats (i, j & x, z).

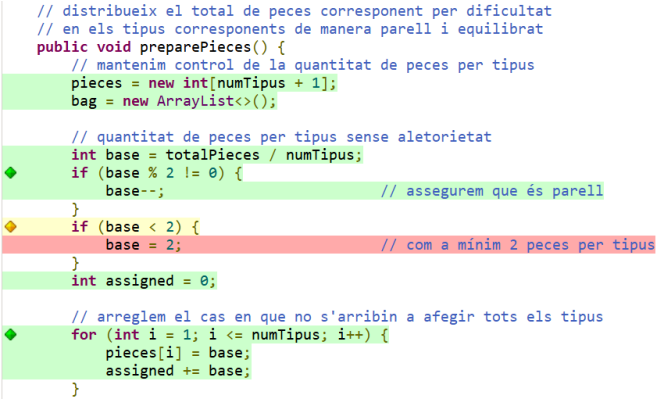
4.4 Generació de peces de manera equilibrada

A randomTileGenerator.java gestionem el nombre total de peces depenent de la dificultat, la distribució de parelles de manera aleatòria però equilibrada, l'emplenament de la bossa de peces (on es guarden les peces que queden per afegir al taulell) i l'extracció de peces aleatòries amb genera(). La funció clau és preparePieces():

```java
// distribueix el total de peces corresponent per dificultat
// en els tipus corresponents de manera parell i equilibrat
public void preparePieces() {
    // mantenim control de la quantitat de peces per tipus
    pieces = new int[numTipus + 1];
    bag = new ArrayList<>();

    // quantitat de peces per tipus sense aletorietat
    int base = totalPieces / numTipus;
    if (base % 2 != 0) {
        base--;                         // assegurem que és parell
    }
    if (base < 2) {
        base = 2;                       // com a mínim 2 peces per tipus
    }
    int assigned = 0;

    // arreglem el cas en que no s'arribin a afegir tots els tipus
    for (int i = 1; i <= numTipus; i++) {
        pieces[i] = base;
        assigned += base;
    }
    // ajustem si han quedat peces sense assignar
    while (assigned < totalPieces) {
        int i = rand.nextInt(numTipus) + 1;
        pieces[i] += 2;
        assigned += 2;
    }

    // omplim la bossa de peces
    for (int i = 1; i <= numTipus; i++) {
        for (int j = 0; j < pieces[i]; j++) {
            bag.add(i);
        }
    }

    // barregem la bossa
    Collections.shuffle(bag, rand);

}
```

Aquesta funció té el test de caixa negra de testPairDistribution(), que comprova el flux d'instruccions per cadascun dels nivells possibles. Després dels tests de caixa negra, la funció té aquest coverage:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
preparePieces()		97 %		91 %	1	7	1	20	0	1
genera()		88 %		50 %	1	2	0	2	0	1



Veiem que, per la distribució normal del nombre de peces per taulell, hi ha una instrucció a la que mai arriba a accedir.

Per solucionar això, fem testPreparePieces_BaseLessThanTwo(), on forcem la base a que sigui menor de 2. D'aquesta manera, aconseguim el 100% de coverage:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
preparePieces()		100 %		100 %	0	7	0	21	0	1
RandomTileGenerator(int)		100 %		100 %	0	4	0	9	0	1

Aquesta funció es crida des del constructor:

```java
public class RandomTileGenerator {

    private int numTipus;
    private int totalPieces;
    // cada posició de l'array és un tipus de peça i conté la seva quantitat
    private int[] pieces;
    // lista de les peces barrejades aleatòriament
    private List<Integer> bag;
    private Random rand = new Random();

    public RandomTileGenerator(){}

    // amb la dificultat podem conèixer la quantitat de peces i tipus
    public RandomTileGenerator(int dificultat) {
        switch (dificultat) {
            case 1 -> { numTipus = 5; totalPieces = 40; }   // fàcil
            case 2 -> { numTipus = 7; totalPieces = 60; }   // intermig
            case 3 -> { numTipus = 10; totalPieces = 90; }  // difícil
            default -> throw new IllegalArgumentException(s: "Dificultat fora de rang (1-3)");
        }

        preparePieces();
    }
```

La qual té el test de caixa negra testConstructor() fet amb particions i límits. Amb aquest test ja assolim el 100% de coverage:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
preparePieces()		100 %		100 %	0	7	0	21	0	1
RandomTileGenerator(int)		100 %		100 %	0	4	0	9	0	1
setNumTipus(int)		100 %		100 %	0	4	0	8	0	1
genera()		100 %		100 %	0	2	0	2	0	1
RandomTileGenerator()		100 %		n/a	0	1	0	2	0	1
remainingPieces()		100 %		n/a	0	1	0	1	0	1
getNumTipus()		100 %		n/a	0	1	0	1	0	1
Total	0 of 194	100 %	0 of 22	100 %	0	20	0	43	0	7

L'altre funció rellevant d'aquesta classe és genera(), la qual retorna una peça aleatòria de les que queden a la bossa. A aquesta funció se li fa el test de caixa negra testGeneratesValuesWithinRange() per cadascuna de les dificultats possibles.

```java
// envia una peça restant aleatòria
public int genera() {
    if (bag.isEmpty()) return 0;
    return bag.remove(bag.size() - 1);
}
```

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
preparePieces()		97 %		91 %	1	7	1	20	0	1
genera()		88 %		50 %	1	2	0	2	0	1
RandomTileGenerator(int)		100 %		100 %	0	4	0	9	0	1

```java
// envia una peça restant aleatòria
public int genera() {
    if (bag.isEmpty()) return 0;
    return bag.remove(bag.size() - 1);
}
```

Per tal d'assolir el 100% de coverage, s'havia de cridar a la funció genera() un cop ja s'hagués omplert el taulell. Això es va fer amb testGeneraReturnsZeroWhenBagIs Empty.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
preparePieces()		100 %		100 %	0	7	0	21	0	1
RandomTileGenerator(int)		100 %		100 %	0	4	0	9	0	1
setNumTipus(int)		100 %		100 %	0	4	0	8	0	1
genera()		100 %		100 %	0	2	0	2	0	1
RandomTileGenerator()		100 %		n/a	0	1	0	2	0	1
remainingPieces()		100 %		n/a	0	1	0	1	0	1
getNumTipus()		100 %		n/a	0	1	0	1	0	1
Total	0 of 194	100 %	0 of 22	100 %	0	20	0	43	0	7

Llavors, a RandomTileGenerator() fem statement coverage, decision coverage, condition coverage, loop testing als bucles per omplir la bossa i path coverage.

4.5 GameController

El controlador, encarregat de connectar la vista i el model, rep els clics de l'usuari a través dels botons del tauler, gestiona les seleccions de peces i decideix quan s'ha de realitzar un moviment, actualitzar la vista o canviar l'estat del joc. Una part significativa d'aquesta lògica i dels seus tests parteix del codi base proporcionat per Cristobal Pio al seu GitHub, especialment la gestió dels listeners i la integració amb els estats del joc. Sobre aquesta base, s'han ampliat les funcionalitats i s'ha completat la cobertura necessària per la pràctica.

Testejar aquestes classes implica la creació de tres mock objects, un pel taulell, un pel panel i un per l'estat del joc:

```java
public class GameControllerTest {

    // fem servir classes mock per veure si la vista reacciona
    // al joc com esperavem
    private Board mockBoard;
    private BoardPanel mockPanel;
    private GameController controller;
    private PlayState mockState;
    private JButton[][] fakeButtons;

    @BeforeEach
    void setUp() {
        mockBoard = mock(classToMock: Board.class);
        mockPanel = mock(classToMock: BoardPanel.class);
        mockState = mock(classToMock: PlayState.class);
        //preparem un taulell 2x2 perquè addListeners realment recorri els bucles
        when(mockBoard.getSize()).thenReturn(value: 2);

        fakeButtons = new JButton[2][2];
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                fakeButtons[i][j] = new JButton();
                when(mockPanel.getButton(i, j)).thenReturn(fakeButtons[i][j]);
            }
        }
        controller = new GameController(mockBoard, mockPanel, mockState);
    }
```

La funció handleClick és l'encarregada de desar les posicions seleccionades:

```java
// provar d'eliminar caselles
void handleClick(int x, int y) {
    if (firstSelection == null) {
        firstSelection = new int[]{x, y};
        if (panel != null && panel.getButton(x, y) != null) {
            panel.getButton(x, y).setBackground(java.awt.Color.CYAN);
        }
    } else {
        int x1 = firstSelection[0], y1 = firstSelection[1];
        playState.handleMove(x1, y1, x, y);

        firstSelection = null;
        panel.updateBoard();
    }
}
```

A aquesta funció es fan els següents tests de caixa negra:

- testFirstHandleClick()
- testSecondHandleClickValid()
- testSecondHandleClickInvalid()
- testFirstClickHighlightsButtonAndDoesNotTriggerMove() verifica que es marqui la primera casella seleccionada però no passi res més.
- testVictoria()
- testDerrota()

Amb això, tenim aquest coverage:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
addListeners()		28 %		25 %	2	3	4	7	0	1
handleClick(int, int)		100 %		83 %	1	4	0	9	0	1
GameController(Board, BoardPanel, PlayState)		100 %		n/a	0	1	0	7	0	1
Total	25 of 108	76 %	4 of 10	60 %	3	8	4	23	0	3

I veiem que especialment falten tests per addListeners()

```java
    // controlar accions de l'usuari
    private void addListeners() {
        int size = board.getSize();
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                int row = i;
                int col = j;
                panel.getButton(i, j).addActionListener(new ActionListener() {
                    @Override
                    public void actionPerformed(ActionEvent e) {
                        handleClick(row, col);
                    }
                });
            }
        }
    }
```

Afegim el test de caixa blanca testActionPerformed() i aconseguim el 100% d'statement coverage:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
handleClick(int, int)		100 %		66 %	2	4	0	9	0	1
addListeners()		100 %		100 %	0	3	0	7	0	1
GameController(Board, BoardPanel, PlayState)		100 %		n/a	0	1	0	7	0	1
Total	0 of 108	100 %	2 of 10	80 %	2	8	0	23	0	3

Així obtenim loop coverage de l'addListeners(), i decision coverage i path coverage als tests de handleClick.

Dels estats, destaca també handleMove():

```
public void handleMove(int x1, int y1, int x2, int y2) {
    Board board = _game_session.getBoard();

    //comprovem si es poden eliminar les fitxes seleccionades
    boolean matched = board.tryMatch(x1, y1, x2, y2);

    if (matched) {
        //si s'eliminen, augmentem la puntuació
        _game_session.addScore(points: 100);
        _ui.updateScore(_game_session.getScore());

        //si el taulell queda buit -> victòria
        if (board.isEmpty()) {
            _ui.showDialog(message: "level_completed");
        }
        //si no queden moviments -> derrota
        else if (!board.hasAvailableMoves()) {
            _audio.stopMusic();
            _ui.showDialog(message: "no_moves_left");
            _audio.startMusic(track: "game_over");
        }
    } else {
        //si les fitxes seleccionades no son correctes, so d'error
        _audio.startMusic(track: "error_sound");
        _ui.showDialog(message: "invalid_move");
    }
}
```

Aquesta funció determina si una jugada és vàlida, si cal actualitzar la puntuació i si s'ha arribat a un estat de victòria o de derrota. Aquesta funció també és l'encarregada de mostrar els diàlegs a la vista i de reproduir els sons corresponents segons el resultat. Aquesta es va desenvolupar a partir del codi proporcionat pel professor, però s'han ampliat les funcionalitats. El test de caixa negra que se li ha realitzat és:

- handleMoveUpdatesScoreAndLevelWhenBoardCleared()

Amb aquest obtenim el coverage inicial de:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
handleMove(int, int, int, int)		59 %		33 %	3	4	6	15	0	1
exit()		0 %		0 %	2	2	6	6	1	1
resume()		0 %		n/a	1	1	1	1	1	1
start()		0 %		n/a	1	1	1	1	1	1
restart()		100 %		100 %	0	2	0	9	0	1
configure()		100 %		n/a	0	1	0	3	0	1
pause()		100 %		n/a	0	1	0	3	0	1
PlayState(AudioManager, UIManager, GameSession)		100 %		n/a	0	1	0	5	0	1
Total	47 of 166	71 %	6 of 10	40 %	7	13	14	43	3	8

Després es van afegir els tests de caixa blanca, pels quals varen fer falta quatre mock objects:

```java
private AudioManager audio;
private UIManager ui;
private GameSession gameSession;
private Board board;
private PlayState playState;

@BeforeEach
void setUp() {
    audio = mock(classToMock: AudioManager.class);
    ui = mock(classToMock: UIManager.class);
    gameSession = mock(classToMock: GameSession.class);
    board = mock(classToMock: Board.class);

    when(gameSession.getBoard()).thenReturn(board);

    playState = new PlayState(audio, ui, gameSession);
}
```

I es van implementar els tests:

- handleMove_MatchedAndBoardEmpty_ShowsLevelCompleted()
- handleMove_MatchedAndNoMovesLeft_ShowsGameOver()
- handleMove_NotMatched_PlaysErrorAndShowsInvalidMove()

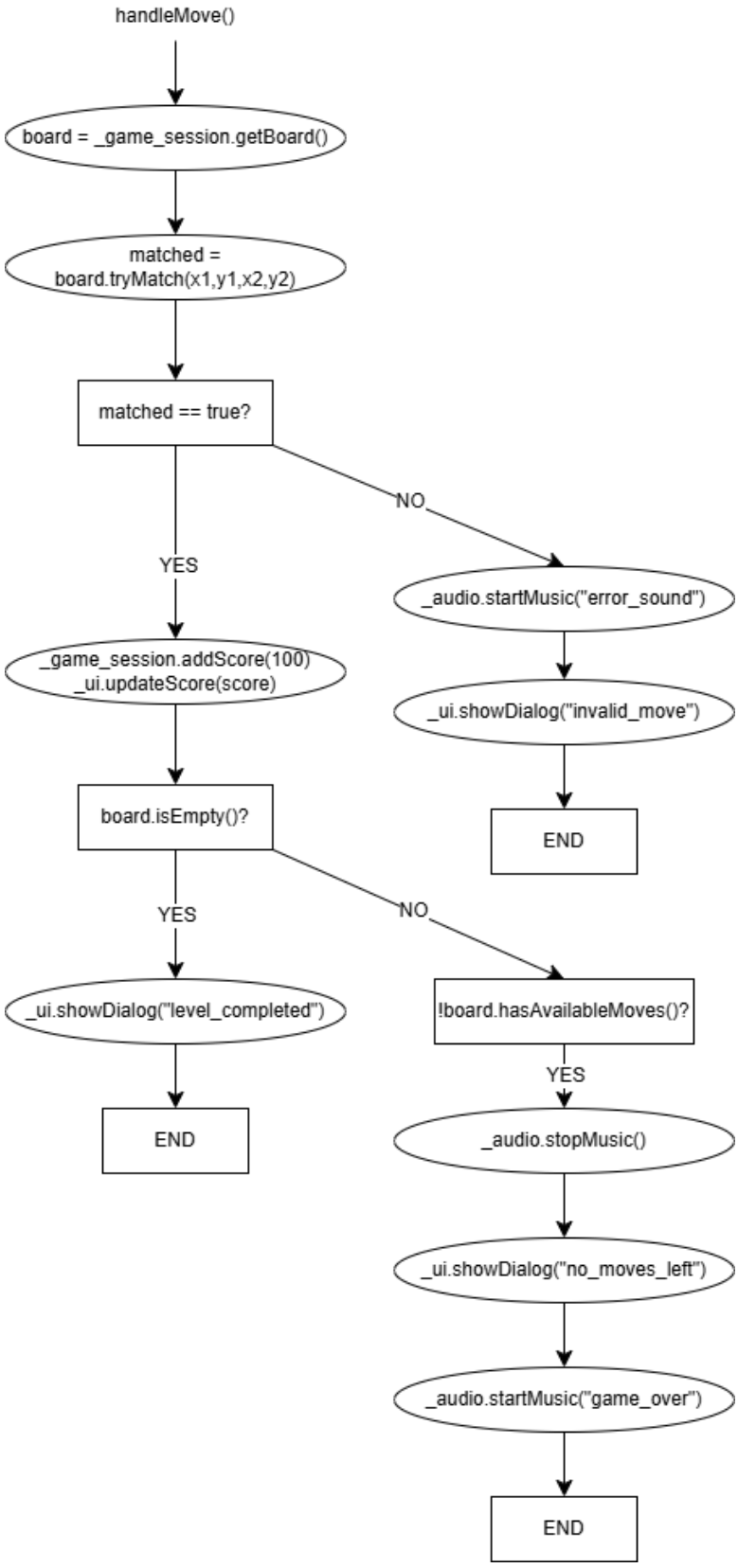
I altres tests pels estats de pausa, exit, etc.

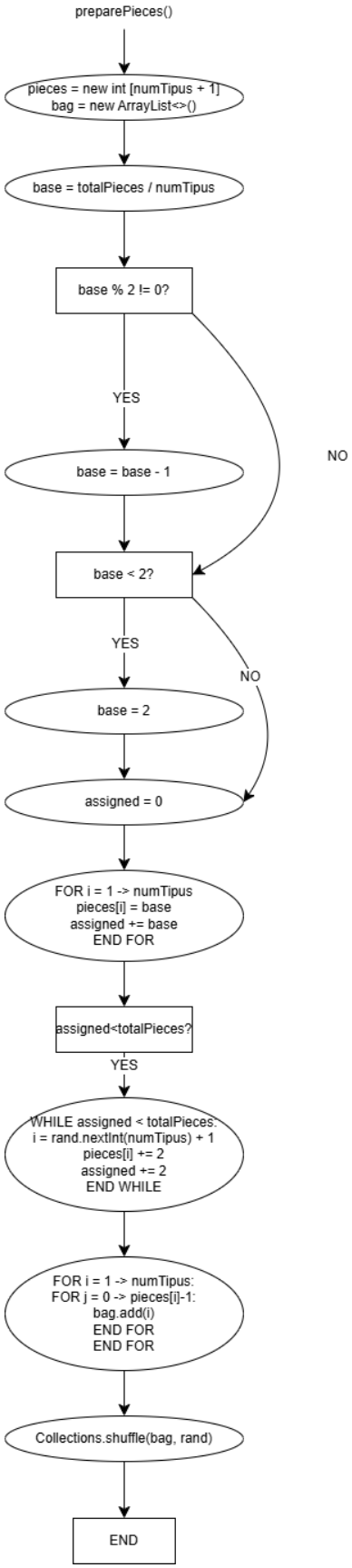
Amb això, assolim el següent coverage, amb statement coverage, decision coverage, condition coverage i path coverage:

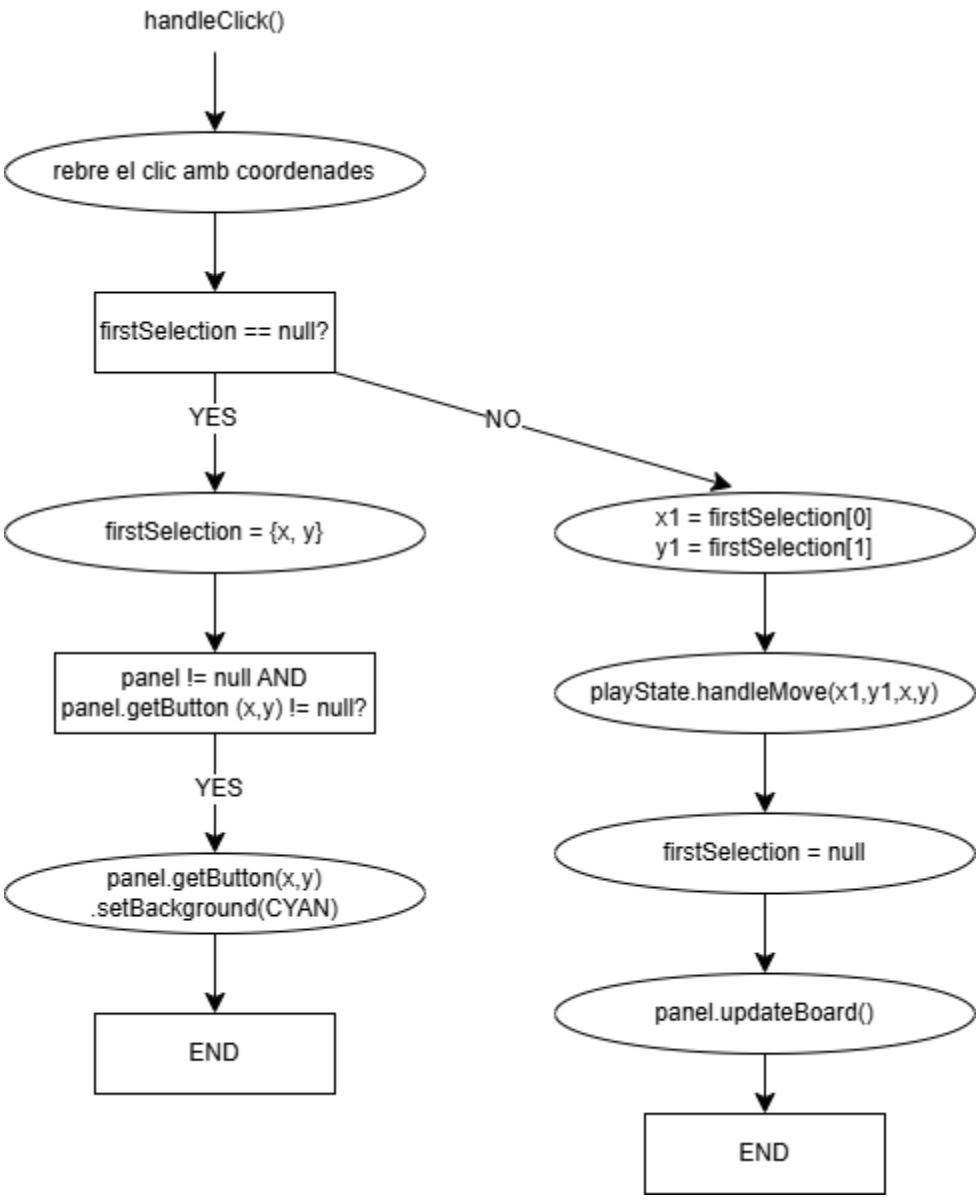
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
exit()		88 %		50 %	1	2	1	6	0	1
handleMove(int, int, int, int)		100 %		83 %	1	4	0	14	0	1
restart()		100 %		100 %	0	2	0	8	0	1
configure()		100 %		n/a	0	1	0	3	0	1
pause()		100 %		n/a	0	1	0	3	0	1
PlayState(AudioManager, UIManager, GameSession)		100 %		n/a	0	1	0	5	0	1
resume()		100 %		n/a	0	1	0	1	0	1
start()		100 %		n/a	0	1	0	1	0	1
Total	2 of 160	98 %	2 of 10	80 %	2	13	1	41	0	8

En resum, hi ha statement coverage a tots els mètodes (a excepció de la GUI, que no ha sigut testejada). S'ha aconseguit decision coverage per tryMatch(), hasAvailableMoves(), handleMove() i handleClick(); condition coverage a tryMatch(), hasAvailableMoves(), preparePieces() i handleMove(); loop testing a loops simples a isEmpty() i preparePieces() i a loops aniuats a hasAvailableMoves() i setTiles(); i finalment, path coverage a handleMove(), preparePieces() i handleClick().

4.6 Diagrames de path coverage




preparePieces()
pieces = new int [numTipus + 1]
bag = new ArrayList<>()
base = totalPieces / numTipus
base % 2 != 0?
YES
NO
base = base - 1
base < 2?
YES
NO
base = 2
assigned = 0
FOR i = 1 -> numTipus
pieces[i] = base
assigned += base
END FOR
assigned<totalPieces?
YES
WHILE assigned < totalPieces:
i = rand.nextInt(numTipus) + 1
pieces[i] += 2
assigned += 2
END WHILE
FOR i = 1 -> numTipus:
FOR j = 0 -> pieces[i]-1:
bag.add(i)
END FOR
END FOR
Collections.shuffle(bag, rand)
END



5. Data-driven testing

Addicionalment, s'ha aplicat la tècnica de data-driven testing a dues funcions del projecte. Aquesta tècnica permet executar un mateix test amb múltiples conjunts de dades d'entrada, facilitant la cobertura en diferents casos sense duplicar codi del test.

En concret, s'ha realitzat a RandomTileGenerator(), amb un conjunt de dades per les diferents dificultats possibles i els seus valors esperats en nombre de peces i tipus. A

partir d'aquestes dades, comprovem que la bossa generada té el nombre correcte de peces, que tots els tipus apareixen en parelles i que no es generen valors fora de rang.

També s'ha realitzat a la classe Board, amb la funció tryMatch(), que envia diferents posicions de peces i els resultats esperats d'intentar fer aquesta combinació.

6. Integració continua i desplegament continu

El projecte incorpora un pipeline de CI/CD, també basat en el codi del professor a GitHub. Amb aquest pipeline, ens podem assegurar de la qualitat i l'estabilitat el codi en cada actualització.

Gràcies a això, es fa una execució automàtica de tests en cada merge a main, i si en falla algun, s'impedeix el merge, garantint que els mètodes ja funcionals quedin així.

També s'ha integrat una validació de l'estil del codi, en aquest cas d'un màxim de 100 caràcters per línia. El pipeline comprova l'estil del codi, el format de línies i indentació i l'absència d'errors.

A més, s'ha fet un desenvolupament per branques, on cada nova funcionalitat o correcció s'ha desenvolupat en una branca independent. Un cop la funcionalitat estava completada i revisada, es fusionava amb el main, passant per totes les comprovacions de CI.