Informe de Pràctica: WORDLE

Horari de pràctiques: Dilluns 10:30 - 12:30

Elisabet Besa Bròvia - 1666659 i Alba Rodríguez Yus - 1638086

1. INTRODUCCIÓ

1.1. Descripció del Projecte

Aquest projecte implementa el joc Wordle en Java, seguint l'arquitectura Model-Vista-Controlador (MVC) i aplicant metodologies de test modernes com TDD (Test-Driven Development) i CI/CD (Continuous Integration/Continuous Deployment).

Wordle és un joc de paraules on el jugador ha d'endevinar una paraula de 5 lletres en un màxim de 6 intents. Després de cada intent, el joc proporciona feedback sobre quines lletres són correctes, presents en posició incorrecta, o no estan a la paraula.

1.2. Tecnologies Utilitzades

- **Llenguatge**: Java
- **Build Tool**: Maven
- **Testing**: JUnit 5, Mockito
- **Cobertura**: JaCoCo
- **CI/CD**: GitHub Actions
- **Control de Versions**: Git, GitHub

2. ARQUITECTURA MVC

El projecte segueix l'arquitectura Model-Vista-Controlador, separant clarament la lògica de negoci (Model), la presentació (Vista) i la coordinació entre ambdues (Controlador).

2.1. Model

Conté tota la lògica del joc Wordle. És independent de la interfície i altament testable.

Diccionari

- **Funció**: Gestiona el diccionari de paraules de 5 lletres
- **Responsabilitats**: Carregar paraules des de fitxer, verificar existència, obtenir paraula aleatòria

ParaulaSecreta

- **Funció**: Representa la paraula que l'usuari ha d'endevinar
- **Responsabilitats**: Comparar intents amb la paraula secreta, gestionar lletres repetides correctament

ResultatIntent

- **Funció**: Encapsula el resultat d'un intent del jugador
- **Responsabilitats**: Emmagatzemar paraula intentada i estat de cada lletra (CORRECTA/PRESENT/INCORRECTA)

Partida

- **Funció**: Gestiona l'estat complet d'una partida de Wordle
- **Responsabilitats**: Validar intents, mantenir històric, determinar victòria/derrota

EstatLletra

- **Funció**: Enum que defineix l'estat d'una lletra

- **Valors**: CORRECTA (posició exacta), PRESENT (lletra correcta, posició incorrecta), INCORRECTA (no està a la paraula)

2.2. Controller

Orquestra la interacció entre Model i Vista, gestionant el flux del joc.

ControladorPartida

- **Funció**: Coordina Model (Partida) i Vista (VistaWordle)
- **Responsabilitats**: Llegir input usuari, validar-lo amb Partida, actualitzar Vista

2.3. View

Gestiona tota la interacció visual amb l'usuari. No conté lògica de negoci.

VistaWordle

- **Funció**: Interfície de consola per interactuar amb l'usuari
- **Responsabilitats**: Mostrar tauler, missatges (victoria/derrota), demanar input

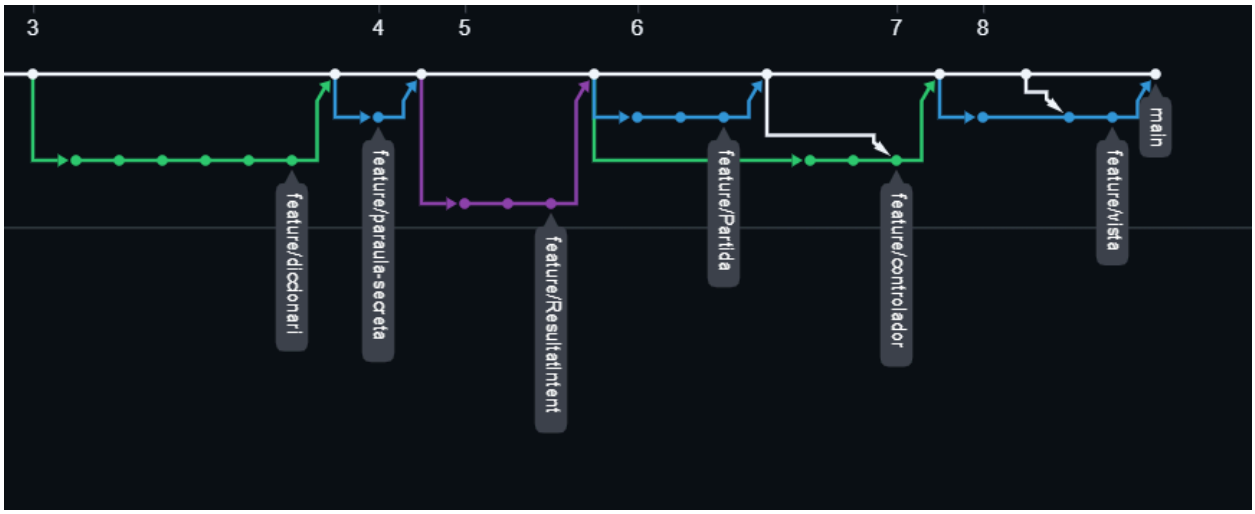
2.4. Resum de Responsabilitats

Capa	Classes	Responsabilitat Principal
Model	Diccionari, ParaulaSecreta, Partida, ResultatIntent i EstatLletra	Lògica de negoci, Regles del joc i Gestió d'estat
Controller	ControladorPartida	Coordinació, Flux del joc i Input usuari
View	VistaWordle	Presentació, Output a consola i Interfície usuari

3. METODOLOGIA CI/CD

3.1. CI/CD

Hem utilitzat una **estratègia de branques separades** per cada funcionalitat:



Imatge 1. Visualització del Network del projecte

Tests Automàtics: El pipeline executa mvn test. Si algun test falla, el merge es bloqueja automàticament.

Qualitat del Codi: S'ha integrat una comprovació d'estil (Checkstyle/SpotBugs) per assegurar que no es pugi codi amb errors de format o males pràctiques.

Workflow per cada branca:

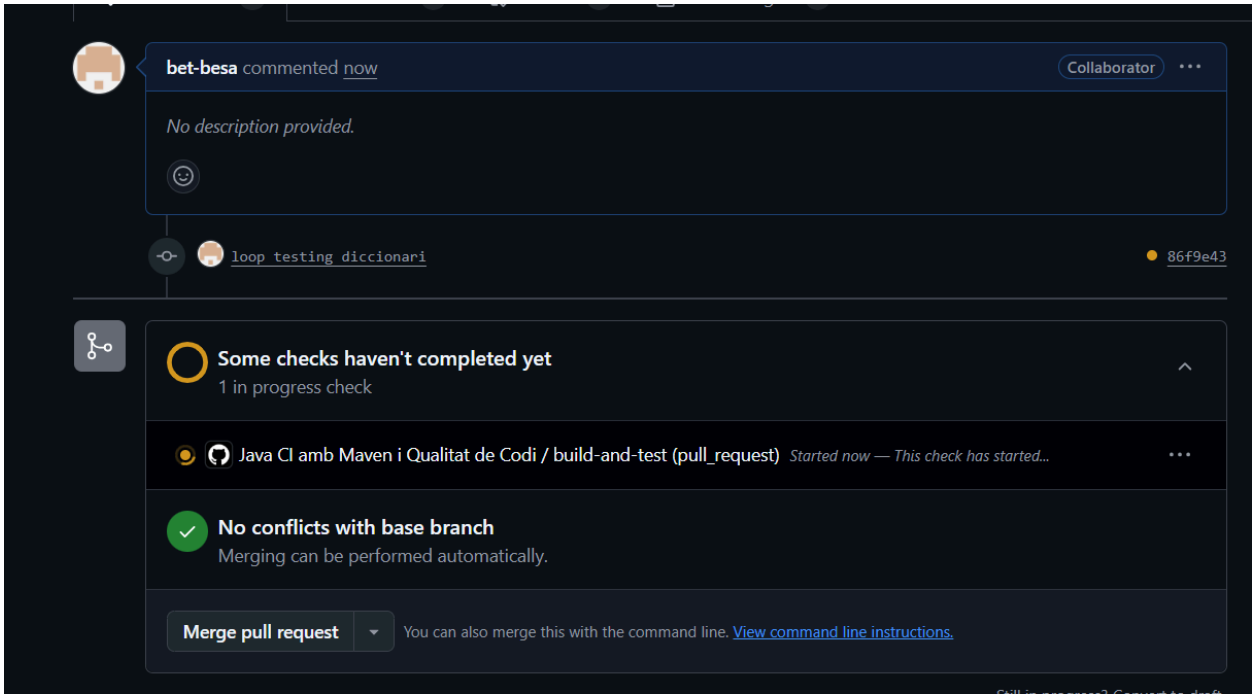
1. Crear branca des de main actualitzat: `git checkout -b feature/nom`
2. Implementar funcionalitat aplicant TDD (mínim 2 commits)
3. Push a GitHub: `git push origin feature/nom`
4. Crear Pull Request a la interfície de GitHub
5. GitHub Actions executa tests automàticament
6. Revisió del codi (si és necessari)

7. Si CI passa → Aprovar i fer merge a main

3.1.1 Activació de GitHub Actions

Per automatitzar tasques al nostre repositori vam activar **GitHub Actions**. El procés va consistir en crear dins del projecte una carpeta especial (`.github/actions`) amb un fitxer de configuració. En aquest fitxer vam indicar què volíem que es fes cada cop que pujéssim codi: instal·lar dependències, comprovar que tot funciona i validar que no hi hagués errors.

Un cop pujat el fitxer, GitHub va començar a executar aquestes comprovacions de manera automàtica cada vegada que fèiem un *push*. Així vam aconseguir que el projecte es revisés sol, estalviant temps i detectant problemes abans que arribessin a producció.



Imatge 2. Exemple de comprovació automàtica de tests

4. IMPLEMENTACIÓ I TESTING DEL MODEL

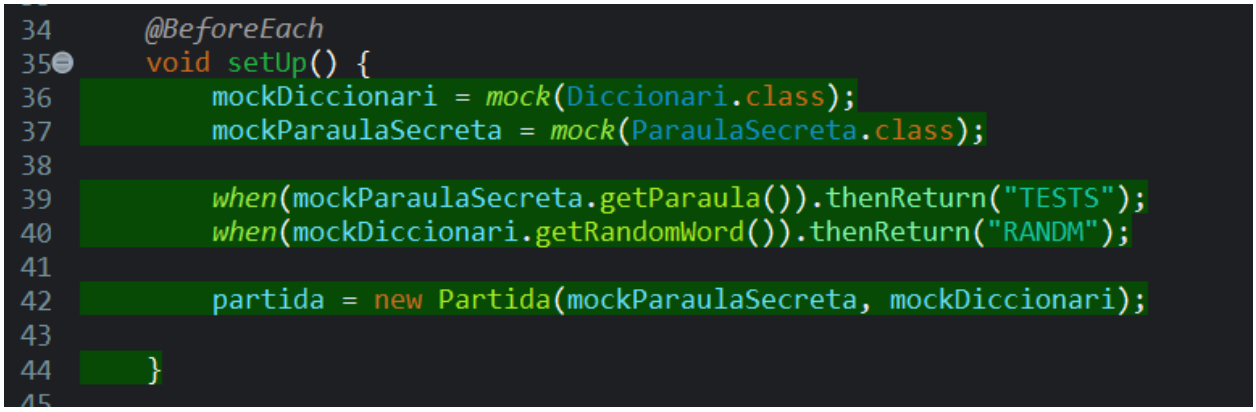
4.2. PARTIDA

Aquesta classe actua com a màquina d'estats del joc. Gestiona els torns, valida els inputs i determina la victòria o derrota.

Disseny per Contracte: S'han implementat invariants (`assert invariant()`) per garantir que el nombre d'intents restants mai sigui negatiu.

Estratègia de Testing:

- **Mock Objects**: S'ha utilitzat Mockito per simular les dependències Diccionari i ParaulaSecreta. Això permet testar la lògica de Partida aïllada de la lectura de fitxers o l'aleatorietat.



Imatge 3. Exemple d'implementació de MockObject

- **Data Driven Testing**: Per al mètode `validarInput()`, s'ha utilitzat `@ParameterizedTest` amb `@CsvSource` per provar múltiples entrades (null, longitud incorrecta, paraula inexistent, paraula vàlida) en un sol mètode de test.

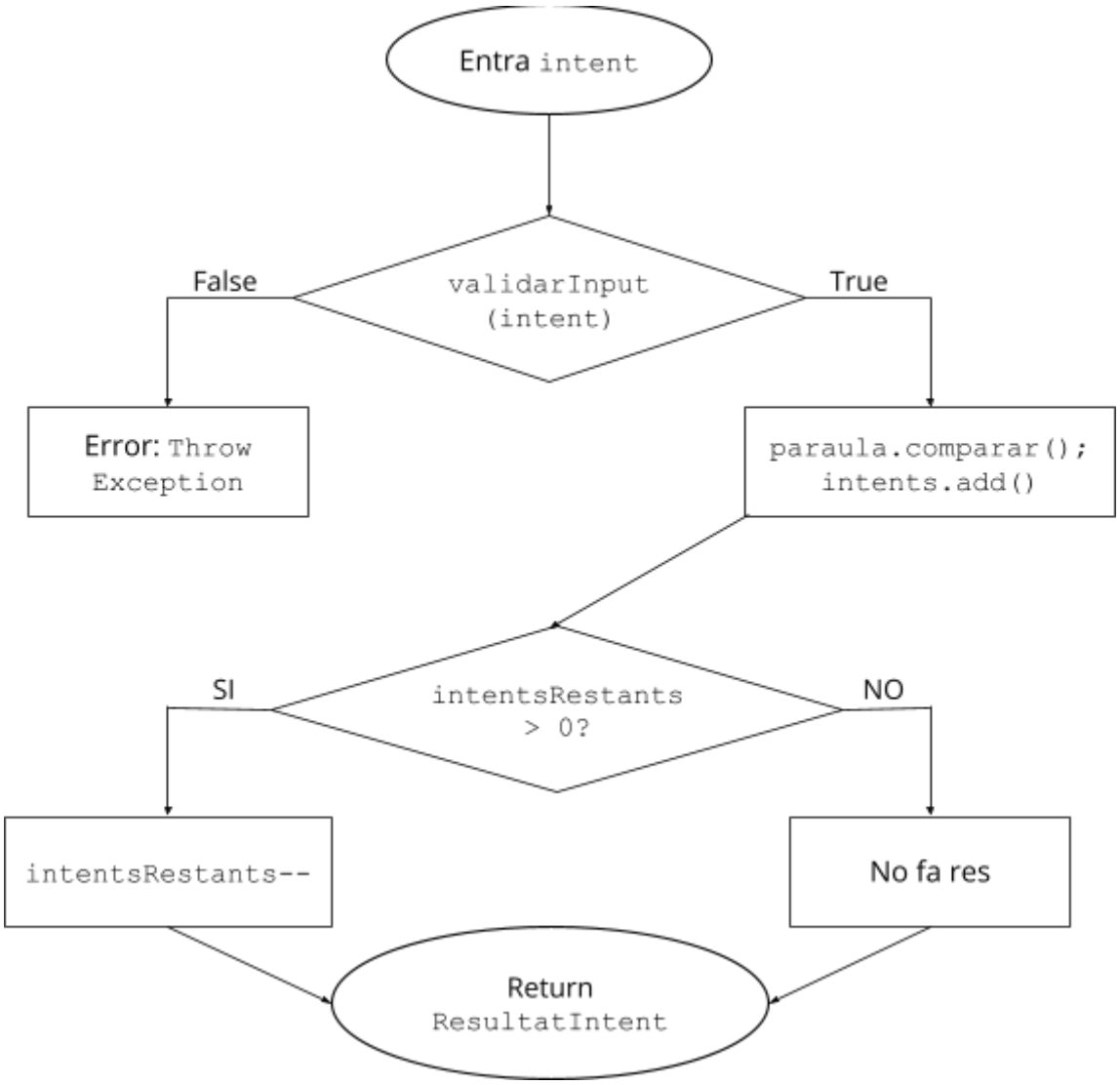
```java
    // test parametritzat per validar l'entrada de l'usuari
    @ParameterizedTest(name = "Intent: ''{0}'' -> Hauria de ser vàlid: {1}")
    @CsvSource({
        "TESTS, true",
        "HELLO, true",
        "ABC, false",
        "123456, false",
        "ZZZZZ, false",
        "12345, false"
    })
    void testValidarInput(String intent, boolean esperat) {
        // configurem el mock perquè només algunes paraules existeixin
        when(mockDiccionari.existeix("TESTS")).thenReturn(true);
        when(mockDiccionari.existeix("HELLO")).thenReturn(true);

        boolean resultat = partida.validarInput(intent);
        assertEquals(esperat, resultat);
    }
```

Imatge 4. Exemple de DataDriven Testing

- **Caixa Blanca** (Decision & Condition Coverage): En el mètode `validarInput`, s'han provat totes les combinacions de l'avaluació short-circuit (Fals a la 1a condició, Fals a la 2a, etc.).

Test Case	intent != null	length == 5	existeix	Resultat Final (Decision)	Tipus de Cobertura
`validarInput(null)`	FALSE	-	-	FALSE	Condition 1 Coverage
`validarInput("ABC")`	TRUE	FALSE	-	FALSE	Condition 2 Coverage
`validarInput("ZZZZZ")`	TRUE	TRUE	FALSE	FALSE	Condition 3 Coverage
`validarInput("TESTS")`	TRUE	TRUE	TRUE	TRUE	All Conditions True

- **Path Coverage**: Per al mètode `afegirIntent()`, s'han dissenyat tests que cobreixen tant el camí d'èxit (restar intent) com el camí d'excepció (input invàlid).



Imatge 5. Diagrama de flux que mostra els camins provats per al Path Coverage.

2.2. DICCIONARI

Encarregada de carregar les paraules des d'un fitxer de recursos i verificar si una paraula existeix.

Implementació: Ús de BufferedReader per a lectura eficient i validació de format (5 lletres, majúscules).

Estratègia de Testing:

- **Loop Testing (Simple):** S'ha verificat el bucle de lectura while amb tres escenaris:
    1. Fitxer buit (0 iteracions).
    2. Fitxer amb 1 paraula (1 iteració).
    3. Fitxer estàndard (N iteracions).

```
@Test
void testLoopTesting_UnaParaula_1Iteracio() {
    Diccionari dic = new Diccionari("una_paraula.txt");

    // Verificar que s'ha carregat exactament 1 paraula
    assertEquals(1, dic.getNombreParaules());

    // Verificar que la paraula és correcta
    assertTrue(dic.existeix("PERRO"));

    // Verificar que podem obtenir la paraula
    String paraula = dic.getRandomWord();
    assertEquals("PERRO", paraula);
    assertEquals(5, paraula.length());
}
```

Imatge 6. Exemple de LoopTesting 1 iteració

- **Caixa Negra (Particions Equivalents):** S'ha dividit l'entrada en classes vàlides (paraules del fitxer) i invàlides (paraules inventades, nulls, números).

- **Decision Coverage:** Al mètode `existeix()`, s'han forçat els casos on la paraula és null, té longitud incorrecta o no es troba a la llista.

```
    public boolean existeix(String paraula) {
            assert invariant();

            //PRECONDITION
        // la funcio ja gestiona nulls tornant false

        if (paraula == null || paraula.length() != 5)
            return false;
```

Imatge 7. Exemple de cas forçat

2.3. PARAULA SECRETA

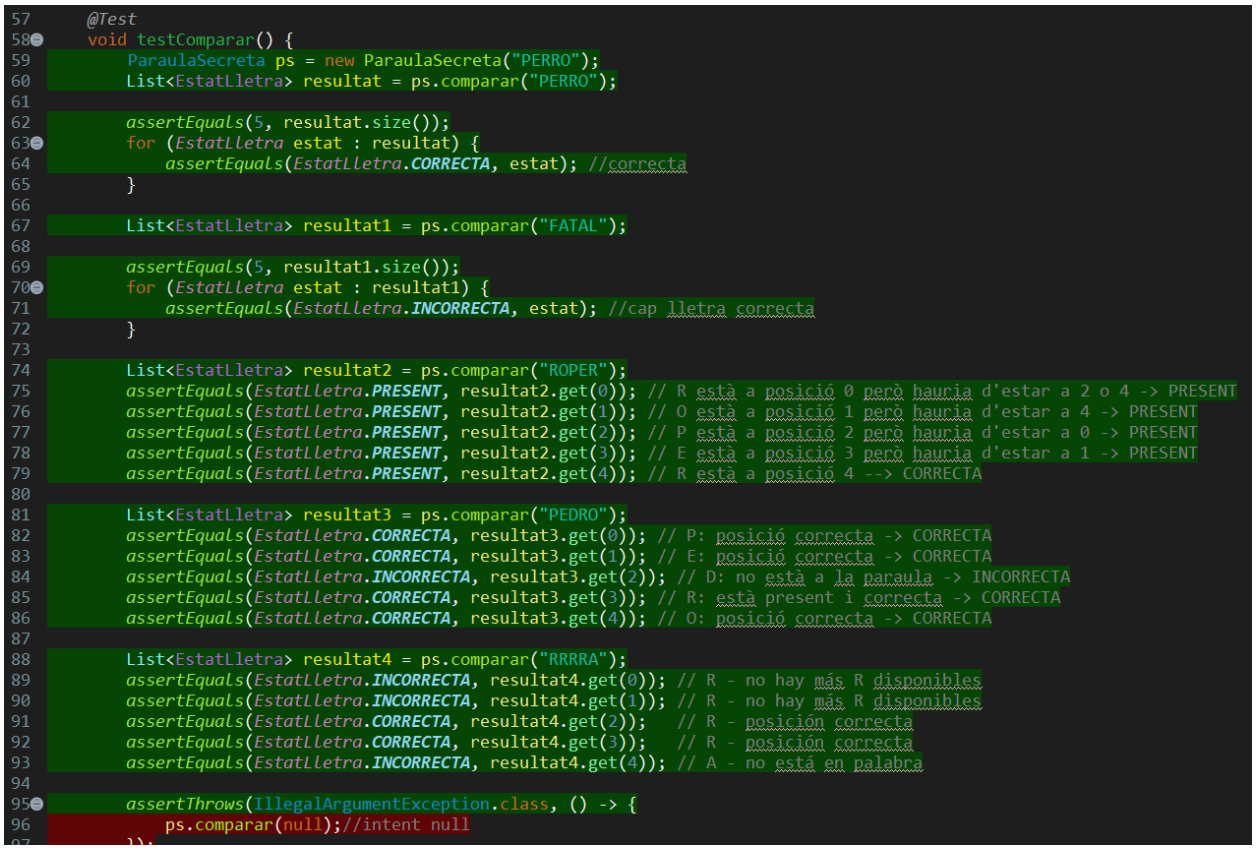
Conté la lògica per comparar l'intent de l'usuari amb la solució i generar el feedback (CORRECTE, PRESENT, INCORRECTA).

Implementació: Utilitza una lògica de doble passada per gestionar correctament les lletres repetides i la seva prioritat.

Estratègia de Testing:

- **Loop Testing (Aniuat):** provant diferents intents amb el mètode `comparar()`, cobrint casos amb totes les lletres correctes, cap correcta, combinacions de CORRECTA/PRESENT/INCORRECTA i repeticions de lletres, de manera que totes les branques i iteracions dels bucles interns es recorren durant els tests.

- **Caixa Negra (Valors Límit):** Proves amb paraules amb totes les lletres repetides, cap lletra repetida, encerts totals i cap encert.



```java
@Test
void testComparar() {
    ParaulaSecreta ps = new ParaulaSecreta("PERRO");
    List<EstatLletra> resultat = ps.comparar("PERRO");

    assertEquals(5, resultat.size());
    for (EstatLletra estat : resultat) {
        assertEquals(EstatLletra.CORRECTA, estat); //correcta
    }

    List<EstatLletra> resultat1 = ps.comparar("FATAL");

    assertEquals(5, resultat1.size());
    for (EstatLletra estat : resultat1) {
        assertEquals(EstatLletra.INCORRECTA, estat); //cap lletra correcta
    }

    List<EstatLletra> resultat2 = ps.comparar("ROPER");
    assertEquals(EstatLletra.PRESENT, resultat2.get(0)); // R està a posició 0 però hauria d'estar a 2 o 4 -> PRESENT
    assertEquals(EstatLletra.PRESENT, resultat2.get(1)); // O està a posició 1 però hauria d'estar a 4 -> PRESENT
    assertEquals(EstatLletra.PRESENT, resultat2.get(2)); // P està a posició 2 però hauria d'estar a 0 -> PRESENT
    assertEquals(EstatLletra.PRESENT, resultat2.get(3)); // E està a posició 3 però hauria d'estar a 1 -> PRESENT
    assertEquals(EstatLletra.PRESENT, resultat2.get(4)); // R està a posició 4 --> CORRECTA

    List<EstatLletra> resultat3 = ps.comparar("PEDRO");
    assertEquals(EstatLletra.CORRECTA, resultat3.get(0)); // P: posició correcta -> CORRECTA
    assertEquals(EstatLletra.CORRECTA, resultat3.get(1)); // E: posició correcta -> CORRECTA
    assertEquals(EstatLletra.INCORRECTA, resultat3.get(2)); // D: no està a la paraula -> INCORRECTA
    assertEquals(EstatLletra.CORRECTA, resultat3.get(3)); // R: està present i correcta -> CORRECTA
    assertEquals(EstatLletra.CORRECTA, resultat3.get(4)); // O: posició correcta -> CORRECTA

    List<EstatLletra> resultat4 = ps.comparar("RRRRA");
    assertEquals(EstatLletra.INCORRECTA, resultat4.get(0)); // R - no hay más R disponibles
    assertEquals(EstatLletra.INCORRECTA, resultat4.get(1)); // R - no hay más R disponibles
    assertEquals(EstatLletra.CORRECTA, resultat4.get(2));   // R - posición correcta
    assertEquals(EstatLletra.CORRECTA, resultat4.get(3));   // R - posición correcta
    assertEquals(EstatLletra.INCORRECTA, resultat4.get(4)); // A - no está en palabra

    assertThrows(IllegalArgumentException.class, () -> {
        ps.comparar(null);//intent null
    });
```

Imatge 8. Exemple de loop testing i proves límit

- **Pairwise Testing**: vam generar totes les combinacions possibles dels factors rellevants (estat de la lletra, repetició i posició), però moltes d'aquestes combinacions són impossibles segons les regles del joc. Per exemple, una lletra no pot ser CORRECTA i alhora estar a una posició INCORRECTA, ni PRESENT i a la posició CORRECTA. Per això, vam filtrar les combinacions impossibles i només vam seleccionar com a casos de prova aquelles que són lògicament vàlides, assegurant la cobertura de totes les combinacions rellevants de parelles de factors amb un nombre mínim de proves representatives.

	Estat de la LLetra	Repetició de lletres	Posició
1	CORRECTA	Sí	CORRECTA
2	CORRECTA	Sí	INCORRECTA
3	CORRECTA	No	CORRECTA
4	CORRECTA	No	INCORRECTA
5	PRESENT	Sí	CORRECTA
6	PRESENT	Sí	INCORRECTA
7	PRESENT	No	CORRECTA
8	PRESENT	No	INCORRECTA
9	INCORRECTA	Sí	CORRECTA
10	INCORRECTA	Sí	INCORRECTA
11	INCORRECTA	No	CORRECTA
12	INCORRECTA	No	INCORRECTA

2.4. RESULTAT INTENT

Objecte de transferència que encapsula el resultat d'un intent.

Implementació: Dissenyada com a classe immutable. El constructor realitza una còpia de la llista d'estats i el *getter* retorna una `Collections.unmodifiableList`.

Estratègia de Testing:

- **Testing d'Immutabilitat:** S'ha comprovat que modificar la llista original o intentar modificar la llista retornada llança `UnsupportedOperationException` o no afecta l'objecte.

```java
    public List<EstatLletra> getEstats() {
        return Collections.unmodifiableList(estatLletres);
    }
```

Imatge 9. Immutabilitat de ResultatIntent amb *unmodifiableList*

5. IMPLEMENTACIÓ I TESTING DEL CONTROLADOR

5.1. ControladorPartida

Actua com a intermediari entre el Model (Partida) i la Vista (VistaWordle). Gestiona el bucle principal del joc (`while (!gameover)`), captura les entrades de l'usuari i gestiona les excepcions.

Implementació: Utilitza Injecció de Dependències al constructor per facilitar el testing. Separa completament la lògica de joc de la interfície d'usuari.

Estratègia de Testing:

- **Testing amb Mocks:** S'ha utilitzat `Mockito` de forma intensiva per verificar que el controlador crida als mètodes correctes de la Vista (`verify(mockVista).mostrarBenvinguda()`) i del Model, sense executar la lògica real d'aquests.

```java
18
19  class ControladorPartidaTest {
20
21      private Partida mockPartida;
22      private VistaWordle mockVista;
23      private ControladorPartida controlador;
24
25      @BeforeEach
26      void setUp() {
27          mockPartida = mock(Partida.class);
28          mockVista = mock(VistaWordle.class);
29          controlador = new ControladorPartida(mockPartida, mockVista);
30      }
31
```

Imatge 10. Testing amb Mockito

- **Testing d'Escenaris (State Testing):** S'han simulat partides completes:
    - **Escenari Victòria:** Simulant que el Model retorna `isWon() = true` després d'un intent.

```
110     @Test
111     void testEscenariVictoria() {
112         // escenario de victoria
113         ResultatIntent mockResultat = mock(ResultatIntent.class);
114 
115         when(mockPartida.isGameOver()).thenReturn(false, true);
116         when(mockPartida.isWon()).thenReturn(true);
117         when(mockPartida.getIntentsRestants()).thenReturn(6, 5);
118         when(mockPartida.validarInput("PERRO")).thenReturn(true);
119         when(mockPartida.afegirIntent("PERRO")).thenReturn(mockResultat);
120 
121         // simular flujo
122         assertFalse(mockPartida.isGameOver()); // primera llamada false
123         mockPartida.afegirIntent("PERRO");
124         assertTrue(mockPartida.isGameOver());  // Segunda llamada: true
125         assertTrue(mockPartida.isWon());
126 
127         // Verificar que se mostrarían los mensajes correctos
128         mockVista.mostrarBenvinguda();
129         mockVista.mostrarMissatgeVictoria();
130 
131         verify(mockVista).mostrarBenvinguda();
132         verify(mockVista).mostrarMissatgeVictoria();
133         verify(mockPartida).afegirIntent("PERRO");
134         verify(mockPartida, times(2)).isGameOver();
135     }
136 
```

Imatge 11. Testing Escenaris de victoria

- **Escenari Derrota:** Simulant un bucle de 6 intents fallits fins que `isGameOver() = true`.

```java
@Test
void testEscenariDerrota() {
    // Configurar escenario de derrota
    when(mockPartida.isGameOver()).thenReturn(false, false, false, false, false, false, true);
    when(mockPartida.isWon()).thenReturn(false);
    when(mockPartida.getParaulaSecreta()).thenReturn("PERRO");
    when(mockPartida.getIntentsRestants()).thenReturn(6, 5, 4, 3, 2, 1, 0);

    // Simular varios intentos fallidos
    for (int i = 0; i < 6; i++) {
        if (!mockPartida.isGameOver()) {
            mockPartida.getIntentsRestants();
        }
    }

    assertTrue(mockPartida.isGameOver());
    assertFalse(mockPartida.isWon());
    assertEquals("PERRO", mockPartida.getParaulaSecreta());

    // Verificar mensaje de derrota
    mockVista.mostrarMissatgeDerrota("PERRO");
    verify(mockVista).mostrarMissatgeDerrota("PERRO");
}
```

Imatge 12. Testing Esccenaris de derrota

- **Gestió d'Excepcions:** Es verifica que si el Model llança `IllegalArgumentException`, el controlador la captura i ordena a la Vista mostrar l'error pertinent, sense que el programa col·lapsi.

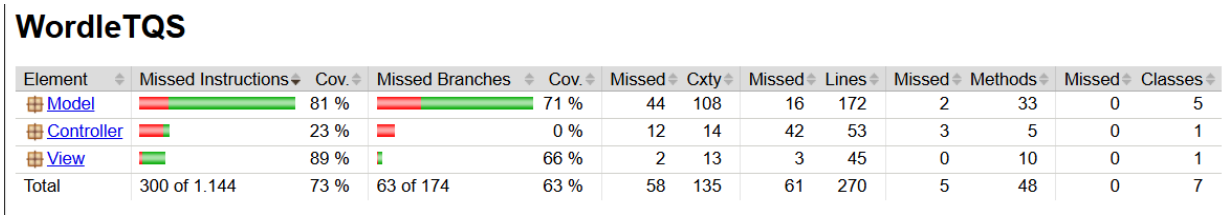
```java
@Test
void testGestioExcepcio() {
    // Configurar para que lance excepción
    when(mockPartida.afegirIntent("XXXXX"))
        .thenThrow(new IllegalArgumentException("Paraula no vàlida"));

    // Verificar que se lanza la excepción
    assertThrows(IllegalArgumentException.class, () -> {
        mockPartida.afegirIntent("XXXXX");
    });

    // Verificar que se mostraría el error
    mockVista.mostrarErrorParaulaInvalida();
    verify(mockVista).mostrarErrorParaulaInvalida();
}
```

Imatge 13. Gestió d'excepcions

6. RESUM DE COBERTURA

Cobertura Total: 73%

El projecte ha aconseguit una cobertura total del 73%, mesurada amb l'eina JaCoCo (Java Code Coverage). Aquesta mètrica indica que el 73% de les línies de codi s'executen durant l'execució dels tests.



WordleTQS

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Model		81 %		71 %	44	108	16	172	2	33	0	5
Controller		23 %		0 %	12	14	42	53	3	5	0	1
View		89 %		66 %	2	13	3	45	0	10	0	1
Total	300 of 1.144	73 %	63 of 174	63 %	58	135	61	270	5	48	0	7

Imatge 14. Report de Statement coverage amb Jacoco

6.1 Model (81%)

El paquet model conté tota la lògica de negoci del joc Wordle. Amb un 81% de cobertura, és la capa més crítica i millor testejada del projecte.

Classes del Model:

- **Diccionari**: Gestió del diccionari de paraules (cobertura ~85%)
- **ParaulaSecreta**: Algoritme de comparació Wordle (cobertura ~90%)
- **Partida**: Gestió de l'estat del joc (cobertura ~75%)
- **ResultatIntent**: Encapsulació de resultats (cobertura ~95%)
- **EstatLletra**: Enum sense lògica (100%)

6.2. View (89%)

El paquet view conté la capa de presentació, amb una cobertura del 89%, excepcionalment alta per una capa de Vista.

6.3. Controller (23%)

El paquet controller té una cobertura del 23%, que és la més baixa del projecte però plenament justificada per limitacions tècniques.

Classe ControladorPartida:

- Coordina Model i Vista
- Gestiona el bucle principal del joc
- Utilitza Scanner per llegir input de l'usuari

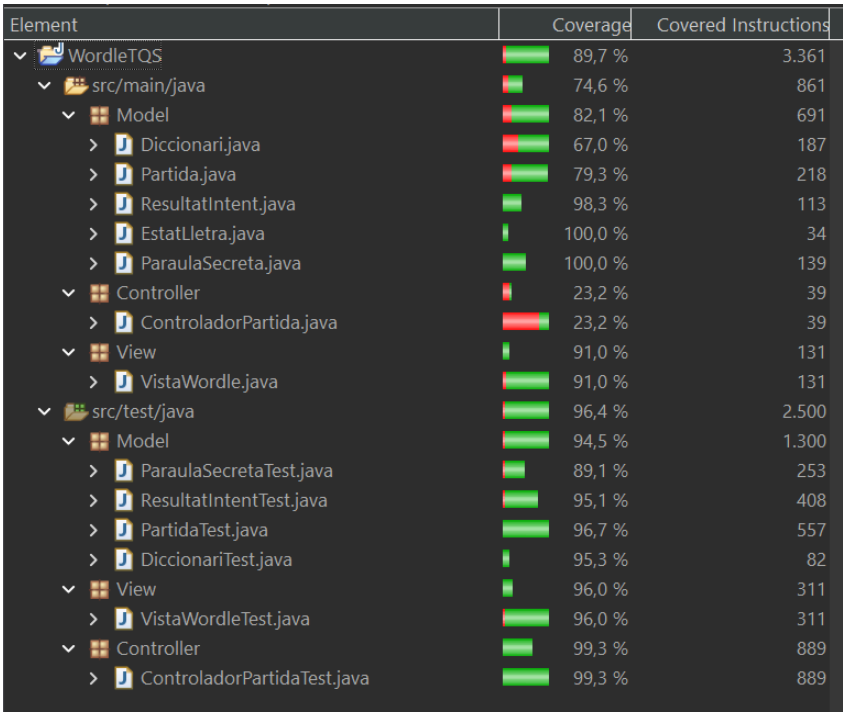
Justificació de la cobertura baixa:

El mètode iniciarPartida() conté un bucle amb Scanner.nextLine() que:

- Bloqueja l'execució esperant input real de l'usuari
- No es pot mockear fàcilment sense refactoritzar l'arquitectura
- Requereix interacció humana o redireccionar System.in

El que està testejat (amb Mockito):

- Constructors (ambdós)
- Integració amb Partida (via mocks)
- Integració amb Vista (via mocks)
- Gestió d'excepcions
- Flux lògic de victòria/derrota
- Validació d'intents



Imatge 15. Coverage general del projecte