Practica1TQS-Wordle

Elisabet Besa Bròvia i Alba Rodriguez Yus
Dilluns 10:30-12:30

Introducció al projecte

Aquest projecte implementa una versió del popular joc Wordle utilitzant la metodologia Test-Driven Development (TDD). El joc consisteix en endevinar una paraula secreta de 5 lletres en un màxim de 6 intents, rebent retroalimentació sobre la posició i presència de cada lletra després de cada intent.

L'arquitectura del projecte segueix el patró Model-View-Controller (MVC), separant clarament la lògica de negoci (Model), la interfície d'usuari (View) i el control del flux de l'aplicació (Controller). S'han implementat precondicions, postcondicions i invariants per assegurar la correcció del codi, i s'han utilitzat diverses tècniques de testing per garantir una cobertura exhaustiva.

Funcionalitat 1: Constructor ParaulaSecreta amb String

Funcionalitat: Constructor que crea una instància de ParaulaSecreta a partir d'un String de 5 lletres. Aquest constructor realitza diverses validacions importants: primer, verifica que el paràmetre no sigui null; segon, comprova que la longitud de la paraula sigui exactament de 5 caràcters; i tercer, converteix automàticament tota la paraula a majúscules per normalitzar l'entrada i facilitar les comparacions posteriors. Si alguna d'aquestes validacions falla, el constructor llança una IllegalArgumentException amb un missatge descriptiu de l'error.

Aquesta funcionalitat és fonamental per al joc, ja que garanteix que totes les paraules secretes tinguin un format consistent i vàlid. La conversió a majúscules és especialment important perquè permet que els jugadors introdueixin les seves respostes en qualsevol combinació de majúscules i minúscules sense afectar la lògica de comparació.

Localització: src/main/java/Model/ParaulaSecreta.java, classe ParaulaSecreta, mètode ParaulaSecreta(String paraula), línies 8-13

Test: src/test/java/Model/ParaulaSecretaTest.java, classe ParaulaSecretaTest, mètode testParaulaSecretaString(), línies 17-43

Tipus de test: Caixa negra amb tècnica de particions equivalents

Descripció detallada del test: El mètode de test implementa una estratègia exhaustiva de particions equivalents per cobrir tots els casos possibles d'entrada. Les particions identificades són:

1. Partició vàlida - Paraula correcta en majúscules: S'utilitza la paraula "PERRO" per verificar que el constructor accepta correctament paraules de 5 lletres en majúscules i les emmagatzema sense modificació.

2. Partició vàlida - Paraula correcta en minúscules: S'utilitza la paraula "perro" per verificar la funcionalitat de conversió automàtica a majúscules. El test comprova que el resultat final sigui "PERRO", demostrant que la normalització funciona correctament.

3. Partició invàlida - Valor null: Es verifica que passar null com a paràmetre llança una IllegalArgumentException. Aquest cas és crucial per prevenir errors de referència nul·la més endavant en l'execució.

4. Partició invàlida - Paraula amb menys de 5 lletres: S'utilitza "GAT" (3 lletres) per verificar que el constructor rebutja paraules massa curtes. Aquest test assegura que el joc mantingui la consistència de tenir sempre paraules de 5 lletres.

5. Partició invàlida - Paraula amb més de 5 lletres: S'utilitza "ORDENADOR" (9 lletres) per verificar que el constructor rebutja paraules massa llargues.

6. Partició invàlida - String buit: Es verifica que un string buit ("") també llança IllegalArgumentException, cobrint el cas límit d'una entrada amb longitud zero.

Funcionalitat 2: Constructor ParaulaSecreta amb Diccionari

Funcionalitat: Aquest constructor alternatiu permet crear una ParaulaSecreta de manera aleatòria a partir d'un diccionari de paraules vàlides. La implementació delega la selecció de la paraula al mètode getRandomWord() del diccionari i després crida al constructor principal amb el String retornat. Aquesta funcionalitat és essencial per a la jugabilitat del Wordle, ja que permet generar partides amb paraules diferents cada vegada, mantenint l'element de sorpresa i rejugabilitat del joc.

El disseny d'aquest constructor segueix el principi de reutilització de codi, ja que no duplica la lògica de validació sinó que la delega al constructor que accepta un String. Això garanteix que totes les paraules, independentment d'on provinguin, passin pels mateixos controls de qualitat.

Localització: src/main/java/Model/ParaulaSecreta.java, classe ParaulaSecreta, mètode ParaulaSecreta(Diccionari dicc), línies 15-17

Test: src/test/java/Model/ParaulaSecretaTest.java, classe ParaulaSecretaTest, mètode testParaulaSecretaDiccionari(), línies 46-55

Tipus de test: Caixa negra amb tècnica de mockups utilitzant Mockito

Descripció detallada del test: Aquest test utilitza Mockito per crear un mock del diccionari, el que permet aïllar la funcionalitat del constructor de la implementació real del diccionari. El test segueix aquests passos:

1. Creació del mock: S'utilitza mock(Diccionari.class) per crear un objecte simulat que implementa la interfície del diccionari sense necessitat de carregar un fitxer real de paraules.

2. Configuració del comportament: S'utilitza `when (mockDicc. getRandomWord()).thenReturn("PERRO")` per especificar que quan es cridi getRandomWord() sobre el mock, aquest ha de retornar "PERRO". Això garanteix un comportament determinista del test.

3. Execució: Es crea la ParaulaSecreta utilitzant el constructor amb el mock del diccionari.

4. Verificacions: El test realitza dues verificacions importants. Primer, comprova que la paraula emmagatzemada sigui "PERRO" utilitzant assertEquals. Segon, utilitza verify(mockDicc).getRandomWord() per confirmar que el mètode getRandomWord() del diccionari s'ha cridat exactament una vegada, assegurant que el constructor efectivament delegui la selecció de la paraula al diccionari.

Funcionalitat 3: Mètode comparar de ParaulaSecreta

Funcionalitat: El mètode comparar() és el cor de la lògica del joc Wordle. Aquest mètode accepta un intent del jugador (un String de 5 lletres) i retorna una llista de 5 estats (EstatLletra) que indiquen la relació de cada lletra de l'intent amb la paraula secreta. Els possibles estats són:

- CORRECTA: La lletra està a la posició correcta

- PRESENT: La lletra existeix a la paraula secreta però en una altra posició

- INCORRECTA: La lletra no existeix a la paraula secreta

La implementació utilitza un algorisme sofisticat de dues passades per gestionar correctament els casos amb lletres duplicades. A la primera passada, marca totes les lletres que estan en la posició exacta com a CORRECTA i les marca com a "usades". A la segona passada, per a les lletres que no són correctes, busca si existeixen en altres posicions de la paraula secreta que encara no hagin estat marcades com a usades. Aquest enfocament evita que una mateixa lletra de la paraula secreta es compti múltiples vegades quan hi ha duplicats a l'intent.

Localització: src/main/java/Model/ParaulaSecreta.java, classe ParaulaSecreta, mètode comparar(String intent), línies 23-67

Test: src/test/java/Model/ParaulaSecretaTest.java, classe ParaulaSecretaTest, mètode testComparar(), línies 58-106

Tipus de test: Caixa negra amb tècnica de particions equivalents i valors límit

Descripció detallada del test: Aquest test és particularment exhaustiu perquè la funcionalitat de comparació és crítica per al joc. S'han identificat i testat les següents particions i casos límit:

1. Cas perfecte - Intent correcte: S'utilitza la mateixa paraula "PERRO" com a intent i com a paraula secreta. El test verifica que totes les 5 lletres retornin EstatLletra.CORRECTA.

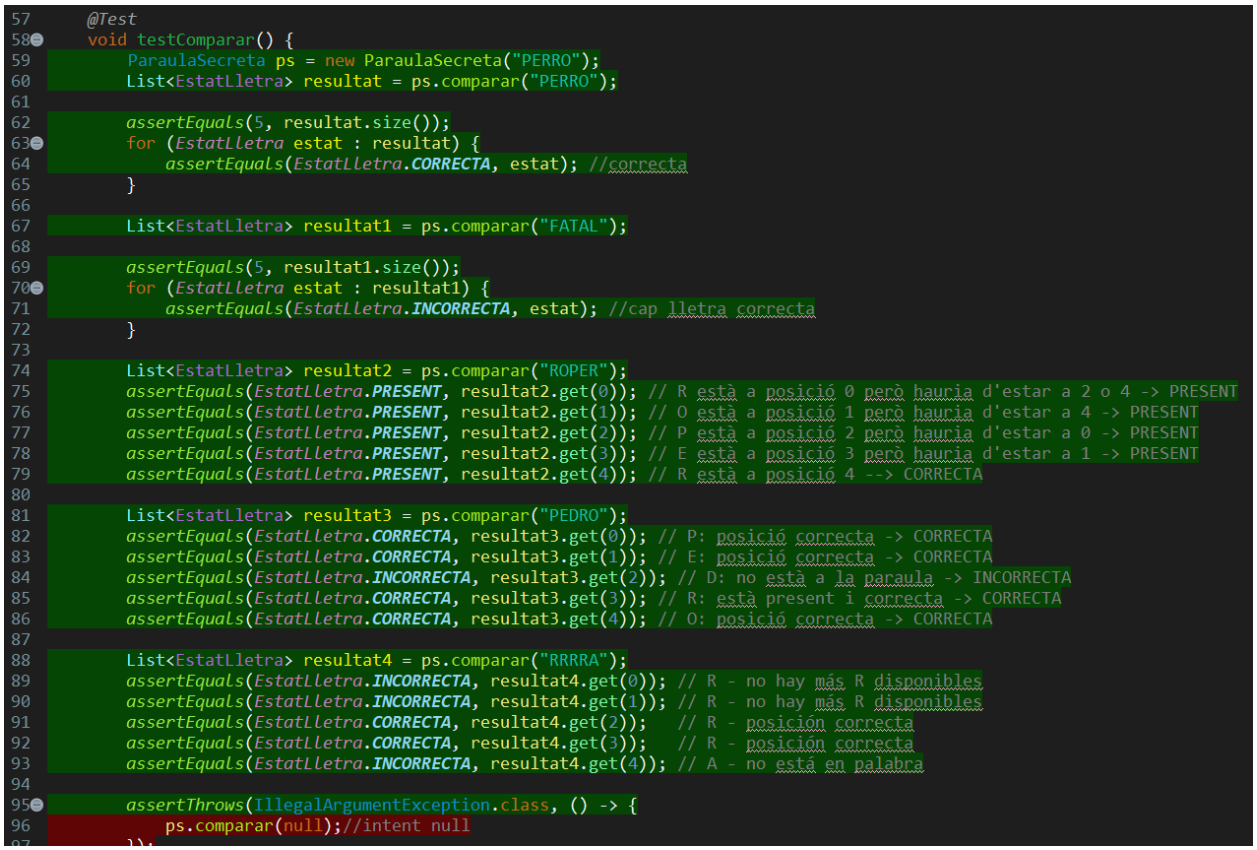
2. Cas oposat - Cap lletra correcta: S'utilitza "FATAL" com a intent contra "PERRO". Com que cap lletra de "FATAL" apareix a "PERRO", el test verifica que totes les lletres retornin EstatLletra.INCORRECTA.

3. Cas de lletres presents però mal posicionades: S'utilitza "ROPER" com a intent. Aquest cas és interessant perquè totes les lletres de l'intent existeixen a "PERRO" però en diferents posicions. El test verifica individualment cada posició: R a la posició 0 hauria d'estar a la 2 o 4 (PRESENT), O a la posició 1 hauria d'estar a la 4 (PRESENT), etc.

4. Cas mixt - Combinació d'estats: S'utilitza "PEDRO" contra "PERRO". Aquest cas genera una combinació dels tres estats possibles: P, E, R, O a les posicions 0, 1, 3, 4 són CORRECTA; D a la posició 2 és INCORRECTA (no existeix a PERRO).

5. Cas complex de duplicats: Aquest és el cas més sofisticat. S'utilitza "RRRRA" contra "PERRO". La paraula secreta només té dues R (a les posicions 2 i 3). El test verifica que: les dues primeres R (posicions 0 i 1) siguin INCORRECTA perquè no hi ha més R disponibles després d'assignar les correctes; les R a les posicions 2 i 3 siguin CORRECTA; i la A final sigui INCORRECTA. Aquest cas és crucial perquè demostra que l'algorisme gestiona correctament els duplicats sense comptar la mateixa lletra múltiples vegades.

6. Validació de precondicions: El test també verifica que el mètode llança IllegalArgumentException per intents invàlids: null, menys de 5 lletres, més de 5 lletres.



```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
        });
```

Funcionalitat 4: Mètode esIgual de ParaulaSecreta

Funcionalitat: El mètode esIgual() proporciona una manera simple i directa de verificar si una paraula donada és exactament igual a la paraula secreta. A diferència del mètode comparar() que retorna informació detallada sobre cada lletra, aquest mètode només retorna un booleà indicant si hi ha coincidència exacta. La implementació converteix el paràmetre a majúscules abans de comparar, assegurant que la comparació sigui insensible a majúscules/minúscules. Aquest mètode és útil internament per determinar ràpidament si un intent ha estat correcte sense necessitat de processar els estats individuals de cada lletra.

Localització: src/main/java/Model/ParaulaSecreta.java, classe ParaulaSecreta, mètode esIgual(String altreParaula), línies 69-71

Test: src/test/java/Model/ParaulaSecretaTest.java, classe ParaulaSecretaTest, mètode testEsIgual(), línies 108-123

Tipus de test: Caixa negra amb tècnica de particions equivalents

Descripció detallada del test: El test cobreix sistemàticament diferents particions d'entrada:

1. Partició vàlida - Coincidència exacta en majúscules: Verifica que "PERRO" és igual a "PERRO".

2. Partició vàlida - Coincidència amb minúscules: Verifica que "perro" també es considera igual a "PERRO", demostrant la conversió automàtica.

3. Partició vàlida - Coincidència amb combinació de majúscules i minúscules: Verifica casos com "PeRrO" per assegurar que qualsevol combinació funcioni correctament.

4. Partició invàlida - Paraules completament diferents: Utilitza "GATOS" i "LAGOS" per verificar que paraules diferents retornin false.

5. Partició invàlida - Paraules similars: Utilitza "PERRI" i "PERRA" per verificar que fins i tot paraules molt similars (amb només una lletra diferent) retornin false. Aquest cas és important perquè en el context del Wordle, només la coincidència exacta compta com a victòria.

Funcionalitat 5: Constructor de Diccionari

Funcionalitat: El constructor del Diccionari és responsable de carregar totes les paraules vàlides des d'un fitxer de text al sistema. La implementació utilitza un BufferedReader per llegir el fitxer línia per línia, afegint cada paraula a una llista interna després de convertir-la a majúscules. El constructor implementa diverses validacions: primer, intenta obrir el fitxer i llança IllegalStateException si no existeix o no es pot llegir; segon, verifica que s'hagi carregat almenys una paraula, ja que un diccionari buit violaria els invariants de la classe. Aquest disseny assegura que qualsevol instància de Diccionari sempre contingui almenys una paraula vàlida.

El diccionari és un component fonamental del joc perquè no només proporciona paraules secretes per a noves partides, sinó que també s'utilitza per validar que els intents dels jugadors siguin paraules vàlides del vocabulari acceptat.

Localització: src/main/java/Model/Diccionari.java, classe Diccionari, constructor Diccionari(String nomFitxer)

Test: src/test/java/Model/DiccionariTest.java, classe DiccionariTest, mètode testDiccionari(), línies 10-18

Tipus de test: Caixa negra amb tècnica de particions equivalents

Descripció detallada del test: El test valida dos escenaris principals:

1. Cas vàlid - Càrrega correcta: Utilitza un fitxer de test anomenat "palabrasTest.txt" que conté exactament 4 paraules. El test crea el diccionari i verifica mitjançant assertEquals que getNombreParaules() retorna 4, confirmant que totes les paraules s'han carregat correctament.

2. Cas invàlid - Fitxer inexistent: Intenta crear un diccionari amb un fitxer que no existeix ("no_existeix.txt"). El test verifica que es llança IllegalStateException utilitzant assertThrows, assegurant que el sistema gestiona correctament els errors de fitxer.

Funcionalitat 6: Loop Testing del Diccionari

Funcionalitat: Aquesta funcionalitat es centra específicament en validar el comportament del bucle while que llegeix les paraules del fitxer dins del constructor del Diccionari. El loop testing és una tècnica de caixa blanca que verifica que un bucle funcioni correctament en tres casos fonamentals: zero iteracions (el bucle no s'executa mai), una iteració (el bucle s'executa exactament una vegada), i múltiples iteracions (el bucle s'executa N vegades). Aquesta tècnica és especialment important per detectar errors off-by-one, condicions de terminació incorrectes i problemes amb l'inicialització de variables.

Localització: src/main/java/Model/Diccionari.java, classe Diccionari, bucle while dins del constructor que llegeix línies del fitxer

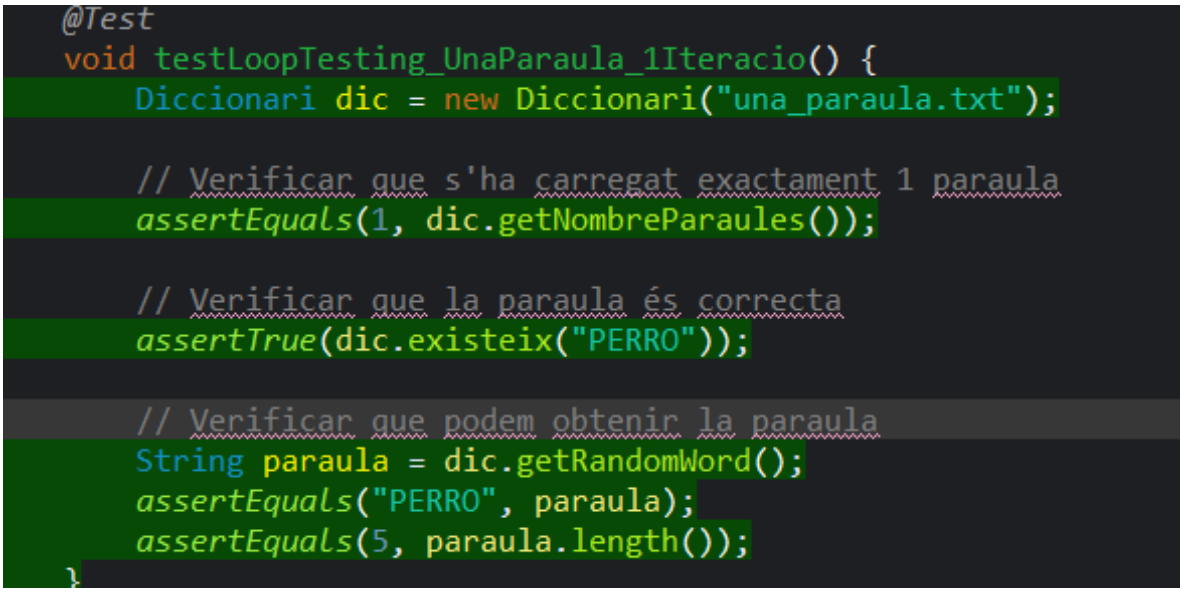
Test: src/test/java/Model/DiccionariTest.java, classe DiccionariTest, tres mètodes: testLoopTesting_FitxerBuit_0Iteracions() (línies 48-54), testLoopTesting_UnaParaula_1Iteracio() (línies 56-70), testLoopTesting_MultiplesParaules_NIteracions() (línies 72-88)

Tipus de test: Caixa blanca amb tècnica de loop testing

Descripció detallada del test: S'han implementat tres tests independents per cobrir els tres casos crítics del loop testing:

Test 1 - Zero iteracions (fitxer buit): Aquest test utilitza un fitxer anomenat "buit.txt" que no conté cap línia de text. Quan el constructor intenta llegir-lo, el bucle while no s'executa ni una sola vegada perquè readLine() retorna immediatament null. Això resulta en un diccionari sense paraules, el que viola l'invariant de la classe que requereix almenys una paraula. El test verifica que aquesta situació llança IllegalStateException amb un missatge descriptiu. Aquest cas és crucial perquè detecta si el codi comprova adequadament que s'hagin carregat paraules abans de considerar el diccionari vàlid.

Test 2 - Una iteració (exactament una paraula): Aquest test utilitza un fitxer anomenat "una_paraula.txt" que conté exactament una línia amb la paraula "PERRO". El bucle s'executa exactament una vegada: llegeix la línia, la converteix a majúscules i l'afegeix a la llista. El test realitza tres verificacions: primer, que getNombreParaules() retorni 1; segon, que existeix("PERRO") retorni true, confirmant que la paraula s'ha afegit correctament; i tercer, que getRandomWord() retorni "PERRO", demostrant que es pot recuperar la paraula. Aquest cas és important perquè verifica el comportament del bucle en el cas mínim vàlid.



```java
@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());
}
```

Test 3 - Múltiples iteracions (N paraules): Aquest test utilitza "palabrasTest.txt" que conté 4 paraules, causant que el bucle s'executi 4 vegades. El test verifica: primer, que getNombreParaules() retorni 4, confirmant que el bucle ha processat totes les línies; segon, que múltiples paraules específiques com "PERRO" i "COCHE" existeixin al diccionari, demostrant que cada iteració del bucle ha afegit correctament la seva paraula; i tercer, que getRandomWord() retorni una paraula vàlida de 5 lletres que existeix al diccionari. Aquest cas verifica que el bucle pot executar-se múltiples vegades sense problemes d'acumulació d'errors.

Funcionalitat 7: Mètode existeix del Diccionari

Funcionalitat: El mètode existeix() és fonamental per a la validació dels intents dels jugadors. Aquest mètode comprova si una paraula donada forma part del diccionari de paraules vàlides. La implementació converteix el paràmetre d'entrada a majúscules abans de buscar-lo a la llista interna de paraules, assegurant que la cerca sigui insensible a majúscules/minúscules. Aquesta característica és important per a l'experiència d'usuari, ja que permet als jugadors escriure els seus intents en qualsevol format sense preocupar-se per les majúscules. El mètode també gestiona correctament casos especials com valors null retornant false en lloc de llançar excepcions.

Localització: src/main/java/Model/Diccionari.java, classe Diccionari, mètode existeix(String paraula)

Test: src/test/java/Model/DiccionariTest.java, classe DiccionariTest, mètode testExisteix(), línies 20-31

Tipus de test: Caixa negra amb tècnica de particions equivalents

Descripció detallada del test: El test utilitza el fitxer "palabrasTest.txt" i verifica múltiples particions d'entrada:

1. Partició vàlida - Paraula existent en minúscules: Verifica que "perro" retorni true, demostrant que les minúscules es gestionen correctament.

2. Partició vàlida - Paraula existent en majúscules: Verifica que "PERRO" també retorni true, confirmant que les majúscules funcionen.

3. Partició invàlida - Paraula no existent: Verifica que "gatos" i "zzzzz" retornin false, ja que no estan al fitxer de test.

4. Partició invàlida - Entrada amb números: Verifica que "12345" retorni false, assegurant que només s'acceptin lletres.

5. Partició invàlida - Longitud incorrecta: Verifica que "gat" (3 lletres) i "ordenador" (9 lletres) retornin false, tot i que podrien ser paraules vàlides, no tenen la longitud requerida de 5 caràcters.

6. Partició invàlida - Valor null: Verifica que null retorni false en lloc de llançar una excepció, demostrant un gestió robusta d'errors.

Funcionalitat 8: Constructor Partida amb ParaulaSecreta

Funcionalitat: Aquest constructor crea una nova partida amb una paraula secreta específica i un diccionari per validar intents. La implementació segueix bones pràctiques de programació defensiva incorporant precondicions explícites i invariants. Les precondicions verifiquen que ni la paraula secreta ni el diccionari siguin null, llançant IllegalArgumentException si aquesta condició no es compleix. Després d'inicialitzar els camps de la classe (paraula secreta, diccionari, llista d'intents buida i 6 intents restants), el constructor invoca el mètode invariant() per verificar que l'objecte s'ha creat en un estat consistent. Aquest disseny basat en contractes ajuda a detectar errors de programació ràpidament i facilita el manteniment del codi.

El mètode invariant() comprova diverses condicions que sempre han de ser certes: que la paraula secreta no sigui null, que el diccionari no sigui null, que la llista d'intents no sigui null, que els intents restants estiguin entre 0 i MAX_INTENTS (6), i que la suma d'intents realitzats més intents restants sempre sigui igual a MAX_INTENTS. Aquest últim invariant és especialment important perquè garanteix la consistència de l'estat de la partida.

Localització: src/main/java/Model/Partida.java, classe Partida, mètode Partida(ParaulaSecreta paraula, Diccionari dicc), línies 33-44, i mètode invariant(), línies 14-31

Test: src/test/java/Model/PartidaTest.java, classe PartidaTest, mètode testPartida(), línies 46-64

Tipus de test: Caixa negra amb tècnica de particions equivalents i mockups

```
34      @BeforeEach
35      void setUp() {
36          mockDiccionari = mock(Diccionari.class);
37          mockParaulaSecreta = mock(ParaulaSecreta.class);
38
39          when(mockParaulaSecreta.getParaula()).thenReturn("TESTS");
40          when(mockDiccionari.getRandomWord()).thenReturn("RANDM");
41
42          partida = new Partida(mockParaulaSecreta, mockDiccionari);
43
44      }
```

Descripció detallada del test: El test utilitza mockups de ParaulaSecreta i Diccionari creats amb Mockito per aïllar la funcionalitat del constructor. El mètode setUp() configura aquests mocks abans de cada test. Les verificacions inclouen:

1. Estat inicial correcte: Crea una nova partida i verifica que l'objecte no sigui null, que la llista d'intents estigui buida (size 0), que hi hagi 6 intents restants, i que la paraula secreta sigui "TESTS" (valor configurat al mock).

2. Validació de paràmetre null per ParaulaSecreta: Utilitza assertThrows per verificar que passar null com a primer paràmetre llança IllegalArgumentException. Això confirma que la precondició s'està comprovant correctament.

3. Validació de paràmetre null per Diccionari: De manera similar, verifica que passar null com a segon paràmetre també llança IllegalArgumentException.

Funcionalitat 9: Validació d'input de Partida

Funcionalitat: El mètode validarInput() és una funció de validació crítica que determina si un intent del jugador és acceptable per processar. Aquest mètode aplica tres criteris de validació en cascada: primer, verifica que l'intent no sigui null; segon, comprova que tingui exactament 5 caràcters de longitud; i tercer, consulta el diccionari per assegurar que la paraula existeix al vocabulari acceptat. Només si les tres condicions es compleixen, el mètode retorna true. Aquest disseny multi-capa de validació és essencial per mantenir la integritat del joc i proporcionar una experiència justa als jugadors, evitant que es provin paraules inventades o amb longitud incorrecta.

La implementació utilitza l'operador lògic AND (&&) amb avaluació de curt-circuit, el que significa que si alguna condició falla, les següents no s'avaluen. Això és eficient i evita possibles errors de referència nul·la quan s'intenta accedir a la longitud d'un string null.

Localització: src/main/java/Model/Partida.java, classe Partida, mètode validarInput(String intent), línies 59-62

Test: src/test/java/Model/PartidaTest.java, classe PartidaTest, mètode testValidarInput(), línies 81-98

Tipus de test: Caixa negra amb tècnica de particions equivalents i tests parametritzats amb JUnit 5

Descripció detallada del test: Aquest test utilitza l'anotació @ParameterizedTest amb @CsvSource de JUnit 5 per provar múltiples casos d'entrada de manera eficient i llegible. La font de dades CSV defineix parells de valors: l'intent a provar i el resultat esperat (true o false). Abans d'executar els casos parametritzats, el test configura el mock del diccionari per retornar true només per a les paraules "TESTS" i "HELLO". Els casos de test són:

1. **"TESTS, true":** Paraula vàlida de 5 lletres que existeix al diccionari.
2. **"HELLO, true":** Altra paraula vàlida de 5 lletres que existeix al diccionari.
3. **"ABC, false":** Paraula massa curta (3 lletres), per tant invàlida.
4. **"123456, false":** Paraula massa llarga (6 caràcters), per tant invàlida.
5. **"ZZZZZ, false":** Paraula de 5 lletres però que no existeix al diccionari.
6. **"12345, false":** Tot i tenir 5 caràcters, no existeix al diccionari.

```
81      // test parametritzat per validar l'entrada de l'usuari
82      @ParameterizedTest(name = "Intent: ''{0}'' -> Hauria de ser vàlid: {1}")
83      @CsvSource({
84          "TESTS, true",
85          "HELLO, true",
86          "ABC, false",
87          "123456, false",
88          "ZZZZZ, false",
89          "12345, false"
90      })
91      void testValidarInput(String intent, boolean esperat) {
92          // configurem el mock perquè només algunes paraules existeixin
93          when(mockDiccionari.existeix("TESTS")).thenReturn(true);
94          when(mockDiccionari.existeix("HELLO")).thenReturn(true);
95
96          boolean resultat = partida.validarInput(intent);
97          assertEquals(esperat, resultat);
98      }
99
```

Aquest enfocament de test parametritzat ofereix diversos avantatges: redueix la duplicació de codi, fa que sigui fàcil afegir nous casos de test simplement afegint línies a la font CSV, i proporciona informació clara sobre quin cas ha fallat si hi ha un error (gràcies al paràmetre 'name' de l'anotació).

Funcionalitat 10: Afegir intent a la Partida

Funcionalitat: Processa un intent vàlid del jugador, actualitza l'estat de la partida i retorna el resultat. Inclou precondicions (partida no acabada, intent vàlid) i postcondicions (intent afegit, intents restants decrementats).

Localització: src/main/java/Model/Partida.java, línies 64-93

Test: src/test/java/Model/PartidaTest.java, testAfegirIntent(), línies 100-130. Tipus: Caixa negra amb mockups. Casos: victòria (encerta en primer intent), derrota (6 intents fallits). Verifica postcondicions amb assertions.

Funcionalitat 11: Verificació de victòria

Funcionalitat: Determina si el jugador ha guanyat verificant si l'últim intent és correcte (totes lletres CORRECTA).

Localització: src/main/java/Model/Partida.java, mètode isWon(), línies 95-100

Test: src/test/java/Model/PartidaTest.java, testIsWon(), línies 132-152. Tipus: Caixa negra amb mockups. Casos: sense intents (false), intent fallit (false), intent correcte (true).

Funcionalitat 12: Verificació de final de partida

Funcionalitat: Determina si la partida ha acabat per victòria o exhauriment d'intents. És la funció de control principal per al bucle del joc.

Localització: src/main/java/Model/Partida.java, mètode isGameOver(), línies 102-104

Test: src/test/java/Model/PartidaTest.java, testIsGameOver(), línies 154-179. Tipus: Caixa negra amb mockups. Casos: game over per victòria, game over per 6 intents fallats.

Funcionalitat 13: Obtenció d'intents realitzats

Funcionalitat: Retorna una llista immutable dels intents realitzats fins al moment. La immutabilitat protegeix l'estat intern de modificacions externes.

Localització: src/main/java/Model/Partida.java, mètode getIntents(), línies 106-108

Test: src/test/java/Model/PartidaTest.java, testGetIntents(), línies 182-191. Tipus: Caixa negra. Verifica que retorna llista correcta i que llançar UnsupportedOperationException en intentar modificar-la.

Funcionalitat 14: ControladorPartida - Gestió d'escenaris

Funcionalitat: El controlador orquestra el flux complet del joc: demanar intents, validar-los, processar-los i mostrar resultats fins que la partida acabi. Gestiona els escenaris de victòria i derrota.

Localització: src/main/java/Controller/ControladorPartida.java, mètode iniciarPartida()

Test: src/test/java/Controller/ControladorPartidaTest.java, testEscenariVictoria() i testEscenariDerrota(), línies 110-159. Tipus: Caixa negra amb mockups. Simula fluxos complets de joc i verifica crides correctes a la vista.

Funcionalitat 15: ControladorPartida - Flux complet

Funcionalitat: Simulació i validació d'un flux realista de joc amb múltiples intents: alguns incorrectes seguits d'un encert.

Localització: src/main/java/Controller/ControladorPartida.java, mètode iniciarPartida()

Test: src/test/java/Controller/ControladorPartidaTest.java, testFluxComplet(), línies 203-236. Tipus: Caixa negra amb mockups. Simula 3 intents consecutius i verifica estats intermedis.

```
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
```

```
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
```

```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");
    }
```

```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();
    }
```

Funcionalitat 16: ResultatIntent

Funcionalitat: Classe immutable que encapsula el resultat d'un intent: la paraula provada i els estats de cada lletra. Proporciona el mètode esCorrecte() per determinar si totes les lletres són correctes.

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

Immutabilitat de ResultatIntent amb *unmodifiableList*

Localització: src/main/java/Model/ResultatIntent.java

Test: src/test/java/Model/ResultatIntentTest.java. Tipus: Caixa negra. Casos: tots CORRECTA (true), algun INCORRECTA o PRESENT (false).

Funcionalitat 17: VistaWordle - Interfície d'usuari

Funcionalitat: Gestiona tota la interacció amb l'usuari per consola: missatges de benvinguda, visualització del tauler amb colors ANSI, missatges d'error, victòria i derrota, i captura d'input de l'usuari.

Localització: src/main/java/View/VistaWordle.java

Test: src/test/java/View/VistaWordleTest.java, testCridesVista(), línies 238-259. Tipus: Caixa negra amb mockups. Verifica que tots els mètodes de visualització es poden invocar correctament.

Anàlisi de tècniques de testing utilitzades

Al llarg d'aquest projecte s'han aplicat diverses tècniques de testing de caixa negra i caixa blanca per assegurar la màxima cobertura i qualitat del codi:

Tècniques de Caixa Negra

1. Particions Equivalents: S'han identificat classes d'equivalència per a tots els inputs possibles, testejant almenys un representant de cada classe vàlida i invàlida. Per exemple, en la validació de paraules: vàlides de 5 lletres, invàlides per longitud (curtes/llargues), invàlides per contingut (null, números).

2. Anàlisi de Valors Límit: S'han testat específicament els valors límit com strings buits, paraules d'exactament 4 i 6 lletres, i el cas de 0 intents restants.

3. Tests Parametritzats: Utilitzant @ParameterizedTest i @CsvSource de JUnit 5, s'han creat tests eficients que proven múltiples casos amb una sola implementació de test, millorant la mantenibilitat.

4. Mockups amb Mockito: S'han utilitzat mocks per aïllar components i testar-los independentment. Això és especialment important en el ControladorPartida, on es mocken tant el Model com la View per testar només la lògica de control. Els mocks també permeten verificar que els mètodes s'han cridat amb els paràmetres correctes i el nombre de vegades esperat.

```
34      @BeforeEach
35      void setUp() {
36          mockDiccionari = mock(Diccionari.class);
37          mockParaulaSecreta = mock(ParaulaSecreta.class);
38
39          when(mockParaulaSecreta.getParaula()).thenReturn("TESTS");
40          when(mockDiccionari.getRandomWord()).thenReturn("RANDM");
41
42          partida = new Partida(mockParaulaSecreta, mockDiccionari);
43
44      }
```

5. Pairwise Testing: S'utilitza per al mètode comparar() de la classe ParaulaSecreta. Es van generar totes les combinacions possibles dels factors rellevants (estat de la lletra, repetició i posició). Es van filtrar les combinacions

impossibles (per exemple, una lletra no pot ser CORRECTA i INCORRECTA alhora). Només es van seleccionar com a casos de prova aquelles combinacions 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

Tècniques de Caixa Blanca

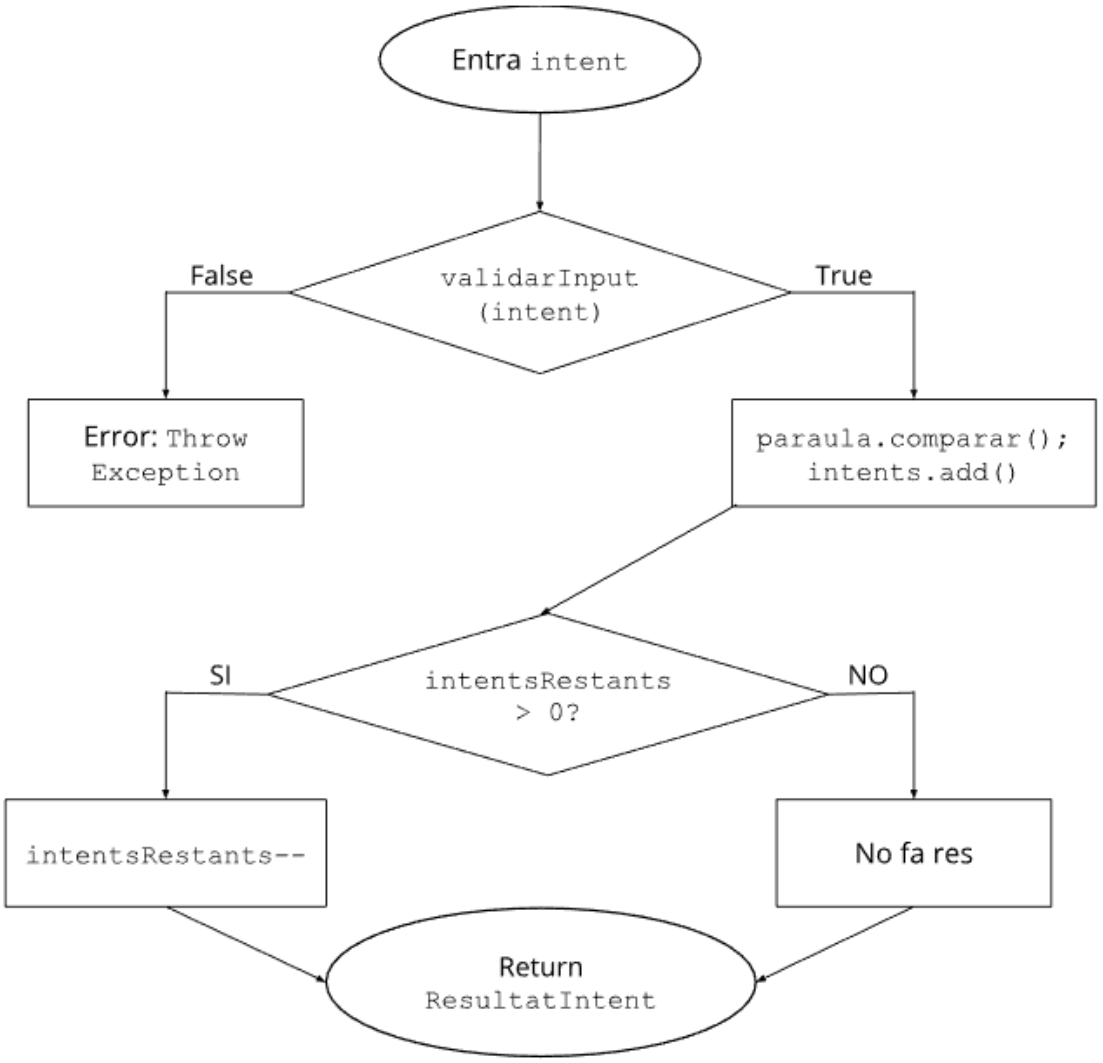
1. Loop Testing: S'han implementat tests específics per als bucles, cobrint els casos de 0 iteracions (fitxer buit), 1 iteració (una paraula), i N iteracions (múltiples paraules). Aquesta tècnica ha estat crucial per detectar errors off-by-one i problemes de inicialització.

```
@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());
}
```

2. Statement Coverage: S'ha aconseguit alta cobertura de sentències assegurant que cada línia de codi executable es provi almenys una vegada. Això s'ha verificat amb JaCoCo.

3. Branch Coverage: S'han testat totes les branques de les estructures condicionals (if/else), assegurant que tant el cas true com el false de cada condició s'executen.

4. 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).



5. Decision Coverage i Condition Coverage: Per al mètode validarInput() de la classe Partida, es van provar totes les combinacions de l'avaluació de curt-circuit de les condicions. Això va incloure forçar que la 1a condició (null), la 2a condició (longitud), i la 3a condició (existència al diccionari) fossin falses per separat per validar tots els camins de decisió.

```
81      // test parametritzat per validar l'entrada de l'usuari
82      @ParameterizedTest(name = "Intent: ''{0}'' -> Hauria de ser vàlid: {1}")
83      @CsvSource({
84          "TESTS, true",
85          "HELLO, true",
86          "ABC, false",
87          "123456, false",
88          "ZZZZZ, false",
89          "12345, false"
90      })
91      void testValidarInput(String intent, boolean esperat) {
92          // configurem el mock perquè només algunes paraules existeixin
93          when(mockDiccionari.existeix("TESTS")).thenReturn(true);
94          when(mockDiccionari.existeix("HELLO")).thenReturn(true);
95
96          boolean resultat = partida.validarInput(intent);
97          assertEquals(esperat, resultat);
98      }
99
```

Programació per Contracte

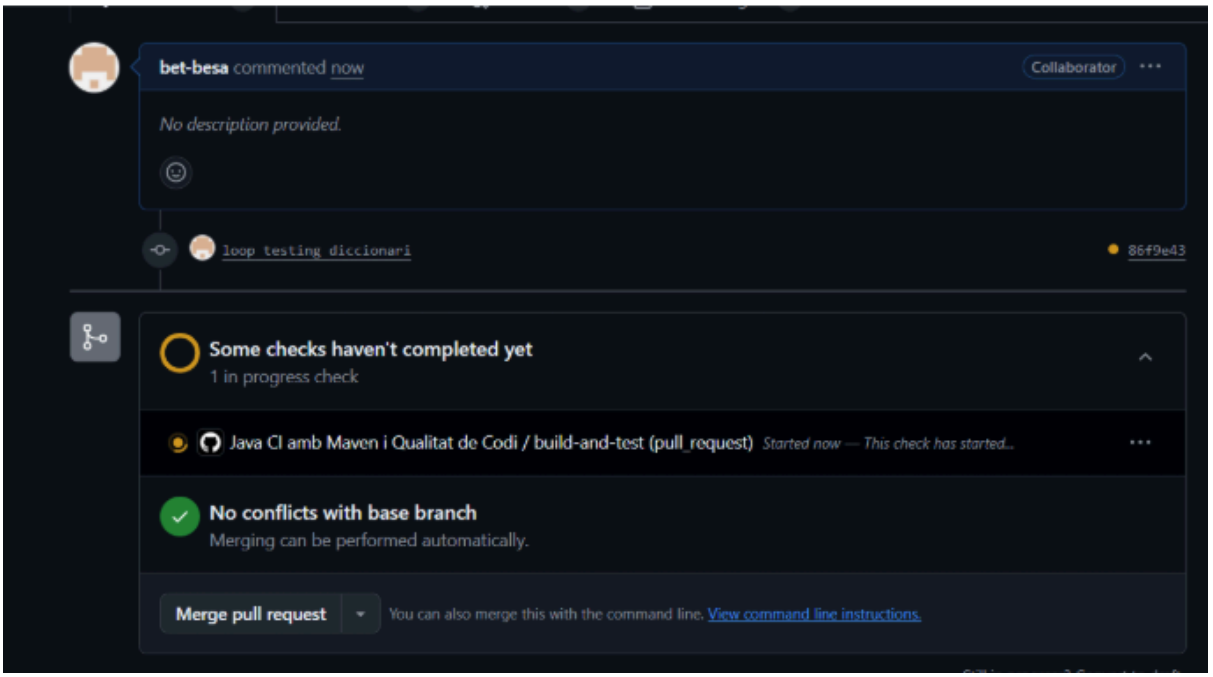
Precondicions: S'han implementat verificacions explícites dels paràmetres d'entrada abans d'executar la lògica dels mètodes. Això inclou comprovacions de null, longituds correctes i estats vàlids del sistema.

Postcondicions: Després d'operacions crítiques com afegirIntent(), s'han afegit assertions per verificar que l'estat del sistema sigui el correcte (nombre d'intents incrementat, intents restants decrementats, etc.).

Invariants: S'ha implementat un mètode invariant() a la classe Partida que verifica que l'estat de l'objecte sigui sempre consistent. Aquest mètode es crida al principi i final de cada mètode públic per assegurar la integritat de l'estat.

Integració Contínua i Coverage

El projecte incorpora GitHub Actions per executar automàticament tots els tests a cada commit i pull request. S'utilitza JaCoCo per generar informes de cobertura de codi, assegurant que es manté un mínim del 75% de coverage en tot moment. Aquesta pràctica de CI/CD garanteix que els canvis nous no introdueixin regressions i que la qualitat del codi es mantingui constant.



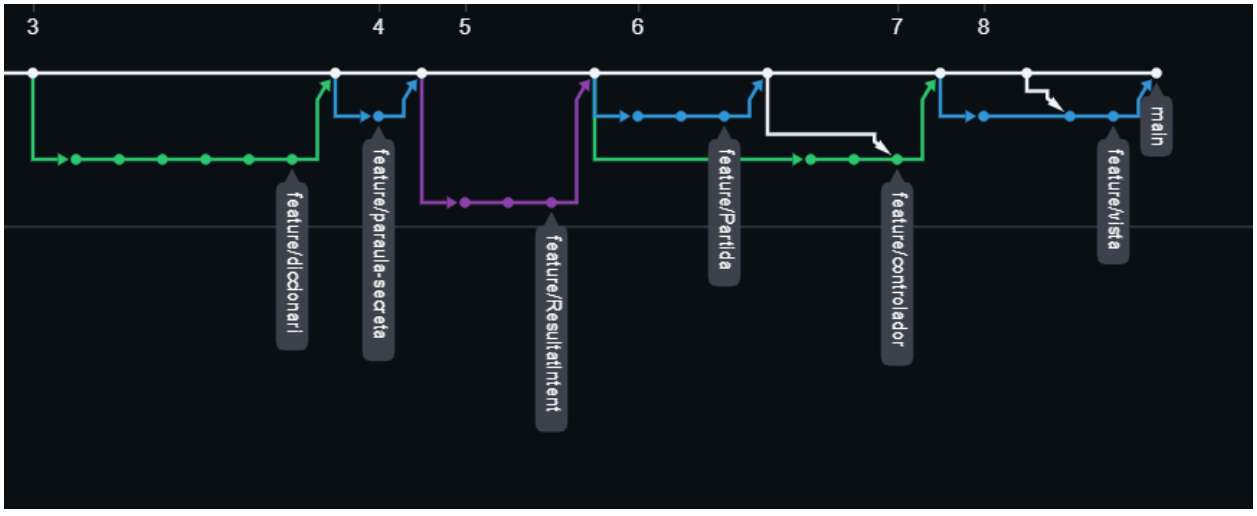
WordleTQS

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

Element	Coverage	Covered Instructions
WordleTQS	89,7 %	3.361
src/main/java	74,6 %	861
Model	82,1 %	691
Diccionari.java	67,0 %	187
Partida.java	79,3 %	218
ResultatIntent.java	98,3 %	113
EstatLletra.java	100,0 %	34
ParaulaSecreta.java	100,0 %	139
Controller	23,2 %	39
ControladorPartida.java	23,2 %	39
View	91,0 %	131
VistaWordle.java	91,0 %	131
src/test/java	96,4 %	2.500
Model	94,5 %	1.300
ParaulaSecretaTest.java	89,1 %	253
ResultatIntentTest.java	95,1 %	408
PartidaTest.java	96,7 %	557
DiccionariTest.java	95,3 %	82
View	96,0 %	311
VistaWordleTest.java	96,0 %	311
Controller	99,3 %	889
ControladorPartidaTest.java	99,3 %	889

Metodologia CI/CD

Hem utilitzat una estratègia de branques separades per cada funcionalitat:



- **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:
    - Crear branca des de main actualitzat: git checkout -b feature/nom.
    - Implementar funcionalitat aplicant TDD (mínim 2 commits).
    - Push a GitHub: git push origin feature/nom.
    - Crear Pull Request a la interfície de GitHub.
    - GitHub Actions executa tests automàticament.
    - Revisió del codi (si és necessari).
    - Si CI passa → Aprovar i fer merge a main.

Conclusions

Aquest projecte demostra una aplicació exhaustiva de la metodologia Test-Driven Development, utilitzant una combinació estratègica de tècniques de testing de caixa negra i caixa blanca. L'arquitectura MVC proporciona una separació clara de responsabilitats que facilita el testing independent de cada component.

L'ús de mockups amb Mockito ha permès testar components de manera aïllada, mentre que els tests parametritzats han millorat l'eficiència i la mantenibilitat dels tests. La implementació de precondicions, postcondicions i invariants segueix les millors pràctiques de programació defensiva i contractes.

El loop testing del bucle de càrrega del diccionari ha estat especialment valuós per assegurar la correcta gestió dels casos límit. La integració amb GitHub Actions i JaCoCo garanteix que la qualitat del codi es mantingui en tot moment del cicle de desenvolupament.