UAB

– TQS –

PARCHÍS

INFORME PRÀCTICA 1

Dimecres 10:30-12:30

Lucía Sánchez Guillén – 1633311
Youssef Cahouach Guella Ikhlaf – 1638618

ÍNDEX

SQUARE.JAVA

A aquesta classe se li ha aplicat Design By Contract a la totalitat dels seus mètodes.

Prova sobre que hem fet statement coverage sobre tota la classe de Square:

Element	Class, %	Method, %	Line, %	Branch, %
main.model.square	100% (1/1)	100% (10/10)	100% (39/39)	100% (28/28)
Square	100% (1/1)	100% (10/10)	100% (39/39)	100% (28/28)

- getPosition

Funcionalitat: S'encarrega de retornar la posició de la casella.

Localització: src/main/model/square/Square.java, Classe: Square, Mètode: getPosition()

Test: src/test/java/test/model/square/SquareTest.java, Classe: SquareTest, Mètode: getPosition()

Caixa negra:

➢ **Partició equivalent i valors límit i frontera:**

En aquest cas, ens hem basat en diferents posicions, en les quals, de la posició 0 fins a la 67 són vàlides, mentre que la resta són invàlides.

- Cas de posició 0, 67 els quals són valors frontera (vàlid).
- Cas de posició 34, posició vàlida entre valors frontera (vàlid).
- Cas de posició -1 i 1, valors límit a la frontera 0.
- Cas de posició 66 i 68, valors límit a la frontera 67.
- Cas de posició -10 i 70, els quals són invàlids.

Caixa Blanca:

➢ **Statement Coverage:**

Donat que és un getter, ja es fa un statement coverage donat que només es fa un return.

- landHere

Funcionalitat: S'encarrega de posicionar una peça en una casella (Square) concreta. Depenent del tipus de casella, ja sigui ShieldSquare (espai segur) com RegularSquare (casella normal), es tractarà de forma diferent en el cas en què la casella ja estigui ocupada per una altra peça o estigui buida.

Localització: src/main/model/square/Square.java, Classe: Square, Mètode: landHere(Piece piece)

Test: src/test/java/test/model/square/SquareTest.java, Classe: SquareTest, Mètode: landHere()

Caixa negra:

➢ **Partició equivalent:**

En aquest cas, ens hem basat en situacions que poden ocórrer dintre de la partida deforma que cobrim diferents inputs.

- Aterrar en una casella buida.
- Aterrar en una casella ocupada per una peça teva del mateix color.

- Aterrar en una casella ocupada per dues peces del teu equip (formen un bloqueig), de forma que no es pot aterrar.
- Aterrar en una casella ocupada per una peça de l'oponent, pot capturar.
- Aterrar en una casella ocupada per dues peces de l'oponent (formen un bloqueig), de forma que no es pot aterrar.

➢ ***Pairwise Testing*:** Mètode: landHere_pairwiseTesting()

Hem escollit aquesta funció per fer-li *Pairwise Testing*, donada la gran quantitat que situacions la qual podem aplicar-li a l'aterrament d'una peça en una casella qualsevol. De forma que hem generat la següent taula:

Tipus de casella (S)	Estat de la casella (O)
RS (Casella Normal)	Buida
RS (Casella Normal)	Pròpia
RS (Casella Normal)	Oponent
RS (Casella Normal)	Bloquejada (2 peces)
SS (Casella Segura)	Buida
SS (Casella Segura)	Pròpia
SS (Casella Segura)	Oponent
SS (Casella Segura)	Bloquejada (2 peces)

A partir de la taula, hem definit tots els casos de prova necessaris.

Caixa Blanca:

➢ **Statement Coverage:**

Donat que hem cobert gran part de la funció a l'hora de fer tests de caixa negra, només hem hagut de cobrir els casos següents:

- Quan la peça és nul·la.
- Aterrar en una casella segura.

➢ **Decision Coverage:**

Hem escollit aquesta funció per implementar decision coverage degut que inclou diferents elements de decisió com 'if'. Pel compliment d'aquesta prova hem comprovat cada decisió on sigui tant true com false. A continuació tenim un resum dels diferents casos de prova que hem implementat:

- Quan la casella esta ocupada -> if(isOccupied()) retorna true.
- Quan la casella no esta ocupada -> if(isOccupied()) retorna false.
- Quan es una casella segura -> if(isShieldSquare()) retorna true.
- Quan no es una casella segura -> if(isShieldSquare()) retorna false.

A continuació veiem com efectivament es verifiquen totes les decisions del mètode.

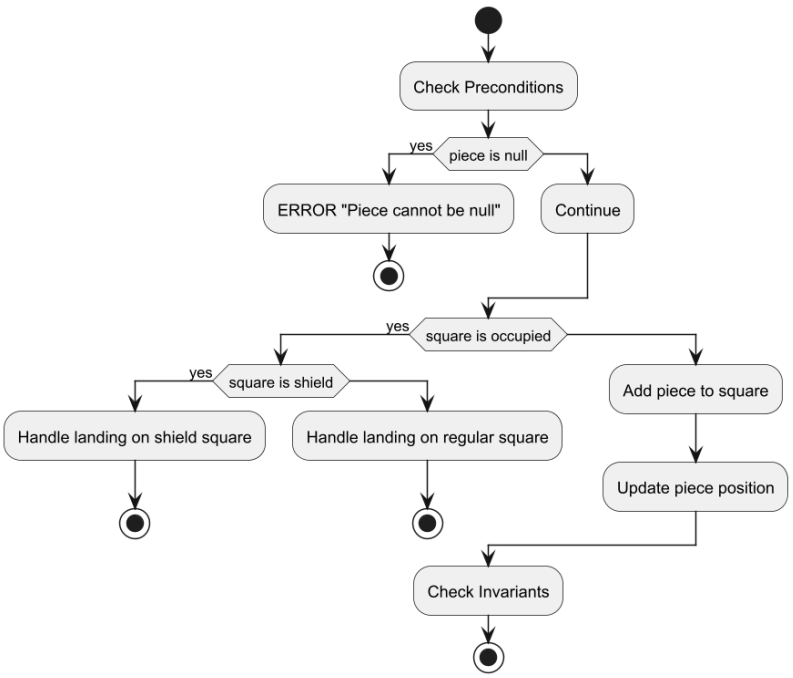
```java
public void landHere(Piece piece) {  YssfDevOps
  // Precondition: piece is not null
  assert piece != null : "Piece cannot be null";

  if (isOccupied()) {
    if (isShieldSquare()) {
      handleLandingOnShieldSquare(piece);
    } else {
      handleLandingOnRegularSquare(piece);
    }
  } else {
    // Square is empty
    pieces.add(piece);
    piece.setSquare(this);
  }

  // Invariant check
  invariant();
}
```

➢ **Path Coverage:**

En aquesta funció també hem implementat *path coverage* on validarem que es revisin tots els seus camins possibles. Els casos de prova són:

- Quan tenim una casella buida.
- Quan tenim una casella ja ocupada i és una casella segura (ShieldSquare).
- Quan tenim una casella ja ocupada i és una casella normal (RegularSquare).
- Quan no tenim peça.

Aquí veiem el diagrama d'activitats d'aquesta funció que s'ha seguit per implementar el *path coverage*:



Mock objects:

➢ **MockSquare:**

Hem utilitzat el MockSquare per poder simular els moviments correctes, a més de poder fer canvis en les caselles com posar-les a ocupades o modificar-les a caselles d'escut.

➢ **Implementació de mockito:**

Hem implementat un mockito on fem un mock de board i les peces d'un player per poder simular que les peces aterraven a una casella sense utilitzar les peces reals.

- **leave**

Funcionalitat: Funció que s'encarrega de treure una peça d'una casella concreta. Aquesta funció es truca, quan un jugador mou la seva peça. Simplement, treu de la llista de peces la peça concreta.

Localització: src/main/model/square/Square.java, Classe: Square, Mètode: leave(Piece piece)

Test: src/test/java/test/model/square/SquareTest.java, Classe: SquareTest, Mètode: leave()

Caixa negra:

➢ **Partició equivalent:**

En aquest cas, ens hem basat en situacions que poden ocórrer dintre de la partida deforma que cobrim diferents inputs.

- Sortir de la casella quan la peça està en la pròpia casella.
- Sortir de la casella quan la peça no està en aquella casella (impossible).

Caixa Blanca:

➢ **Statement Coverage:**

Donat que hem cobert gran part de la funció a l'hora de fer tests de caixa negra, només hem hagut de cobrir el cas següent:

- Quan la peça és nul·la.

- **isOccupied**

Funcionalitat: Funció que s'encarrega d'indicar si una casella està buida o no. En aquest cas, només comprova que la llista de peces d'una casella concreta estigui buida o no.

Localització: src/main/model/square/Square.java, Classe: Square, Mètode: isOccupied()

Test: src/test/java/test/model/square/SquareTest.java, Classe: SquareTest, Mètode: isOccupied()

Caixa negra:

➢ **Partició equivalent:**

En aquest cas, ens hem basat en situacions que poden ocórrer dintre de la partida deforma que cobrim diferents inputs.

- Cas en el que la casella estigui buida.

- Cas en el que la casella tingui una peça.
- Cas en el que la casella tingui una peça, se'n va, i tornem a comprovar si està buida.

Caixa Blanca:

➢ **Statement Coverage:**

Donat que hem cobert gran part de la funció a l'hora de fer tests de caixa negra, no queda més casos a cobrir.

- isBlocked

Funcionalitat: Funció que s'encarrega de comprovar si en una casella s'ha format un bloqueig per dues peces del mateix o diferent color. Això dependrà del tipus de casella, ja que cada tipus de casella té la seva implementació.

Localització: src/main/model/square/Square.java, Classe: Square, Mètode: isBlocked()

Test: src/test/java/test/model/square/SquareTest.java, Classe: SquareTest, Mètode: isBlocked()

Caixa negra:

➢ **Partició equivalent:**

Donat que aquesta funció no està implementada en la classe pare (Square), hem fet tests simples de partició equivalent.

- Cas en el que la casella estigui buida o té una peça.
- Cas en el que la casella té dues peces del mateix color (formen bloqueig).
- Cas en el que la casella té dues peces de diferent color (depenent del tipus de casella pot capturar o formar bloqueig).

Caixa Blanca:

➢ **Statement Coverage:**

El statement coverage es fa quan testegem les pròpies implementacions.

- isShieldSquare

Funcionalitat: Funció que s'encarrega d'indicar si una casella és segura o normal.

Localització: src/main/model/square/Square.java, Classe: Square, Mètode: isShieldSquare()

Test: src/test/java/test/model/square/SquareTest.java, Classe: SquareTest, Mètode: isShieldSquare()

Caixa negra i blanca:

➢ **Partició equivalent & Statement Coverage:**

Donat que aquesta funció no està implementada en la classe pare (Square), hem fet tests simples de partició equivalent i *statement coverage* junts, donat que només retorna true o false depenent del tipus de casella.

- Cas en el que la casella és normal, retornaria false.
- Cas en el que la casella és segura, retornaria true.

- **toString**

Funcionalitat: Funció que s'encarrega de transformar l'estat del tauler en un *String* per poder imprimir-ho en la consola, on s'indica en quin tipus de casella estem, la posició i quina peça és.

Localització: src/main/model/square/Square.java, Classe: Square, Mètode: toString()

Test: src/test/java/model/square/SquareTest.java, Classe: SquareTest, Mètode: testToString()

Caixa negra i blanca:

➢ **Partició equivalent & Statement Coverage:**

Provem per a diferents tipus de caselles en diferents posicions:

- Cas en el que no hi ha cap peça en la casella.
- Cas en el que la peça està situada en una casella segura.
- Cas en el que la peça està situada en una casella normal.

REGULARSQUARE.JAVA

A aquesta classe se li ha aplicat Design By Contract a la totalitat dels seus mètodes.

Prova sobre que hem fet statement coverage sobre tota la classe de RegularSquare:

Element	Class, %	Method, %	Line, %	Branch, %
main.model.square	100% (1/1)	100% (7/7)	100% (24/24)	100% (14/14)
RegularSquare	100% (1/1)	100% (7/7)	100% (24/24)	100% (14/14)

- **handleLandingOnRegularSquare**

Funcionalitat: Aquesta funció és trucada quan portem a terme un aterrament d'una peça en una casella concreta. Depenent del tipus de casella, es truca una funció o una altra. En aquest cas, tractem el cas en la qual una casella aterra en una casella normal. Donat que es una casella normal, si una peça d'un color concret cau en una casella on hi ha una altra peça de color diferent, aquesta mor i torna a casa.

Localització: src/main/model/square/RegularSquare.java, Classe: RegularSquare, Mètode: handleLandingOnRegularSquare(Piece piece)

Test: src/test/java/test/model/square/RegularSquareTest.java, Classe: RegularSquareTest, Mètode: handleLandingOnRegularSquare()

Caixa negra:

➢ **Partició equivalent:**

Per portar a terme la partició equivalent hem hagut de definir uns casos, que poden ocórrer durant una partida corrent, els quals són els següents:

- Aterrar una peça en una casella buida.
- Aterrar una peça en una casella que té una peça del mateix color. Això forma un bloqueig.
- Aterrar una peça en una casella que té una peça de diferent color. Això permet capturar una peça i enviar-la a casa.

Caixa Blanca:

- **Statement Coverage:**

Donat que hem de cobrir tots els casos hem hagut de crear mockups, donat que havien estats difícils de tractar. S'han pogut tractar els següents casos:

- La casella està totalment buida (*landHere* no ha controlat bé si estava buida) llavors comprovem un altre cop.
- Cas en el que afegim més de dues peces a una casella, de forma que això no està permès.
- Cas en el que truquem la funció abstracta handleLandingOnShieldSquare, cosa que no s'hauria de fer, si estem tractant amb caselles normals.

- **Decision Coverage:**

Hem escollit aquesta funció per implementar decision coverage degut que inclou diferents elements de decisió com 'if'. Pel compliment d'aquesta prova hem comprovat cada decisió on sigui tant true com false. A continuació tenim un resum dels diferents casos de prova que hem implementat:

- Quan no tenim peces → if(pieces.isEmpty()) retorna true.
- Quan tenim una peça → if(pieces.isEmpty()) retorna false i if(pieces.size() == 1) retorna true.
- Quan tenim més d'una peça → if(pieces.isEmpty()) retorna false i if(pieces.size() == 1) retorna false.
- Quan el color de la peça que ja es troba a la casella és del mateix color de la nova casella → if(occupant.getColor().equals(piece.getColor())) retorna true.
- Quan el color de la peça que ja es troba a la casella no és del mateix color de la nova casella → if(occupant.getColor().equals(piece.getColor())) retorna false.

A continuació veiem com efectivament es verifiquen totes les decisions del mètode.

```java
@Override  2 usages  1 override  YssfDevOps
protected void handleLandingOnRegularSquare(Piece piece) {
  // No pre-condition because is already in landHere

  if (pieces.isEmpty()) {
    // Square is empty
    pieces.add(piece);
  } else if (pieces.size() == 1) {
    Piece occupant = pieces.get(0);
    if (!occupant.getColor().equals(piece.getColor())) {
      // Capturing occurs
      occupant.sendHome();
      pieces.clear();
      pieces.add(piece);
    } else {
      // Form a blockage
      piece.setSquare(this);
      pieces.add(piece);
    }
  }

  // Invariant check
  invariant();
}
```

- **isBlocked**

Funcionalitat: Funció que s'encarrega de comprovar si en una casella s'ha format un bloqueig per dues peces del mateix color. Si no són del mateix color, una peça pot capturar l'altre.

Localització: src/main/model/square/RegularSquare.java, Classe: RegularSquare, Mètode: isBlocked(Piece piece)

Test: src/test/java/test/model/square/RegularSquareTest.java, Classe: RegularSquareTest, Mètode: isBlocked()

Caixa negra:

➢ **Partició equivalent:**

En aquest cas, tractarem els casos en el que es forma bloqueig o no en una casella normal, on es poden capturar les peces. S'han tractat els diferents casos:

- Cas en el que la casella estigui buida o té una peça.
- Cas en el que la casella té dues peces del mateix color (formen bloqueig).
- Cas en el que la casella té dues peces de diferent color, una peça pot capturar la peça que ja estigui en la casella.

Caixa Blanca:

➢ **Statement Coverage:**

Encara que hàgim cobert bastants casos en la caixa negra, vam haver de cobrir els casos següents per obtenir un statement coverage:

- Cas en el que la casella formi un bloqueig de color vermell i la peça de color vermell no pugui passar.
- Cas en el que la casella formi un bloqueig de color vermell i la peça de color vermell si pot passar.
- Cas en el que forcem dues peces de diferent color a estar dins d'una mateixa casella no formen un bloqueig.

➢ **Condition coverage:**

Com aquesta funció té decisions amb dues condicions o predicats, hem decidit aplicar condition coverage. Aquesta decisió està composta per les condicions:

pieces.size() == 2 && pieces.get(0).getColor().equals(pieces.get(1).getColor())

Els casos de prova que hem escollit per tal de complir amb aquest tipus de test son:

- Tenim una peça a la casella.
- Tenim dues peces a la casella del mateix color.
- Tenim dues peces a la casella de diferent color.
- Tenim una casella buida.

A continuació veiem com efectivament es verifiquen totes les condicions del mètode.

```
46      @Override  1 override  YssfDevOps
47      public boolean isBlocked(Piece piece) {
48        // Precondition
49        assert piece != null : "Piece must not be null";
50
51        // Invariant check
52        invariant();
53
54        if (pieces.size() == 2 && pieces.get(0).getColor().equals(pieces.get(1).getColor())) {
55          // Blockage by two pieces of the same color
56          return !pieces.get(0).getColor().equals(piece.getColor());
57        }
58
59        return false;
60      }
```

- **isShieldSquare**

Funcionalitat: Funció que indica si una casella és segura o no. Donat que aquesta classe és per tractar caselles normals, aquesta funció retorna fals.

Localització: src/main/model/square/RegularSquare.java, Classe: RegularSquare, Mètode: isShieldSquare()

Test: src/test/java/test/model/square/RegularSquareTest.java, Classe: RegularSquareTest, Mètode: isShieldSquare()

Caixa negra i blanca:

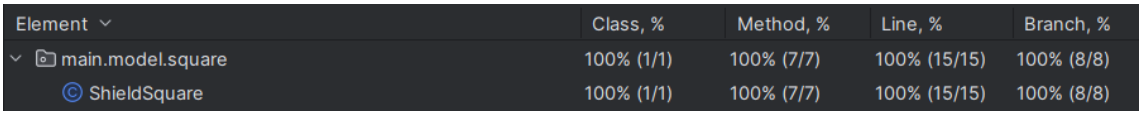
➢ **Partició equivalent & Statement Coverage:**

Donat que només hem de comprovar que realment retorn fals, hem portat a terme diferents tests amb diferents posicions de caselles per comprovar que realment retorna fals. En aquest cas, aquesta funció només retorna fals, de forma que el statement coverage està en part fet.

SHIELDSQUARE.JAVA

A aquesta classe se li ha aplicat Design By Contract a la totalitat dels seus mètodes.

Prova sobre que hem fet statement coverage sobre tota la classe de ShieldSquare:



Element	Class, %	Method, %	Line, %	Branch, %
main.model.square	100% (1/1)	100% (7/7)	100% (15/15)	100% (8/8)
ShieldSquare	100% (1/1)	100% (7/7)	100% (15/15)	100% (8/8)

- handleLandingOnShieldSquare

Funcionalitat: Aquesta funció és trucada quan portem a terme un aterrament d'una peça en una casella concreta. Depenent del tipus de casella, es truca una funció o una altra. En aquest cas, tractem el cas en la qual una casella aterra en una casella segura. Donat que és una casella segura, si una peça d'un color concret cau en una casella on hi ha una altra peça de color diferent, no podrà capturar-la de forma que han de conviure en la mateixa casella.

Localització: src/main/model/square/ShieldSquare.java, Classe: ShieldSquare, Mètode: handleLandingOnShieldSquare(Piece piece)

Test: src/test/java/test/model/square/ShieldSquareTest.java, Classe: ShieldSquareTest, Mètode: handleLandingOnShieldSquare()

Caixa negra:

➢ **Partició equivalent:**

Per portar a terme la partició equivalent hem hagut de definir uns casos, que poden ocórrer durant una partida corrent, els quals són els següents:

- Aterrar una peça en una casella buida.
- Aterrar una peça en una casella que té una peça.
- Aterrar una peça en una casella que té dues peces.
- Aterrar una peça en una casella que forma un bloqueig.

Caixa Blanca:

➢ **Statement Coverage:**

Donat que hem de cobrir tots els casos hem hagut de crear Mockup's, donat que havien estats difícils de tractar. S'han pogut tractar els següents casos:

- Intentar aterrar una peça que és nul·la.
- Afegir més de dues peces en una casella segura (això provoca error).
- Cas en el que obliguem que la casella segura no ho sigui.

- isBlocked

Funcionalitat: Funció que s'encarrega de comprovar si en una casella s'ha format un bloqueig per dues peces del mateix color. Si no són del mateix color, no passa res, donat que la casella és segura.

Localització: src/main/model/square/ShieldSquare.java, Classe: ShieldSquare, Mètode: isBlocked(Piece piece)

Test: src/test/java/test/model/square/ShieldSquareTest.java, Classe: ShieldSquareTest, Mètode: isBlocked()

Caixa negra:

➢ **Partició equivalent:**

En aquest cas, tractarem els casos en el que es forma bloqueig o no en una casella normal, on es poden capturar les peces. S'han tractat els diferents casos:

- Cas en el que la casella estigui buida, no forma bloqueig.
- Cas en el que la casella té una peça, no forma bloqueig.
- Cas en el que la casella té dues peces, formen bloqueig.

Caixa Blanca:

➢ **Statement Coverage:**

Encara que hàgim cobert bastants casos en la caixa negra, ens queda comprovar que la peça que enviem no sigui nul·la.

- Cas en el que la peça que intentem comprovar és nul·la.

- **isShieldSquare**

Funcionalitat: Funció que indica si una casella és segura o no. Donat que aquesta classe és per tractar caselles segures, aquesta funció retorna veritat.

Localització: src/main/model/square/ShieldSquare.java, Classe: ShieldSquare, Mètode: isShieldSquare()

Test: src/test/java/test/model/square/ShieldSquareTest.java, Classe: ShieldSquareTest, Mètode: isShieldSquare()

Caixa negra i blanca:

➢ **Partició equivalent & Statement Coverage:**

Donat que només hem de comprovar que realment retorna veritat, hem portat a terme diferents tests amb diferents posicions de caselles per comprovar que realment retorna veritat. En aquest cas, aquesta funció només retorna veritat, de forma que el statement coverage està en part fet.

FINALPATHSQUARE.JAVA

A aquesta classe se li ha aplicat Design By Contract a la totalitat dels seus mètodes.

Prova sobre que hem fet statement coverage sobre tota la classe de FinalPathSquare:

Element	Class, %	Method, %	Line, %	Branch, %
main.model.square	100% (1/1)	100% (9/9)	100% (21/21)	100% (12/12)
FinalPathSquare	100% (1/1)	100% (9/9)	100% (21/21)	100% (12/12)

- **getColor**

Funcionalitat: Aquesta funció ens retorna el color del recorregut final de cada jugador.

Localització: src/main/model/square/FinalPathSquare.java, Classe: FinalPathSquare, Mètode: getColor()

Test: src/test/java/test/model/square/FinalPathSquareTest.java, Classe: FinalPathSquareTest, Mètode: getColor()

Caixa negra i blanca:

➢ **Partició equivalent & Statement Coverage:**

Per portar aquest test, hem definit els colors groc, vermell, blau i verd com a colors valids, i la resta de colors com a erronis.

- Validem colors Groc, Vermell, Blau i Verd.
- Validem que qualsevol altre color, hauria de retornar un error.

- getIndex

Funcionalitat: Aquesta funció ens retorna la posició en la que una peça està en el recorregut final.

Localització: src/main/model/square/FinalPathSquare.java, Classe: FinalPathSquare, Mètode: getIndex()

Test: src/test/java/test/model/square/FinalPathSquareTest.java, Classe: FinalPathSquareTest, Mètode: getIndex()

Caixa negra:

➢ **Partició equivalent i valors límit i frontera:**

Definim partició equivalent on 0 i 7 són els valors frontera. Per tant, qualsevol valor entre 0 i 7 inclosos, és una posició vàlida, mentre que qualsevol posició fora d'aquell rang, és una posició invàlida. S'han comprovat els següents casos:

- Comprovem valors frontera 0 i 7 (vàlids).
- Comprovem valor 4 que està dintre del rang vàlid.
- Comprovem -1 i 1 els quals són valors límit a la frontera 0.
- Comprovem 6 i 8 els quals són valors límit a la frontera 7.
- Comprovem valors -10 i 10, els quals són invàlids i llunys de la frontera.

Caixa Blanca:

➢ **Statement Coverage:**

Donat que hem cobert molts casos en els tests de caixa negra, només ens queda fer el següent cas:

- Assignar una posició negativa a un camí final d'un jugador, de forma que al fer un getIndex() ens salti error.

- landHere

Funcionalitat: Aquesta funció és trucada des de la classe pare Square donat que en el camí final no tenim restriccions o normes com en una casella normal o segura.

Localització: src/main/model/square/Square.java, Classe: FinalPathSquare, Mètode: landHere(Piece piece)

Test: src/test/java/test/model/square/FinalPathSquareTest.java, Classe: FinalPathSquareTest, Mètode: handleLandingOnFinalPathSquare()

Caixa negra:

➢ **Partició equivalent:**

Per portar a terme la partició equivalent hem hagut de definir uns casos, que poden ocórrer durant una partida corrent, els quals són els següents:

- Aterrar una peça de color concret en un camí final del mateix color.
- Aterrar una peça de color concret en un camí final que és d'un altre color (no seria correcte).

Caixa Blanca:

➢ **Statement Coverage:**

Donat que hem tractat la majoria dels casos en la partició equivalent, en aquest cas, hem hagut de crear mockups per obligar a fer que la casella estigui ocupada i intentem aterrar poder cobrir els casos d'assercions:

- Intentar trucar el mètode *handleLandingOnShieldSquare,* hauria de donar error.
- Intentar trucar el mètode *handleLandingOnRegularSquare,* hauria de donar error.

- isBlocked

Funcionalitat: Aquesta funció indica que no es poden formar bloqueigs en un camí final, de forma que retorn fals.

Localització: src/main/model/square/FinalPathSquare.java, Classe: FinalPathSquare, Mètode: isBlocked(Piece piece)

Test: src/test/java/test/model/square/FinalPathSquareTest.java, Classe: FinalPathSquareTest, Mètode: isBlocked()

Caixa negra i blanca:

➢ **Partició equivalent & Statement Coverage:**

Donat que aquesta funció retorna directament fals, només comprovem que realment retorna fals. També comprovem que si li enviem una peça nul·la, salti error.

- isShieldSquare

Funcionalitat: Aquesta funció indica si aquesta casella és una casella segura de forma que només retornem fals.

Localització: src/main/model/square/FinalPathSquare.java, Classe: FinalPathSquare, Mètode: isShieldSquare()

Test: src/test/java/test/model/square/FinalPathSquareTest.java, Classe: FinalPathSquareTest, Mètode: isShieldSquare()

Caixa negra i blanca:

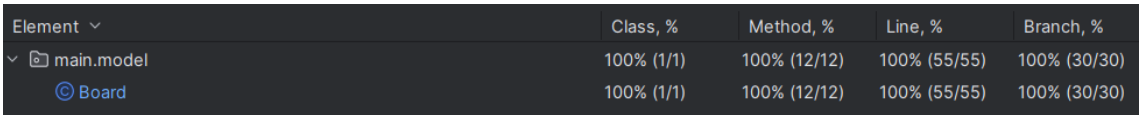
➢ **Partició equivalent & Statement Coverage:**

Donat que aquesta funció retorna directament fals, només comprovem que realment retorna fals, per a diferents posicions del camí final.

BOARD.JAVA

A aquesta classe se li ha aplicat Design By Contract a la totalitat dels seus mètodes.

Prova sobre que hem fet statement coverage sobre tota la classe de Board:



Element	Class, %	Method, %	Line, %	Branch, %
main.model	100% (1/1)	100% (12/12)	100% (55/55)	100% (30/30)
Board	100% (1/1)	100% (12/12)	100% (55/55)	100% (30/30)

- getGlobalSquare

Funcionalitat: Aquesta funció ens retorna la casella a partir d'una posició donada. Depenent de la posició donada ens pot retornar una casella segura o una casella normal.

Localització: src/main/model/Board.java, Classe: Board, Mètode: getGlobalSquare(int position)

Test: src/test/java/test/model/BoardTest.java, Classe: BoardTest, Mètode: getGlobalSquare()

Caixa negra i blanca:

➢ **Partició equivalent i valors límit i frontera & Statement Coverage:**

Donat que és un getter, el statement coverage està fet, de forma que només hem de definir una partició equivalent. En aquest cas, hem definit tota posició donada entre 0 i 67 inclosos, són posicions vàlides, mentre que qualsevol altra és errònia. Els casos són els següents:

- Cas de posició 0, 67 els quals són valors frontera (vàlids).
- Cas de posició 20, posició vàlida entre valors frontera.
- Cas de posició -1 i 1, valors límit a la frontera 0.
- Cas de posició 66 i 68, valors límit a la frontera 67.
- Cas de posició -10 i 100, els quals són invàlids.

Mock Object:

➢ **Implementació de mockito:** Mètode: getGlobalSquare_Mockito()

Hem implementat un mockito on fem un mock sobre el camí global de la partida getGlobalSquare_Mockito(). En aquest cas, simulem que Square no està implementat, i podem crear un camí global amb total normalitat i poder obtenir un tipus de Square, depenent de la posició. Comprovem tant posició vàlida com invàlida.

- getPlayerStartSquare

Funcionalitat: Aquesta funció ens retorna la posició inicial en el camí global de cada jugador depenent del seu color.

Localització: src/main/model/Board.java, Classe: Board, Mètode: getPlayerStartSquare(Color color)

Test: src/test/java/test/model/BoardTest.java, Classe: BoardTest, Mètode: getPlayerStartSquare()

Caixa negra:

➢ **Partició equivalent:**

Definim partició on els colors permesos són Vermell, Groc, Verd i Blau. Qualsevol altre color és invàlid.

- Cas en què la casella retornada pel jugador RED és 38.
- Cas en què la casella retornada pel jugador GREEN és 55.
- Cas en què la casella retornada pel jugador BLUE és 21.
- Cas en què la casella retornada pel jugador YELLOW és 4.
- Cas en què el color enviat és nul de forma que és erroni.

Caixa Blanca:

➢ **Statement Coverage:**

Donat que és un getter, el statement coverage, ja està definit.

Mock Object:

➢ **Implementació de mockito:** Mètode: getPlayerStartSquare_Mockito()

Hem implementat un mockito on fem un mock sobre el camí global de la partida i també sobre les posicions inicials de cada jugador getPlayerStartSquare_Mockito(). En aquest cas, simulem que ShieldSquare no està implementat, i podem crear un camí global i definir posicions inicials amb total normalitat i poder obtenir un tipus la posició concreta, depenent del color. Comprovem tant color vàlid com invàlid.

- setUpPlayerFinalPath

Funcionalitat: Aquesta funció defineix per a cada color, un camí final d'una mida determinada.

Localització: src/main/model/Board.java, Classe: Board, Mètode: setUpPlayerFinalPath(List<Color> colors, int NUM_SQUARES)

Test: src/test/java/test/model/BoardTest.java, Classe: BoardTest, Mètode: setUpPlayerFinalPaths_loopTesting()

Caixa Blanca:

➢ **Loop Testing Aniuat:**

Per fer loop testing aniuat, hem seguit la regla de començar amb un test simple pel loop més interior, fixant els altres loops al valor mínim. Testejar un loop més extern (com si fos un loop simple) mantenint el nombre d'iteracions dels loops interiors a valors habituals.

MAX_COLOURS = 4 (for n) MAX_SQUARES = 8 (for m) (n,m) = (1,0) (1,1) (1,2) (1,m<MAX_COLOURS-1) (1,MAX_COLOURS-1) (1,MAX_COLOURS) (0,M<MAX_COLOURS) (1,M<MAX_COLOURS) (2,m<MAX_COLOURS) (n<MAX_SQUARES-1,m<MAX_COLOURS) (MAX_SQUARES-1,m<MAX_COLOURS) (MAX_SQUARES,m<MAX_COLOURS)

```
87      // For testing purposes
88  @   public void setUpPlayerFinalPaths_Custom(List<Color> colors, int NUM_SQUARES) {  1 usage  new *
89          for (Color color : colors) {
90              List<FinalPathSquare> finalPath = new ArrayList<>(NUM_SQUARES);
91              for (int i = 0; i < NUM_SQUARES; i++) {
92                  finalPath.add(new FinalPathSquare(i, color));
93              }
94              playerFinalPaths.put(color, finalPath);
95          }
```

En aquesta funcionalitat s'ha hagut de crear una nova funció anomenada setUpPlayerFinalPaths_Custom, la qual ens permet definir el nombre de colors i el nombre de

caselles que pot tenir el camí final. De forma que per aquesta funció només s'ha creat per portar a terme loop testing. La funció setUpPlayerFinalPaths la qual és privada té la implementació original, la qual es trucada per construir el taulell (Board).

- **getNextSquare**

Funcionalitat: Aquesta funció ens retorna la següent casella a partir de la posició de la peça que volem moure. Aquesta funció no només s'encarrega de donar-nos la següent casella sinó que també controla la transició entre el camí global i el camí final de cada jugador. A partir d'una fórmula que hem definit, podem saber si una peça de cert color ha arribat a la seva posició final, ja que cada jugador té una posició final definida.

Localització: src/main/model/Board.java, Classe: Board, Mètode: getNextSquare(Square currentSquare, Piece piece)

Test: src/test/java/test/model/BoardTest.java, Classe: BoardTest, Mètode: getNextSquare()

Caixa negra:

➢ **Partició equivalent:**

Definim partició basant-nos en situacions que poden ocórrer durant una partida. Els casos són els següents:

- Obtenir una casella que estigui en el camí global, passant-li una casella del camí global.
- Obtenir una casella que estigui en el camí final, passant-li una casella del camí global, és a dir, produir una transició.
- Obtenir una casella que estigui en el camí global, passant-li una casella del camí final.
- Arribar a l'última casella del camí final, posar peça com a acabada i square com a nul.
- Passar-li una casella nul·la, hauria de donar error.
- Passar-li una peça nul·la, hauria de donar error.

Caixa Blanca:

➢ **Statement Coverage:**

Donat que hem definit tots aquells casos en la caixa negra, ja hem cobert tots els casos de forma que totes les línies s'executen.

Mock Object:

➢ **Implementació de mockito:** Mètode: getNextSquare_Mockito()

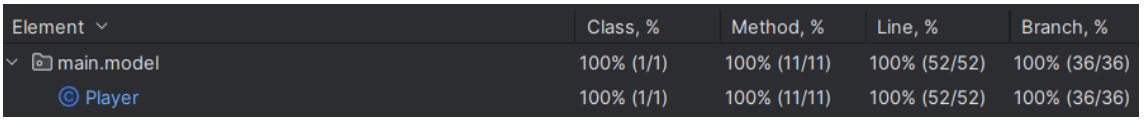
Hem implementat un mockito on fem un mock sobre el camí global de la partida, sobre les posicions inicials i camí finals de cada jugador. En aquest cas, simulem que RegularSquare, ShieldSquare i FinalPathSquare no estan implementats, de forma que podem simular els moviments de les peces entre diferents tipus de caselles. En aquest cas, hem fet els següents tests:

- Obtenir una casella que estigui en el camí global, passant-li una casella del camí global.
- Obtenir una casella que estigui en el camí global, passant-li una casella del camí final.
- Arribar a l'última casella del camí final, posar peça com a acabada i square com a nul.
- Passar-li una casella nul·la, hauria de donar error.
- Passar-li una peça nul·la, hauria de donar error.

PLAYER.JAVA

A aquesta classe se li ha aplicat Design By Contract a la totalitat dels seus mètodes.

Prova sobre que hem fet statement coverage sobre tota la classe de Player:



Element	Class, %	Method, %	Line, %	Branch, %
main.model	100% (1/1)	100% (11/11)	100% (52/52)	100% (36/36)
Player	100% (1/1)	100% (11/11)	100% (52/52)	100% (36/36)

- **movePiece**

Funcionalitat: Aquesta funció s'encarrega de moure una peça donada a partir del nombre de moviments donats. Fa una iteració per cada moviment, on demana constantment la següent casella amb la funció *getNextSquare* i després fa les comprovacions pertinents per no incomplir cap regla del joc.

Localització: src/main/model/Player.java, Classe: Player, Mètode: movePiece(Piece piece, int moves, Board board)

Test: src/test/java/test/model/PlayerTest.java, Classe: PlayerTest, Mètode: movePiece()

Caixa negra:

➢ **Partició equivalent i valors límit i frontera:**

Per fer la partició equivalent, ens hem basat en la quantitat de moviments que pots fer en un peça. Donat que un dau pot donar entre 1 i 6 moviments, hem definit la partició equivalent a partir d'aquests valors. Tot valor entre 1 i 6 inclosos, són vàlids mentre que la resta són invàlids.

- Cas de moviment 1 i 6 els quals són valors frontera (vàlids).
- Cas de moviment 4, posició vàlida entre valors frontera.
- Cas de moviment 0 i 2, valors límit a la frontera 1.
- Cas de moviment 5 i 7, valors límit a la frontera 6.
- Cas de moviment -10 i 10, els quals són invàlids.

Caixa blanca:

➢ **Statement Coverage:**

Per cobrir totes les sentències hem seguit la mateixa lògica que fèiem a l'hora de fer proves de caixa negra. Aquests casos de prova s'han definit a partir de casos que es poden produir durant una partida de parxís:

- Moure una peça amb normalitat sense cap mena de bloqueig pel mig.
- Moure una peça que acabi el joc.
- Intentar moure una peça que està bloquejada, ja que hi ha un bloqueig davant.
- Intentar moure una peça que està a casa.
- Intentar moure una peça que ja ha acabat.

➢ **Decision Coverage:**

Hem escollit aquesta funció per implementar decision coverage degut que inclou diferents elements de decisió com 'if'. Pel compliment d'aquesta prova hem comprovat cada decisió on sigui tant true com false. A continuació tenim un resum dels diferents casos de prova que hem implementat:

- Quan no tenim nextSquare → if(nextSquare == null) retorna true.

- Quan si tenim nextSquare → if(nextSquare == null) retorna false.
- Quan nextSquare esta bloquejat → if(nextSquare.isBlocked(piece)) retorna true.
- Quan nextSquare no esta bloquejat → if(nextSquare.isBlocked(piece)) retorna false.
- Quan → if(lastAccessibleSquare != currentSquare) retorna true.
- Quan → if(lastAccessibleSquare != currentSquare) retorna false.

A continuació veiem com efectivament es verifiquen totes les decisions del mètode.

```
48      // Move a piece by a certain number of moves
49  @   public void movePiece(Piece piece, int moves, Board board) {  26 usages  luciasg14 +1
50          // Preconditions
51          assert piece != null : "Piece cannot be null";
52          assert moves > 0 : "Moves must be positive";
53          assert moves <= 6 : "Moves must be less than 6";
54          assert board != null : "Board cannot be null";
55
56          Square currentSquare = piece.getSquare();
57          Square lastAccessibleSquare = currentSquare;
58
59          for (int i = 0; i < moves; i++) {
60              Square nextSquare = board.getNextSquare(lastAccessibleSquare, piece);
61
62              if (nextSquare == null) {
63                  // Reached the end of final path
64                  piece.setHasFinished(true);
65                  currentSquare.leave(piece);
66                  piece.setSquare(null);
67                  return;
68              }
69
70              if (nextSquare.isBlocked(piece)) {
71                  // Cannot pass through blockage, stop to square behind
72                  break;
73              }
```

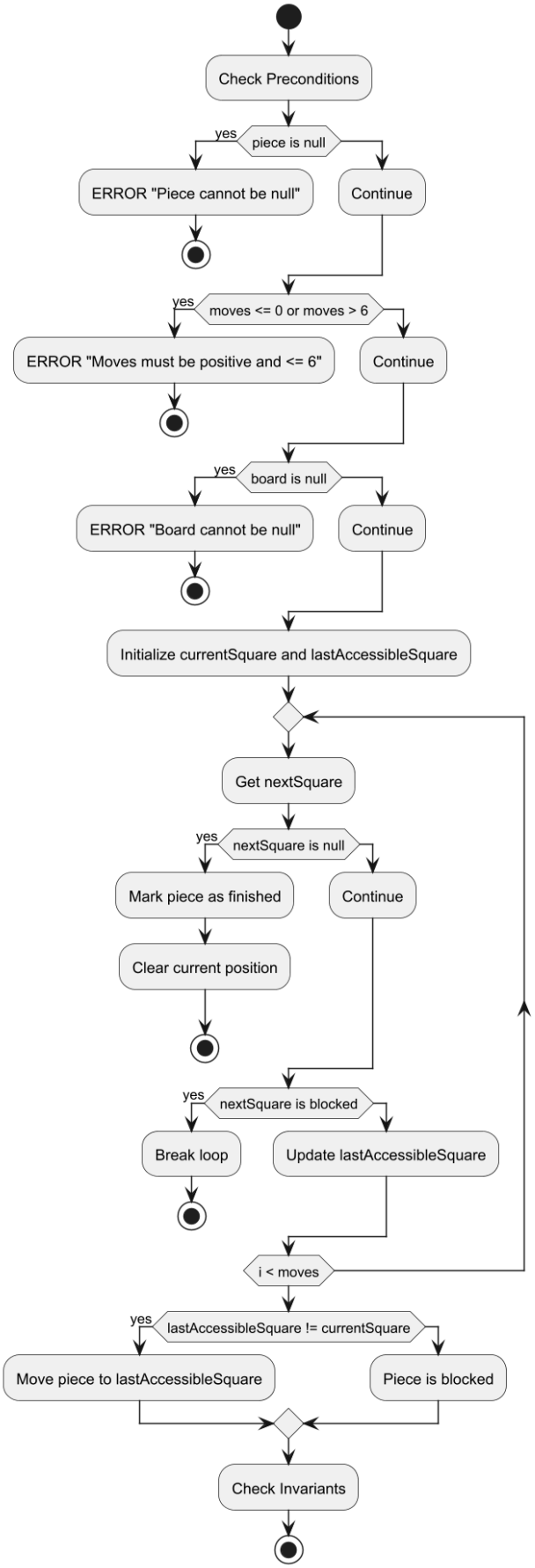
```
74
75          // Update the last accessible square
76          lastAccessibleSquare = nextSquare;
77      }
78
79      if (lastAccessibleSquare != currentSquare) {
80          // Move the piece to the last accessible square
81          currentSquare.leave(piece);
82          lastAccessibleSquare.landHere(piece);
83          piece.setSquare(lastAccessibleSquare);
84      } else {
85          // Could not move due to blockage; stay on the same square
86          System.out.println(piece.getColor() + " Piece " + piece.getId() + " is blocked.");
87      }
88
89      // Invariant
90      invariant();
91  }
```

➢ **Path Coverage:**

En aquesta funció també hem implementat path coverage on validarem que es revisin tots els seus camins possibles. Els casos de prova són:

- Moviment normal.
- Quan hem arribat al final del tauler.
- Quan nextSquare està bloquejat.
- Quan una peça no es pot moure (bloquejada).
- Moviments invàlids.
- Sense peça o sense tauler.

Aquí veiem el diagrama d'activitats d'aquesta funció que s'ha seguit per implementar el path coverage:


Check Preconditions
yes
piece is null
ERROR "Piece cannot be null"
Continue
yes
moves <= 0 or moves > 6
ERROR "Moves must be positive and <= 6"
Continue
yes
board is null
ERROR "Board cannot be null"
Continue
Initialize currentSquare and lastAccessibleSquare
Get nextSquare
yes
nextSquare is null
Mark piece as finished
Continue
Clear current position
yes
nextSquare is blocked
Break loop
Update lastAccessibleSquare
i < moves
yes
lastAccessibleSquare != currentSquare
Move piece to lastAccessibleSquare
Piece is blocked
Check Invariants

➢ **Loop testing Simple:** Mètode: movePiece_loopTesting()

Per fer loop testing simple hem seguit la regla d'evitar loop, una passada pel loop, dues passades pel loop, m passades pel loop m<n, (n-1) i ***n*** pasades pel loop (n és el nombre màxim de passades). Llavors hem definit el següent conjunt de moviments:

MAX_MOVES = 6
Simple loop: 0, 1, 2, n < MAX_MOVES - 1, MAX_MOVES - 1, MAX_MOVES

```
9   public class Player {  79 usages  YssfDevOps +1 *
48    // Move a piece by a certain number of moves
49    public void movePiece(Piece piece, int moves, Board board) {  25 usages  luciasg14 +1 *
50      // Preconditions
51      assert moves > 0 : "Moves must be positive";
52      assert moves <= 6 : "Moves must be less than 6";
53
54      Square currentSquare = piece.getSquare();
55      Square lastAccessibleSquare = currentSquare;
56
57      for (int i = 0; i < moves; i++) {
58        Square nextSquare = board.getNextSquare(lastAccessibleSquare, piece);
59
60        if (nextSquare == null) { // Reached the end of final path
61          piece.setHasFinished(true);
62          currentSquare.leave(piece);
63          piece.setSquare(null);
64          return;
65        }
66
67        // Cannot pass through blockage, stop to square behind
68        if (nextSquare.isBlocked(piece)) {
69          break;
70        }
71
72        // Update the last accessible square
73        lastAccessibleSquare = nextSquare;
74      }
75
76      if (lastAccessibleSquare != currentSquare) {
77        // Move the piece to the last accessible square
78        currentSquare.leave(piece);
79        lastAccessibleSquare.landHere(piece);
80        piece.setSquare(lastAccessibleSquare);
81      } else {
82        // Could not move due to blockage; stay on the same square
83        System.out.println(piece.getColor() + " Piece " + piece.getId() + " is blocked.");
84      }
85
86      // Invariant
87      invariant();
88    }
```

Mock objects:

➢ **MockSquare i MockBoard:**

Hem utilitzat el MockSquare i MockBoard per poder simular els moviments correctes. MockSquare l'hem utilitzat per crear diferents squares on representaven caselles inicials, mitjanes i finals on cadascuna li atorgàvem diferents propietats. MockBoard l'hem utlitzat per poder fer els tests correctament sense tenir encara implementat del tot el board real.

➢ **Implementació de mockito:**

Hem implementat un mockito on fem un mock de board i les peces d'un player per poder fer modificacions als objectes directament. Per exemple modificavem el getNextSquare de la classe Board perquè retornés una casella concreta, i així, poder verificar el moviment de les peces del jugador a diferents caselles.

- ## enterPieceIntoGame

Funcionalitat: Aquesta funció s'utilitza per posar una peça en joc en la seva primera casella corresponent. S'utilitza quan el jugador ha tret un 5 amb el dau. En aquest cas, s'agafa la primera peça que estigui a casa i després es truca la funció *enterGame* la qual s'encarrega de tot el procés d'addició a la partida. Si no tens cap peça a casa, aquesta funció no es truca.

Localització: src/main/model/Player.java, Classe: Player, Mètode: enterPieceIntoGame()

Test: src/test/java/test/model/PlayerTest.java, Classe: PlayerTest, Mètode: enterPieceIntoGame()

Caixa negra i blanca:

➢ **Partició equivalent & Statement coverage:**

Per fer els casos de prova ens hem basat en situacions que poden ocórrer durant el transcurs d'una partida. S'han definit els següents casos:

- Treure una peça a jugar quan la casella inicial està disponible.
- Treure una peça a jugar quan la casella inicial està plena.
- Intentar treure una peça, quan ja tens totes les peces en joc.

- ## isWinner

Funcionalitat: Funció que s'encarrega d'analitzar l'estat de cada peça d'un jugador concret, comprovant si cada peça ha acabat. Si totes les peces han acabat tot el recorregut, es considera que aquell jugador és el guanyador de la partida.

Localització: src/main/model/Player.java, Classe: Player, Mètode: isWinner()

Test: src/test/java/test/model/PlayerTest.java, Classe: PlayerTest, Mètode: isWinner()

Caixa negra i blanca:

➢ **Partició equivalent & Statement Coverage:**

Donat que aquesta funció només fa un recorregut sobre les peces del jugador comprovant l'estat de cadascuna hem definit uns casos, els quals cobreixen cada possibilitat i a més, executen cada línia assegurant statement coverage. Els casos són els següents:

- Cas en el que el jugador encara té peces en partida (no és guanyador).
- Cas en el que totes les peces del jugador han finalitzat, i es proclama guanyador de la partida.

- **hasPiecesAtHome**

Funcionalitat: Funció que s'encarrega d'analitzar l'estat de cada peça d'un jugador concret, comprovant si el jugador té alguna peça a casa. Aquesta funció s'utilitza durant el bucle principal del joc, on si trèiem un 5 i totes les peces ja estan jugant, no fa falta trucar la funció per treure una peça de casa i podem moure directament una peça determinada.

Localització: src/main/model/Player.java, Classe: Player, Mètode: hasPiecesAtHome()

Test: src/test/java/test/model/PlayerTest.java, Classe: PlayerTest, Mètode: hasPiecesAtHome()

Caixa negra i blanca:

➢ **Partició equivalent & Statement Coverage:**

Donat que aquesta funció només fa un recorregut sobre les peces del jugador comprovant l'estat de cadascuna hem definit uns casos, els quals cobreixen cada possibilitat i a més, executen cada línia assegurant statement coverage. Els casos són els següents:

- Cas en el que el jugador no té cap peça a casa.
- Cas en el que el jugador té peces a casa.

➢ **Condition coverage:**

Hem decidit aplicar condition coverage per comprovar tots els casos possibles de les condicions. En aquest cas només hi tenim una condició:

piece.isAtHome()

Els casos de prova que hem escollit per tal de complir amb aquest tipus de test són:

- Un jugador té una peça a casa (totes).
- Un jugador no té cap peça a casa.

A continuació veiem com efectivament es verifiquen totes les condicions del mètode.

```
117     // Check if the player has any pieces at home
118     public boolean hasPiecesAtHome() {  7 usages   YssfDevOps
119         for (Piece piece : pieces) {
120             if (piece.isAtHome()) {
121                 return true;
122             }
123         }
124         return false;
125     }
```

- **hasPiecesOnBoard**

Funcionalitat: Funció que s'encarrega d'analitzar l'estat de cada peça d'un jugador concret, comprovant si el jugador té alguna peça en joc. Aquesta funció s'utilitza durant el bucle principal del joc, on depenent de la sortida d'aquesta funció, el joc ens mostra la possibilitat de moure una peça en cas d'una sortida verdadera o passem al següent jugador en cas d'una sortida falsa.

Localització: src/main/model/Player.java, Classe: Player, Mètode: hasPiecesOnBoard()

Test: src/test/java/test/model/PlayerTest.java, Classe: PlayerTest, Mètode: hasPiecesOnBoard()

Caixa negra i blanca:

➢ **Partició equivalent & Statement Coverage:**

Donat que aquesta funció només fa un recorregut sobre les peces del jugador comprovant l'estat de cadascuna hem definit uns casos, els quals cobreixen cada possibilitat i a més, executen cada línia assegurant statement coverage. Els casos són els següents:

- Cas en el que el jugador té totes les peces a casa, de forma que no pot portar a terme cap moviment sobre cap peça.
- Cas en el que el jugador té almenys una peça en partida.
- Cas en el que totes les peces del jugador han finalitzat el seu recorregut de forma que el jugador ja no té peces per jugar.

➢ **Condition coverage:**

Com aquesta funció té decisions amb dues condicions o predicats, hem decidit aplicar condition coverage. Aquesta decisió està composta per les condicions:

!piece.isAtHome() && !piece.hasFinished()

Els casos de prova que hem escollit per tal de complir amb aquest tipus de test son:

- Un jugador té una peça al tauler.
- Un jugador té totes les peces a casa.
- Un jugador té totes les peces a la casella final.
- Un jugador té almenys una peça al taulell.
- Un jugador no té peces.

A continuació veiem com efectivament es verifiquen totes les condicions del mètode.

```
128     // Check if the player has any pieces on the board
129     public boolean hasPiecesOnBoard() {  10 usages  YssfDevOps
130         for (Piece piece : pieces) {
131             if (!piece.isAtHome() && !piece.hasFinished()) {
132                 return true;
133             }
134         }
135         return false;
136     }
```

PIECE.JAVA

A aquesta classe se li ha aplicat Design By Contract a la totalitat dels seus mètodes.

Prova sobre que hem fet statement coverage sobre tota la classe de Piece:

Element	Class, %	Method, %	Line, %	Branch, %
main.model	100% (1/1)	100% (13/13)	100% (41/41)	100% (14/14)
Piece	100% (1/1)	100% (13/13)	100% (41/41)	100% (14/14)

- **getId**

Funcionalitat: Cada peça que té un jugador se li associa una identificació incrementativa, de forma que cada peça té un identificador únic per a tota la partida. Aquesta funció retorna la identificació d'una peça donada.

Localització: src/main/model/Piece.java, Classe: Piece, Mètode: getId()

Test: src/test/java/test/model/PieceTest.java, Classe: PieceTest, Mètode: getId()

Caixa negra i blanca:

➢ **Partició equivalent & Statement coverage:**

Hem definit els casos de prova basant-nos en situacions reals del joc. En aquest cas, comprovem que cada vegada que creem un jugador se li assigna a cada peça un identificador únic. Pel que fa al statement coverage, donat que només retorna l'identificador, en aquest cas ja es compleix.

- **isAtHome**

Funcionalitat: Funció que retorna si una peça està a casa o no.

Localització: src/main/model/Piece.java, Classe: Piece, Mètode: isAtHome()

Test: src/test/java/test/model/PieceTest.java, Classe: PieceTest, Mètode: isAtHome()

Caixa negra i blanca:

➢ **Partició equivalent & Statement coverage:**

Pel que fa al statement coverage, donat que només retorna la variable *atHome*, en aquest cas ja es compleix. Hem definit els casos de prova basant-nos en situacions reals del joc. Els casos són els següents:

- Quan iniciem una partida, totes les peces haurien d'estar a casa.
- Quan posem una peça en joc, aquesta peça indica en les seves propietats que ja no està a casa.
- Enviar a casa una peça que estava en joc, ens hauria d'indicar per les seves propietats que ara està a casa.

- **sendHome**

Funcionalitat: Funció que s'encarrega d'enviar una peça concreta a casa. Aquesta funció és trucada quan una peça captura a una altra.

Localització: src/main/model/Piece.java, Classe: Piece, Mètode: sendHome()

Test: src/test/java/test/model/PieceTest.java, Classe: PieceTest, Mètode: sendHome()

Caixa negra i blanca:

➢ **Partició equivalent & Statement Coverage:**

Donat que és una funció simple, hem cobert tots els casos de prova, definint uns casos que poden ocórrer durant una partida real. Els casos són els següents:

- Enviar a casa una peça que està en joc.
- Intentar enviar a casa una peça que ja està a casa.

Mock Object:

- ➢ **Implementació de mockito:** Mètode: sendHome_Mockito()

Hem implementat un mockito on fem un mock sobre la classe Square el qual ens permet simular l'aterrament d'una peça en una casella qualsevol, i després utilitzar la funció de sendHome per comprovar que realment s'ha enviat a casa.

- enterGame

Funcionalitat: Funció que s'encarrega de posicionar una peça concreta que està a casa en la casella inicial del jugador. En aquest cas, es comprova si hi ha espai en la casella inicial per col·locar-la. Aquesta funció és trucada, quan un jugador ha tret un 5 i ha decidit treure una de les peces que té a casa.

Localització: src/main/model/Piece.java, Classe: Piece, Mètode: enterGame(Board board)

Test: src/test/java/test/model/PieceTest.java, Classe: PieceTest, Mètode: enterGame()

Caixa negra i blanca:

- ➢ **Partició equivalent & Statement Coverage:**

A l'hora de definir els casos, ens hem basat en situacions que poden ocòrrer en una partida real. Aquests casos de prova, ens han cobert totes les línies de forma que es compleix l'statement coverage. Els casos són els següents:

- Posar una peça en joc quan la casella està lliure.
- Intentar posar una peça en joc, quan la casella està plena. En aquest cas, retornem un missatge dient que la posició inicial està plena.

Mock Object:

- ➢ **Implementació de mockito:** Mètode: enterGame_Mockito()

Hem implementat un mockito on fem un mock sobre les classes Board i ShieldSquare. Al fer mocks sobre aquestes classes, podem simular el funcionament de la funció enterGame on portem a terme la col·locació d'una peça en joc i després comprovem que realment està en la casella determinada.

- toString

Funcionalitat: Funció que s'encarrega de transformar l'estat de les peces del jugador en un *String* per poder imprimir-ho en la consola, on s'indica la posició de la peça, si està a casa o ha acabat o en quina part del camí global o final està.

Localització: src/main/model/Piece.java, Classe: Piece, Mètode: toString()

Test: src/test/java/test/model/PieceTest.java, Classe: PieceTest, Mètode: testToString()

Caixa negra i blanca:

- ➢ **Partició equivalent & Statement Coverage:**

A l'hora de definir els casos, ens hem basat en situacions que poden ocórrer en una partida real. Aquests casos de prova, ens han cobert totes les línies de forma que es compleix l'statement coverage. Els casos són els següents:

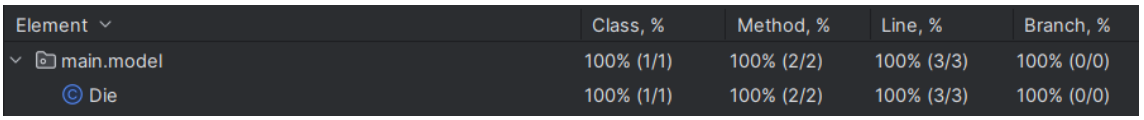
- La peça està a casa.
- La peça ha acabat el seu recorregut.
- La peça està situada en una casella del camí global.

- La peça està situada en una casella del camí final.

DIE.JAVA

A aquesta classe se li ha aplicat Design By Contract a la totalitat dels seus mètodes.

Prova sobre que hem fet statement coverage sobre tota la classe de Die:



Element	Class, %	Method, %	Line, %	Branch, %
main.model	100% (1/1)	100% (2/2)	100% (3/3)	100% (0/0)
Die	100% (1/1)	100% (2/2)	100% (3/3)	100% (0/0)

- roll

Funcionalitat: S'encarrega de generar un nombre aleatori entre 1 i 6 inclosos. Aquesta funció és trucada en cada torn de la partida. El nombre aleatori el generem a partir d'una fórmula que assegura que el número generat estigui dintre del rang.

Localització: src/main/model/Die.java, Classe: Die, Mètode: roll()

Test: src/test/java/test/model/DieTest.java, Classe: DieTest, Mètode: roll()

Caixa negra i blanca:

➢ **Partició equivalent & Statement coverage:**

Hem portat a terme un test el qual llença el dau 1000 vegades, i en tots els casos, ens ha de donar un resultat que estigui entre 1 i 6 inclosos.

GAMECONTROLLER.JAVA

A aquesta classe se li ha aplicat Design By Contract a la totalitat dels seus mètodes.

- GameController

Funcionalitat: Constructor el qual s'encarrega d'inicialitzar totes les classes del model.

Localització: src/main/controller/GameController.java, Classe: GameController, Mètode: GameController()

Test: src/test/java/test/controller/GameControllerTest.java, Classe: GameControllerTest, Mètode: testGameConstructor()

Tests:

➢ **Test Simple:**

Hem portat a terme un test el qual llença el dau 1000 vegades, i en tots els casos, ens ha de donar un resultat que estigui entre 1 i 6 inclosos.

- **initializePlayers**

Funcionalitat: S'encarrega d'inicialitzar els jugadors amb les seves peces a partir del nombre de jugadors que es dona mitjançant un *user input*. Indiquem el nombre de jugadors a través de la consola, i després definim el nom de cada jugador.

Localització: src/main/controller/GameController.java, Classe: GameController, Mètode: initializePlayers(int numPlayers)

Test: src/test/java/test/controller/GameControllerTest.java, Classe: GameControllerTest, Mètode: initializePlayers()

Tests:

➢ **Test Simple:**

Hem portat a terme uns tests basant-nos en situacions reals que poden ocórrer quan un jugador interacciona amb una consola. Els casos són els següents:

- Cas en el que el nombre de jugadors és major a 0 i menor a 4, i els noms de cada jugador no són buits.
- Cas en el que el nombre de jugadors superi el permès que és 4.

Caixa blanca:

➢ **Loop Testing Simple:** Mètode: initializePlayers_loopTesting()

Per fer loop testing simple hem seguit la regla d'evitar loop, una passada pel loop, dues passades pel loop, m passades pel loop m<n, (n-1) i ***n*** pasades pel loop (n és el nombre màxim de passades). Llavors hem definit el següent conjunt de moviments:

MAX_PLAYERS = 4 Simple loop: 0, 1, 2, MAX_PLAYERS - 1, MAX_PLAYERS

```
43    public void initializePlayers(int numPlayers, List<String> playerNames) {  9 usages  YssfDevOps
44        assert numPlayers > 0 && numPlayers <= Color.values().length : "Invalid number of players";
45
46        Color[] colors = Color.values();
47        for (int i = 0; i < numPlayers; i++) {
48            String playerName = playerNames.get(i);
49            Player player = new Player(playerName, colors[i], board);
50            players.add(player);
51        }
52    }
```

- **playGame**

Funcionalitat: Funció que tracta sobre el bucle principal del joc. Controla el flux del joc trucant a les diferents funcions implementades en el model en el moment concret. La partida no acaba fins que un jugador hagi completat tot el recorregut amb totes les peces.

Localització: src/main/controller/GameController.java, Classe: GameController, Mètode: playGame()

Test: src/test/java/test/controller/GameControllerTest.java, Classe: GameControllerTest

Tests:

➢ **Test Complex:** Mètode: testPlayGameUntilYellowWins()

Hem realitzat un test complex en què simulem la interacció del model i el controlador amb la vista. Com que el joc es juga mitjançant entrades introduïdes pels jugadors, hem creat diverses classes per simular aquesta situació.

- **Classe UserInput (src/main/model/UserInput.java):** Ens permet proporcionar un conjunt d'entrades predefinides, simulant les accions dels jugadors en temps real.

En aquest test, simulem una partida completa entre dos jugadors (groc i blau). Per controlar el resultat del dau, que és aleatori, hem creat un mock object per simular les tirades de dau dels jugadors, assegurant que el jugador groc guanyi la partida. També hem creat un mock object per a la classe GameView per gestionar les sortides per pantalla sense necessitat d'entrades manuals.

Durant aquest test, també es verifica el funcionament correcte de les funcions chooseToEnterPiece i choosePiece de la classe Player.

➢ **Test Complex:** Mètode: testPlayerHasNoMovablePieces()

En el test anterior no es van cobrir tots els casos possibles. Per això, hem creat aquest test específic per al cas en què un jugador no té peces que pugui moure. Utilitzem el mock del Die per simular que el jugador només obté tirades de 3, i la classe UserInput per gestionar les entrades. Comprovem que el joc gestiona adequadament aquesta situació, informant el jugador que no té moviments disponibles.

Caixa blanca:

➢ **Loop Testing Aniuat:** Mètode: testPlayGame_loopTesting()

Hem realitzat un loop testing sobre el bucle principal del joc, que conté un bucle aniuat. Per dur a terme aquest test, hem creat un mock object de la classe GameController amb una implementació modificada del mètode playGame(), limitant el nombre de torns. En la nova implementació del mètode playGame() no hem ficat la lògica del joc, ja que acabava sent un prova bastant complexa. Amb aquest test, assegurem que el bucle principal del joc funciona correctament sota diferents condicions i nombres d'iteracions.

Per fer el loop testing aniuat hem seguit la regla de començar amb un test simple pel loop més interior, fixant els altres loops al valor mínim. Testejar un loop més extern (com si fos un loop simple) mantenint el nombre d'iteracions dels loops interiors a valors habituals.

```
MAX_TURNS = 1000 (for n)
MAX_PLAYERS = 4 (for m)
// (n,m) = (1,0) (1,1) (1,2) (1,MAX_PLAYERS-1) (1,MAX_PLAYERS) (0,m<MAX_PLAYERS)
(1,m<MAX_PLAYERS) (2,m<MAX_PLAYERS) (n<MAX_TURNS-1,m<MAX_PLAYERS)
(MAX_TURNS-1,m<MAX_PLAYERS) (MAX_TURNS,m<MAX_PLAYERS)
```

```java
@Override  4 usages  YssfDevOps
public void playGame() {
  boolean gameWon = false;
  int turnCount = 0;

  while (turnCount < maxTurns) {
    for (Player player : getPlayers()) {
      totalPlayerActions++;
      // We won't simulate game logic (too complex).
    }
    turnCount++;
    actualTurns++;
  }
 }
}
```

Mock objects:

- **MockDie:** Mètode: testPlayGameUntilYellowWins() i testPlayerHasNoMovablePieces()

Aquest mock ens permet definir tirades de dau predefinides per simular situacions específiques durant la partida.

- **MockGameController:** Mètode: testPlayGame_loopTesting()

Hem creat aquest mock per modificar el bucle principal del joc, limitant el nombre de torns i facilitant el loop testing aniuat.

- **MockGameView:** Mètode: testPlayGameUntilYellowWins() i testPlayerHasNoMovablePieces()

Aquest mock ens permet simular les sortides per pantalla, evitant la necessitat d'interacció manual durant els tests.

- playerRollDie

Funcionalitat: Funció que truca a la vista per demanar al jugador una entrada (prèmer enter), donat que el jugador ha de tirar el dau. Aquesta funció retorna un número entre 1 i 6.

Localització: src/main/controller/GameController.java, Classe: GameController, Mètode: playerRollDie(Player player)

Test: src/test/java/test/controller/GameControllerTest.java, Classe: GameControllerTest, Mètode: playerRollDie()

Tests:

- **Test Simple:**

Hem realitzat un test en el qual llencem el dau 100 vegades i comprovem que tots els resultats estan entre 1 i 6.

GAMEVIEW.JAVA

Aquesta classe GameView és la que implementa la vista del nostre joc. Els tests realitzats pels mètodes d'aquesta classe són la verificació que no es produeixen excepcions o que es produeixen per a casos concrets.

- **showWelcomeMessage**

Funcionalitat: Funció que mostra el missatge de benvinguda al joc.

Localització: src/main/view/GameView.java, Classe: GameView, Mètode: showWelcomeMessage()

Test: src/test/java/test/view/GameViewTest.java, Classe: GameViewTest, Mètode: showWelcomeMessage()

- **getNumberOfPlayers**

Funcionalitat: Demana al jugador que introdueixi el nombre de jugadors (entre 2 i 4) i el retorna. Si el número és fora del rang, torna a demanar-lo fins que sigui vàlid.
Localització: src/main/view/GameView.java, Classe: GameView, Mètode: getNumberOfPlayers()

Test: src/test/java/test/view/GameViewTest.java, Classe: GameViewTest, Mètode: getNumberOfPlayers()

- **getPlayerName**

Funcionalitat: Demana al jugador que introdueixi el nom d'un jugador donat un número de jugador (p. ex., “Enter name for player 1”) i retorna el nom com a String.

Localització: src/main/view/GameView.java, Classe: GameView, Mètode: getPlayerName(int playerNumber)

Test: src/test/java/test/view/GameViewTest.java, Classe: GameViewTest, Mètode: getPlayerName()

- **showBoard**

Funcionalitat: Mostra l'estat actual del joc a la consola. Per cada jugador a la llista, mostra el nom, el color i les peces.

Localització: src/main/view/GameView.java, Classe: GameView, Mètode: showBoard(List<Player> players)

Test: src/test/java/test/view/GameViewTest.java, Classe: GameViewTest, Mètode: showBoard()

- **showPlayerTurn**

Funcionalitat: Funció que mostra de qui és el torn actual, indicant el nom i el color del jugador, així com l'estat de les seves peces.

Localització: src/main/view/GameView.java, Classe: GameView, Mètode: showPlayerTurn(Player player)

Test: src/test/java/test/view/GameViewTest.java, Classe: GameViewTest, Mètode: showPlayerTurn()

- **promptRollDie**

Funcionalitat: Demana al jugador actual que premeixi "Enter" per tirar el dau.

Localització: src/main/view/GameView.java, Classe: GameView, Mètode: promptRollDie(Player player)

Test: src/test/java/test/view/GameViewTest.java, Classe: GameViewTest, Mètode: promptRollDie()

- **showDieRoll**

Funcionalitat: Mostra el valor del dau que ha tret un jugador.

Localització: src/main/view/GameView.java, Classe: GameView, Mètode: showDieRoll(Player player, int roll)

Test: src/test/java/test/view/GameViewTest.java, Classe: GameViewTest, Mètode: showDieRoll()

- **showNoMovablePieces**

Funcionalitat: Informa que el jugador actual no té peces movibles en el seu torn.

Localització: src/main/view/GameView.java, Classe: GameView, Mètode: showNoMovablePieces(Player player)

Test: src/test/java/test/view/GameViewTest.java, Classe: GameViewTest, Mètode: showNoMovablePieces()

- **showWinner**

Funcionalitat: Mostra un missatge felicitant el guanyador del joc.

Localització: src/main/view/GameView.java, Classe: GameView, Mètode: showWinner(Player player)

Test: src/test/java/test/view/GameViewTest.java, Classe: GameViewTest, Mètode: showWinner()

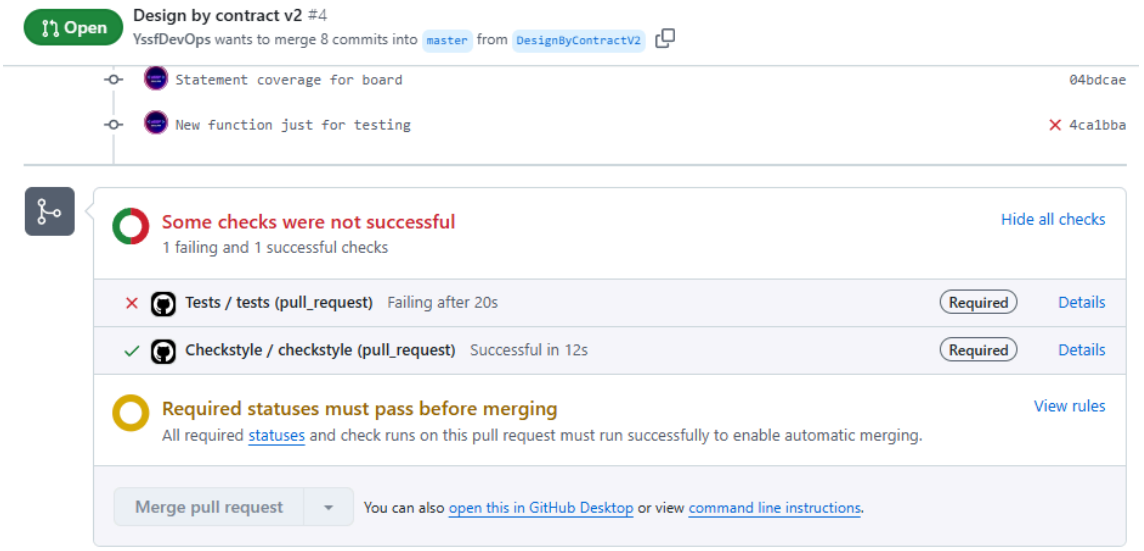
CI del projecte:

Per portar a terme el CI del projecte hem creat una carpeta anomenada /.github/workflows, on tenim dos arxius de tipus **yml**. Aquests dos arxius s'encarreguen de comprovar que tot el codi implementat estigui correcte. Els arxius són els següents:

- checkstyle.yml: Comprova que la forma en la qual està implementat el codi segueix la normativa de codi.
- tests.yml: Comprova que tots els tests que hi ha a la carpeta de tests donin correcte.

En aquest cas, hem implementat un ruleset en la branca master la qual ens impedeix fer push directament, de forma que s'ha de crear una branca nova i fer la implementació aquí. Per poder fer un merge d'una branca nova amb la branca master, s'ha de complir que tant el checkstyle com els tests es completin de manera satisfactòria. Si no es completen no ens deixa fer un merge, però si es compleixen ens deixa.

- Cas de tests malament, no ens deixa fer merge:



- Cas on tot és correcte, podem fer merge:

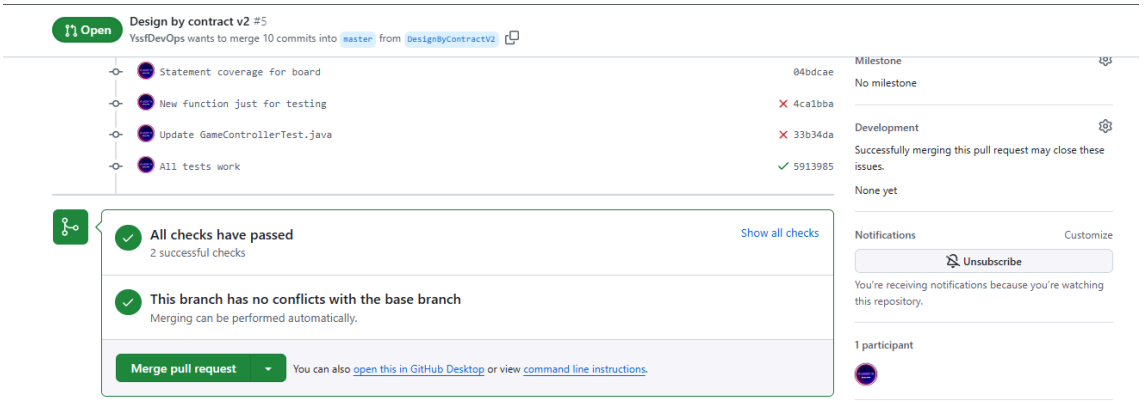


Diagrama de classes UML: